teletype - manual

Contents

Introduction **3**

Quickstart **4**
- Panel . . . 4
- LIVE mode . . . 5
- EDIT mode . . . 6
- Patterns . . . 7
- Scenes . . . 9
- USB Backup . . . 10
- Commands . . . 10
- Continuing . . . 12

Keys **13**
- Global key bindings . . . 13
- Text editing . . . 14
- Live mode . . . 15
- Edit mode . . . 16
- Tracker mode . . . 17
- Preset read mode . . . 18
- Preset write mode . . . 19
- Help mode . . . 19

OPs and MODs **20**
- Variables . . . 21
- Hardware . . . 24
- Pitch . . . 28
- Rhythm . . . 34
- Metronome . . . 37
- Randomness . . . 38
- Control flow . . . 40
- Maths . . . 49
- Delay . . . 53
- Stack . . . 55
- Patterns . . . 56
- Queue . . . 62
- Turtle . . . 68
- Grid . . . 69

MIDI in . . . 85
Calibration . . . 87
Generic I2C . . . 90
Ansible . . . 92
Just Type . . . 96
Remote control . . . 97
Panel queries . . . 100
Synthesis mode . . . 101
Geode . . . 104
Just Friends reference . . . 109
16n . . . 112
TELEXi . . . 113
TELEXo . . . 117
Crow . . . 132
W/ . . . 134
W/2.0 . . . 135
W/2.0 tape . . . 136
W/2.0 delay . . . 138
W/2.0 synth . . . 139
i2c2midi . . . 141

Advanced **161**
Teletype terminology . . . 161
Sub commands . . . 162
Aliases . . . 163
Avoiding non-determinism . . . 164

Grid integration **165**
Grid operators . . . 167
Grid studies . . . 173
Grid visualizer . . . 186
Grid control mode . . . 188

Alphabetical list of OPs and MODs **197**

Introduction

Teletype is a dynamic, musical event triggering platform.

- Teletype Studies[1] - guided series of tutorials
- PDF command reference chart[2] — PDF scene recall sheet[3] — Default scenes[4]
- Current version: *5.0.0* — Firmware update procedure[5]

[1] https://monome.org/docs/modular/teletype/studies-1
[2] https://monome.org/docs/teletype/TT_commands_3.0.pdf
[3] https://monome.org/docs/teletype/TT_scene_RECALL_sheet.pdf
[4] http://monome.org/docs/teletype/scenes-10/
[5] https://monome.org/docs/modular/update/

Quickstart

Panel

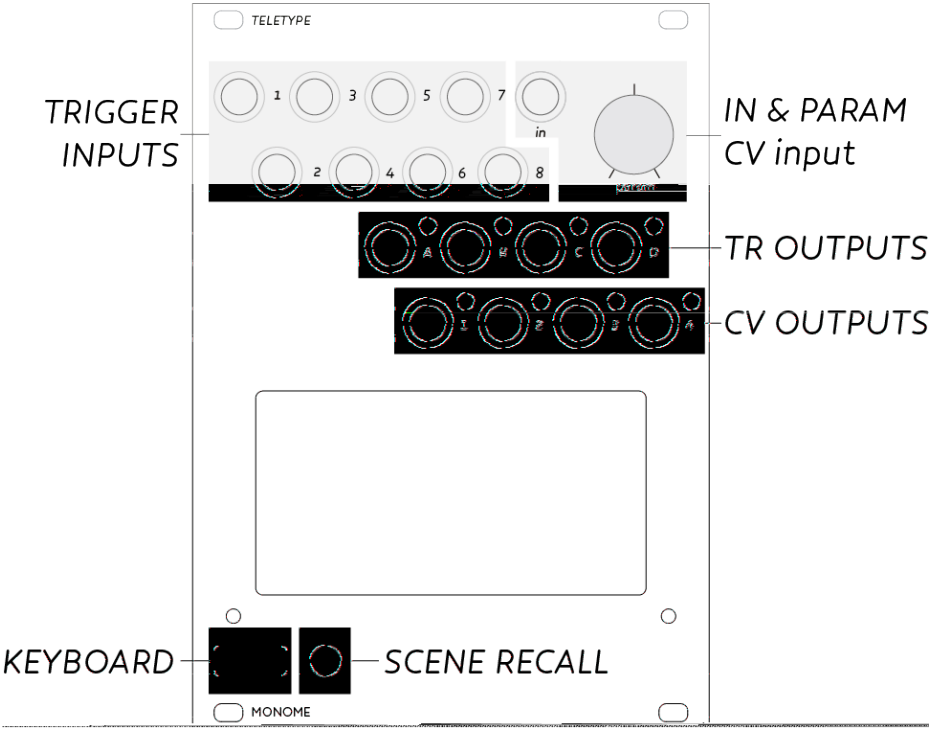


Figure 1: Panel Overlay

The keyboard is attached to the front panel, for typing commands. The commands can be executed immediately in *LIVE mode* or assigned to one of the eight trigger inputs in *EDIT mode*. The knob and in jack can be used to set and replace values.

LIVE mode

Teletype starts up in LIVE mode. You'll see a friendly **>** prompt, where commands are entered. The command:

```
TR.TOG A
```

will toggle trigger A after pressing enter. Consider:

```
CV 1 V 5
CV 2 N 7
CV 1 0
```

Here the first command sets CV 1 to 5 volts. The second command sets CV 2 to note 7 (which is 7 semitones up). The last command sets CV 1 back to 0.

Data flows from right to left, so it becomes possible to do this:

```
CV 1 N RAND 12
```

Here a random note between 0 and 12 is set to CV 1.

We can change the behavior of a command with a *PRE* such as **DEL**:

```
DEL 500 : TR.TOG A
```

TR.TOG A will be delayed by 500ms upon execution.

A helpful display line appears above the command line in dim font. Here any entered commands will return their numerical value if they have one.

SCRIPTS, or several lines of commands, can be assigned to trigger inputs. This is when things get musically interesting. To edit each script, we shift into EDIT mode.

LIVE mode icons

Four small icons are displayed in LIVE mode to give some important feedback about the state of Teletype. These icons will be brightly lit when the above is true, else will remain dim. They are, from left to right:

- Slew: CV outputs are currently slewing to a new destination.
- Delay: Commands are in the delay queue to be executed in the future.
- Stack: Commands are presently on the stack waiting for execution.
- Metro: Metro is currently active and the Metro script is not empty.

EDIT mode

Toggle between EDIT and LIVE modes by pushing **TAB**.

The prompt now indicates the script you're currently editing:

- **1-8** indicates the script associated with corresponding trigger
- **M** is for the internal metronome
- **I** is the init script, which is executed upon scene recall

Script 1 will be executed when trigger input 1 (top left jack on the panel) receives a low-to-high voltage transition (trigger, or front edge of a gate). Consider the following as script 1:

1:

```
TR.TOG A
```

Now when input 1 receives a trigger, **TR.TOG A** is executed, which toggles the state of output trigger A.

Scripts can have multiple lines:

1:

```
TR.TOG A
CV 1 P RAND 4
```

Now each time input 1 receives a trigger, CV 1 is set to a random volt between 0 and 4, in addition to output trigger A being toggled.

Metronome

The **M** script is driven by an internal metronome, so no external trigger is required. By default the metronome interval is 1000ms. You can change this readily (for example, in LIVE mode):

```
M 500
```

The metronome interval is now 500ms. You can disable/enable the metronome entirely with **M.ACT**:

```
M.ACT 0
```

Now the metronome is off, and the **M** script will not be executed. Set **M.ACT** to 1 to re-enable.

Patterns

Patterns facilitate musical data manipulation– lists of numbers that can be used as sequences, chord sets, rhythms, or whatever you choose. Pattern memory consists four banks of 64 steps. Functions are provided for a variety of pattern creation, transformation, and playback. The most basic method of creating a pattern is by directly adding numbers to the sequence:

```
P.PUSH 5
P.PUSH 11
P.PUSH 9
P.PUSH 3
```

P.PUSH adds the provided value to the end of the list– patterns keep track of their length, which can be read or modified with **P.L**. Now the pattern length is 4, and the list looks something like:

```
5, 11, 9, 3
```

Patterns also have an index **P.I**, which could be considered a playhead. **P.NEXT** will advance the index by one, and return the value stored at the new index. If the playhead hits the end of the list, it will either wrap to the beginning (if **P.WRAP** is set to 1, which it is by default) or simply continue reading at the final position.

So, this script on input 1 would work well:

1:

```
CV 1 N P.NEXT
```

Each time input 1 is triggered, the pattern moves forward one then CV 1 is set to the note value of the pattern at the new index. This is a basic looped sequence. We could add further control on script 2:

2:

```
P.I 0
```

Since **P.I** is the playhead, trigger input 2 will reset the playhead back to zero. It won't change the CV, as that only happens when script 1 is triggered.

We can change a value within the pattern directly:

```
P 0 12
```

This changes index 0 to 12 (it was previously 5), so now we have *12, 11, 9, 3*.

We've been working with pattern **0** up to this point. There are four pattern banks, and we can switch banks this way:

```
P.N 1
```

Now we're on pattern bank 1. **P.NEXT**, **P.PUSH**, **P**, (and several more commands) all reference the current pattern bank. Each pattern maintains its own play index,

Scenes

A *SCENE* is a complete set of scripts and patterns. Stored in flash, scenes can be saved between sessions. Many scenes ship as examples. On startup, the last used scene is loaded by Teletype.

Access the SCENE menu using ESCAPE. The bracket keys ([and]) navigate between the scenes. Use the up/down arrow keys to read the scene *text*. This text will/should describe what the scene does generally along with input/output functions. ENTER will load the selected scene, or ESCAPE to abort.

To save a scene, hold ALT while pushing ESCAPE. Use the brackets to select the destination save position. Edit the text section as usual– you can scroll down for many lines. The top line is the name of the scene. ALT-ENTER will save the scene to flash.

Keyboard-less Scene Recall

To facilitate performance without the need for the keyboard, scenes can be recalled directly from the module's front panel.

- Press the *SCENE RECALL* button next to the USB jack on the panel.
- Use the *PARAM* knob to highlight your desired preset.
- Hold the *SCENE RECALL* button for 1 second to load the selected scene.

Init Script

The *INIT* script (represented as `I`) is executed when a preset is recalled. This is a good place to set initial values of variables if needed, like metro time `M` or time enable `TIME.ACT` for example.

USB Backup

Teletype's scenes can be saved and loaded from a USB flash drive. When a flash drive is inserted, Teletype will recognize it and go into disk mode. First, all 32 scenes will be written to text files on the drive with names of the form 'tt##s.txt'. For example, scene 5 will be saved to 'tt05s.txt'. The screen will display **WRITE.....** as this is done.

Once complete, Teletype will attempt to read any files named 'tt##.txt' and load them into memory. For example, a file named 'tt13.txt' would be loaded as scene 13 on Teletype. The screen will display **READ.....** Once this process is complete, Teletype will return to LIVE mode and the drive can be safely removed.

For best results, use an FAT-formatted USB flash drive. If Teletype does not recognize a disk that is inserted within a few seconds, it may be best to try another.

An example of possible scenes to load, as well as the set of factory default scenes, can be found at the Teletype Codex[6].

Commands

Nomenclature

- SCRIPT – multiple *commands*
- COMMAND – a series (one line) of *words*
- WORD – a text string separated by a space: *value*, *operator*, *variable*, *mod*
- VALUE – a number
- OPERATOR – a function, may need value(s) as argument(s), may return value
- VARIABLE – named memory storage
- MOD – condition/rule that applies to rest of the *command*, e.g.: del, prob, if, s

Syntax

Teletype uses prefix notation. Evaluation happens from right to left.

The left value gets assignment (*set*). Here, temp variable **X** is assigned zero:

```
X 0
```

Temp variable **Y** is assigned to the value of **X**:

```
Y X
```

[6] https://github.com/monome-community/teletype-codex

X is being *read* (*get* X), and this value is being used to *set* Y.

Instead of numbers or variables, we can use operators to perform more complex behavior:

```
X TOSS
```

TOSS returns a random state, either 0 or 1 on each call.

Some operators require several arguments:

```
X ADD 1 2
```

Here ADD needs two arguments, and gets 1 and 2. X is assigned the result of ADD, so X is now 3.

If a value is returned at the end of a command, it is printed as a MESSAGE. This is visible in LIVE mode just above the command prompt. (In the examples below ignore the // comments).

```
8           // PRINTS 8
X 4
X           // PRINTS 4
ADD 8 32    // PRINTS 40
```

Many parameters are indexed, such as CV and TR. This means that CV and TR have multiple values (in this case, each has four.) We pass an extra argument to specify which index we want to read or write.

```
CV 1 0
```

Here CV 1 is set to 0. You can leave off the 0 to print the value.

```
CV 1        // PRINTS VALUE OF CV 1
```

Or, this works too:

```
X CV 1      // SET X TO CURRENT VALUE OF CV 1
```

Here is an example of using an operator RAND to set a random voltage:

```
CV 1 V RAND 4
```

First a random value between 0 and 3 is generated. The result is turned into a volt with a table lookup, and the final value is assigned to CV 1.

The order of the arguments is important, of course. Consider:

```
CV RRAND 1 4 0
```

RRAND uses two arguments, 1 and 4, returning a value between these two. This command, then, chooses a random CV output (1-4) to set to 0. This might seem confusing, so it's possible to clarify it by pulling it apart:

```
X RRAND 1 4
CV X 0
```

Here we use `X` as a temp step before setting the final CV.

With some practice it becomes easier to combine many functions into the same command.

Furthermore, you can use a semicolon to include multiple commands on the same line:

```
X RRAND 1 4; CV X 0
```

This is particularly useful in **INIT** scripts where you may want to initialize several values at once:

```
A 66; X 101; TR.TIME 1 20;
```

Continuing

Don't forget to checkout the Teletype Studies[7] for an example-driven guide to the language.

[7] https://monome.org/docs/modular/teletype/studies-1

Keys

Global key bindings

These bindings work everywhere.

Key	Action
tab	change modes, live to edit to pattern and back
esc	preset read mode, or return to last mode
alt-esc	preset write mode
win-esc	clear delays, stack and slews
shift-alt-? / **alt-h**	help text, or return to last mode
F1 to **F8**	run corresponding script
F9	run metro script
F10	run init script
alt-F1 to **alt-F8**	edit corresponding script
alt-F9	edit metro script
alt-F10	edit init script
ctrl-F1 to **ctrl-F8**	mute/unmute corresponding script
ctrl-F9	enable/disable metro script
numpad-1 to **numpad-8**	run corresponding script
num lock / **F11**	jump to pattern mode
print screen / **F12**	jump to live mode

Text editing

These bindings work when entering text or code.

In most cases, the clipboard is shared between *live*, *edit* and the 2 *preset* modes.

Key	Action
left / **ctrl-b**	move cursor left
right / **ctrl-f**	move cursor right
ctrl-left / **alt-b**	move left by one word
ctrl-right / **alt-f**	move right by one word
home / **ctrl-a**	move to beginning of line
end / **ctrl-e**	move to end of line
backspace / **ctrl-h**	backwards delete one character
delete / **ctrl-d**	forwards delete one character
shift-backspace / **ctrl-u**	delete from cursor to beginning
shift-delete / **ctrl-k**	delete from cursor to end
alt-backspace / **ctrl-w**	delete from cursor to beginning of word
alt-d	delete from cursor to end of word
ctrl-x / **alt-x**	cut to clipboard
ctrl-c / **alt-c**	copy to clipboard
ctrl-v / **alt-v**	paste to clipboard

Live mode

Key	Action
down / **C-n**	history next
up / **C-p**	history previous
enter	execute command
~	toggle variables
[/]	switch to edit mode
alt-g	toggle grid visualizer
shift-d	live dashboard
alt-arrows	move grid cursor
alt-shift-arrows	select grid area
alt-space	emulate grid press
alt-/	switch grid pages
alt-	toggle grid control view
alt-prt sc	insert grid x/y/w/h

In full grid visualizer mode pressing alt is not required.

Edit mode

In *edit* mode multiple lines can be selected and used with the clipboard.

Key	Action
down / **C-n**	line down
up / **C-p**	line up
[	previous script
]	next script
enter	enter command
shift-enter	insert command
alt-/	toggle line comment
shift-up	expand selection up
shift-down	expand selection down
alt-delete	delete selection
alt-up	move selection up
alt-down	move selection down
ctrl-z	undo (3 levels)

Tracker mode

The tracker mode clipboard is independent of text and code clipboard.

Key	Action
down	move down
alt-down	move a page down
up	move up
alt-up	move a page up
left	move left
alt-left	move to the very left
right	move right
alt-right	move to the very right
[	decrement by 1
]	increment by 1
alt-[	decrement by 1 semitone
alt-]	increment by 1 semitone
ctrl-[	decrement by 7 semitones
ctrl-]	increment by 7 semitones
shift-[	decrement by 12 semitones
shift-]	increment by 12 semitones
alt-0-9	increment by 0-9 semitones (0=10, 1=11)
shift-alt-0-9	decrement by 0-9 semitones (0=10, 1=11)
backspace	delete a digit
shift-backspace	delete an entry, shift numbers up
enter	commit edit (increase length if cursor in position after last entry)
shift-enter	commit edit, then duplicate entry and shift downwards (increase length as enter)
alt-x	cut value (n.b. ctrl-x not supported)
alt-c	copy value (n.b. ctrl-c not supported)
alt-v	paste value (n.b. ctrl-v not supported)
shift-alt-v	insert value
shift-l	set length to current position
alt-l	go to current length entry
shift-s	set start to current position
alt-s	go to start entry
shift-e	set end to current position

Key	Action
alt-e	go to end entry
-	negate value
space	toggle non-zero to zero, and zero to 1
0 to **9**	numeric entry
shift-2 (@)	toggle turtle display marker (<)
ctrl-alt	insert knob value scaled to 0..31
ctrl-shift	insert knob value scaled to 0..1023

Preset read mode

Key	Action
down / **C-n**	line down
up / **C-p**	line up
left / **[**	preset down
right / **]**	preset up
enter	load preset

Preset write mode

Key	Action
down / **C-n**	line down
up / **C-p**	line up
[	preset down
]	preset up
enter	enter text
shift-enter	insert text
alt-enter	save preset

Help mode

Key	Action
down / **C-n**	line down
up / **C-p**	line up
left / **[**	previous page
right / **]**	next page
C-f / **C-s**	search forward
C-r	search backward

OPs and MODs

Variables

General purpose temp vars: **X**, **Y**, **Z**, and **T**.

T typically used for time values, but can be used freely.

A-D are assigned 1-4 by default (as a convenience for TR labeling, but TR can be addressed with simply 1-4). All may be overwritten and used freely.

OP *(set)*	*(aliases)*	Description
A (x)		get / set the variable **A**, default 1
B (x)		get / set the variable **B**, default 2
C (x)		get / set the variable **C**, default 3
D (x)		get / set the variable **D**, default 4
FLIP (x)		returns the opposite of its previous state (0 or 1) on each read (also settable)
I (x)		get / set the per-script variable **I**. *See also* **L:** *in control flow*
J (x)		get / set the per-script variable **J**
K (x)		get / set the per-script variable **K**
O (x)		auto-increments *after* each access, can be set, starting value 0
O.INC (x)		how much to increment **O** by on each invocation, default 1
O.MIN (x)		the lower bound for **O**, default 0
O.MAX (x)		the upper bound for **O**, default 63
O.WRAP (x)		should **O** wrap when it reaches its bounds, default 1
T (x)		get / set the variable **T**, typically used for time, default 0
TIME (x)		timer value, counts up in ms., wraps after 32s, can be set
TIME.ACT (x)		enable or disable timer counting, default 1
LAST x		get value in milliseconds since last script run time
X (x)		get / set the variable **X**, default 0
Y (x)		get / set the variable **Y**, default 0
Z (x)		get / set the variable **Z**, default 0

I

- I (x)

Get / set the variable **I**. This variable is overwritten by **L**, but can be used freely outside an **L** loop. Each script gets its own **I** variable, so if you call a script from another script's loop you can still use and modify **I** without affecting the calling loop. In this scenario the script getting called will have its **I** value initialized with the calling loop's current **I** value.

]

-] (x)

get / set the variable **]**, each script gets its own **]** variable, so if you call a script from another script you can still use and modify **]** without affecting the calling script.

K

- K (x)

get / set the variable **K**, each script gets its own **K** variable, so if you call a script from another script you can still use and modify **K** without affecting the calling script.

0

- 0 (x)

Auto-increments by **0.INC** *after* each access. The initial value is 0. The lower and upper bounds can be set by **0.MIN** (default 0) and **0.MAX** (default 63). **0.WRAP** controls if the value wraps when it reaches a bound (default is 1).

Example:

```
0           => 0
0           => 1
X 0
X           => 2
0.INC 2
0           => 3 (0 INCREMENTS AFTER IT'S ACCESSED)
0           => 5
0.INC -2
0 2
0           => 2
```

```
0           => 0
0           => 63
0           => 61
```

LAST

- `LAST x`

Gets the number of milliseconds since the given script was run, where `M` is script 9 and `I` is script 10. From the live mode, `LAST SCRIPT` gives the time elapsed since last run of I script.

For example, one-line tap tempo:

```
M LAST SCRIPT
```

Running this script twice will set the metronome to be the time between runs.

Hardware

The Teletype trigger inputs are numbered 1-8, the CV and trigger outputs 1-4. See the Ansible documentation for details of the Ansible output numbering when in Teletype mode.

OP *(set)*	*(aliases)*	Description
CV x (y)		CV target value
CV.OFF x (y)		CV offset added to output
CV.SET x y		Set CV value, ignoring slew
CV.GET x		Get current CV value
CV.SLEW x (y)		Get/set the CV slew time in ms
V x		converts a voltage to a value usable by the CV outputs (x between 0 and 10)
VV x		converts a voltage to a value usable by the CV outputs (x between 0 and 1000, 100 represents 1V)
IN		Get the value of IN jack (0-16383)
IN.SCALE MIN MAX		Set static scaling of the IN CV to between MIN and MAX.
PARAM	PRM	Get the value of PARAM knob (0-16383)
PARAM.SCALE min max		Set static scaling of the PARAM knob to between MIN and MAX.
TR x (y)		Set trigger output x to y (0-1)
TR.PULSE x	TR.P	Pulse trigger output x
TR.TIME x (y)		Set the pulse time of trigger x to y ms
TR.TOG x		Flip the state of trigger output x
TR.POL x (y)		Set polarity of trigger output x to y (0-1)
MUTE x (y)		Disable trigger input x
STATE x		Read the current state of input x
LIVE.OFF	LIVE.O	Show the default live mode screen
LIVE.VARS	LIVE.V	Show variables in live mode
LIVE.GRID	LIVE.G	Show grid visualizer in live mode
LIVE.DASH x	LIVE.D	Show the dashboard with index x
PRINT x (y)	PRT	Print a value on a live mode dashboard or get the printed value

CV

- `CV x (y)`

Get the value of CV associated with output `x`, or set the CV output of `x` to `y`.

CV.OFF

- `CV.OFF x (y)`

Get the value of the offset added to the CV value at output `x`. The offset is added at the final stage. Set the value of the offset added to the CV value at output `x` to `y`.

CV.SET

- `CV.SET x y`

Set the CV value at output `x` bypassing any slew settings.

CV.GET

- `CV.GET x`

Get the current CV value at output `x` with slew and offset applied.

CV.SLEW

- `CV.SLEW x (y)`

Get the slew time in ms associated with CV output `x`. Set the slew time associated with CV output `x` to `y` ms.

IN

- `IN`

Get the value of the IN jack. This returns a valuue in the range 0-16383.

PARAM

- `PARAM`
- *alias*: `PRM`

Get the value of the PARAM knob. This returns a valuue in the range 0-16383.

TR

- **TR x (y)**

Get the current state of trigger output **x**. Set the state of trigger output **x** to **y** (0-1).

TR.PULSE

- **TR.PULSE x**
- *alias*: **TR.P**

Pulse trigger output x.

TR.TIME

- **TR.TIME x (y)**

Get the pulse time of trigger output **x**. Set the pulse time of trigger output **x** to **y**ms.

TR.TOG

- **TR.TOG x**

Flip the state of trigger output **x**.

TR.POL

- **TR.POL x (y)**

Get the current polarity of trigger output **x**. Set the polarity of trigger output **x** to **y** (0-1). When TR.POL = 1, the pulse is 0 to 1 then back to 0. When TR.POL = 0, the inverse is true, 1 to 0 to 1.

MUTE

- **MUTE x (y)**

Mute the trigger input on **x** (1-8) when **y** is non-zero.

STATE

- **STATE x**

Read the current state of trigger input **x** (0=low, 1=high).

LIVE.DASH

- **LIVE.DASH x**
- *alias*: **LIVE.D**

This allows you to show custom text and print values on the live mode screen. To create a dashboard, simply edit the scene description. You can define multiple dashboards by separating them with **===**, and you can select them by specifying the dashboard number as the op parameter.

You can also print up to 16 values using **PRINT** op. To create a placeholder for a value, place **%##** where you want the number to be, where **##** is a value index between 1 and 16. Please note: if you define multiple placeholders for the same value, only the last one will be used, and the rest will be treated as plain text. By default, values are printed in decimal format, but you can also use hex, binary and reversed binary formats by using **%X##**, **%B##** and **%R##** placeholders respectively.

An example of a dashboard:

```
THIS IS A DASHBOARD

CURRENT METRO RATE IS: %1
```

You can use this dashboard by entering the above in a scene description, placing **LIVE.DASH 1** in the init script and placing **PRINT 1 M** in the metro script.

PRINT

- **PRINT x (y)**
- *alias*: **PRT**

This op allows you to display up to 16 values on a live mode dashboard and should be used in conjunction with **LIVE.DASH** op. See **LIVE.DASH** description for information on how to use it. You can also use this op to store up to 16 additional values.

Pitch

Mathematical calcuations and tables helpful for musical pitch.

OP *(set)*	*(aliases)*	Description
`HZ x`		converts 1V/OCT value `x` to Hz/Volt value, useful for controlling non-euro synths like Korg MS-20
`]I x y`		just intonation helper, precision ratio divider normalised to 1V
`N x`		converts an equal temperament note number to a value usable by the CV outputs (`x` in the range -127 to 127)
`N.S r s d`		Note Scale operator, `r` is the root note (0-127), `s` is the scale (0-8) and `d` is the degree (1-7), returns a value from the `N` table.
`N.C r c d`		Note Chord operator, `r` is the root note (0-127), `c` is the chord (0-12) and `d` is the degree (0-3), returns a value from the `N` table.
`N.CS r s d c`		Note Chord Scale operator, `r` is the root note (0-127), `s` is the scale (0-8), `d` is the scale degree (1-7) and `c` is the chord component (0-3), returns a value from the `N` table.
`N.B d (s)`		get degree `d` of scale/set scale root to `r`, scale to `s`, `s` is either bit mask (`s` >= 1) or scale preset (`s` < 1)
`N.BX i d (s)`		multi-index version of N.B, scale at `i` (index) 0 is shared with N.B
`VN x`		converts 1V/OCT value `x` to an equal temperament note number
`QT.B x`		quantize 1V/OCT signal `x` to scale defined by `N.B`
`QT.BX i x`		quantize 1V/OCT signal `x` to scale defined by `N.BX` in scale index `i`
`QT.S x r s`		quantize 1V/OCT signal `x` to scale `s` (0-8, reference N.S scales) with root 1V/OCT pitch `r`

OP *(set)*	*(aliases)*	Description
QT.CS x r s d c		quantize 1V/OCT signal x to chord c (1-7) from scale s (0-8, reference N.S scales) at degree d (1-7) with root 1V/OCT pitch r

M

- M x

The M OP converts an equal temperament note number to a value usable by the CV outputs.

Examples:

```
CV 1 M 60        => SET CV 1 TO MIDDLE C, I.E. 5V
CV 1 M RAND 24   => SET CV 1 TO A RANDOM NOTE FROM THE LOWEST 2 OCTAVES
```

M.S

- M.S r s d

The M.S OP lets you retrieve M table values according to traditional western scales. s and d wrap to their ranges automatically and support negative indexing.

Scales

- 0 = Major
- 1 = Natural Minor
- 2 = Harmonic Minor
- 3 = Melodic Minor
- 4 = Dorian
- 5 = Phrygian
- 6 = Lydian
- 7 = Mixolydian
- 8 = Locrian

M.C

- M.C r c d

The M.C OP lets you retrieve M table values according to traditional western chords. c and d wrap to their ranges automatically and support negative indexing.

Chords

- 0 = Major 7th {0, 4, 7, 11}
- 1 = Minor 7th {0, 3, 7, 10}
- 2 = Dominant 7th {0, 4, 7, 10}
- 3 = Diminished 7th {0, 3, 6, 9}
- 4 = Augmented 7th {0, 4, 8, 10}
- 5 = Dominant 7b5 {0, 4, 6, 10}
- 6 = Minor 7b5 {0, 3, 6, 10}
- 7 = Major 7#5 {0, 4, 8, 11}
- 8 = Minor Major 7th {0, 3, 7, 11}
- 9 = Diminished Major 7th {0, 3, 6, 11}
- 10 = Major 6th {0, 4, 7, 9}
- 11 = Minor 6th {0, 3, 7, 9}
- 12 = 7sus4 {0, 5, 7, 10}

M.CS

- M.CS r s d c

The M.CS OP lets you retrieve M table values according to traditional western scales and chords. s, c and d wrap to their ranges automatically and support negative indexing.

Chord Scales - Refer to chord indices in M.C OP

- 0 = Major {0, 1, 1, 0, 2, 1, 6}
- 1 = Natural Minor {1, 6, 0, 1, 1, 0, 2}
- 2 = Harmonic Minor {8, 6, 7, 1, 2, 0, 3}
- 3 = Melodic Minor {8, 1, 7, 2, 2, 6, 6}

- 4 = Dorian {1, 1, 0, 2, 1, 6, 0}
- 5 = Phrygian {1, 0, 2, 1, 6, 0, 1}
- 6 = Lydian {0, 2, 1, 6, 0, 1, 1}
- 7 = Mixolydian {2, 1, 6, 0, 1, 1, 0}
- 8 = Locrian {6, 0, 1, 1, 0, 2, 1}

N.B

- **N.B d (s)**

Converts a degree in a user-defined equal temperament scale to a value usable by the CV outputs. Default values of `r` and `s` are 0 and R101011010101, corresponding to C-major. To make it easier to generate bit-masks in code, LSB (bit 0) represent the first note in the octave. To avoid having to mirror scales in our heads when entering them by hand, we use `R...` (reverse binary) instead of `B...` (binary).

The bit-masks uses the 12 lower bits.

Note that N.B is using scale at index 0 as used by N.BX ,so N.B and N.BX 0 are equivalent.

Examples:

```
CV 1 N.B 1            ==>  SET CV 1 TO 1ST DEGREE OF DEFAULT SCALE
                           (C, VALUE CORRESPONDING TO N 0)
N.B 0 R101011010101   ==>  SET SCALE TO C-MAJOR (DEFAULT)
CV 1 N.B 1            ==>  SET CV 1 GET 1ST DEGREE OF SCALE
                           (C, VALUE CORRESPONDING TO N 0)
N.B 2 R101011010101   ==>  SET SCALE TO D-MAJOR
CV 1 N.B 3            ==>  SET CV 1 TO 3RD DEGREE OF SCALE
                           (F#, VALUE CORRESPONDING TO N 6)
N.B 3 R100101010010   ==>  SET SCALE TO EB-MINOR PENTATONIC
CV 1 N.B 2            ==>  SET CV 1 TO 2ND DEGREE OF SCALE
                           (GB, VALUE CORRESPONDING TO N 6)
N.B 5 -3              ==>  SET SCALE TO F-LYDIAN USING PRESET
```

Values of `s` less than 1 sets the bit mask to a preset scale:

- 0: Ionian (major)
- -1: Dorian
- -2: Phrygian
- -3: Lydian

- -4: Mixolydian
- -5: Aeolean (natural minor)
- -6: Locrian
- -7: Melodic minor
- -8: Harmonic minor
- -9: Major pentatonic
- -10: Minor pentatonic
- -11 Whole note (1st Messiaen mode)
- -12 Octatonic (half-whole, 2nd Messiaen mode)
- -13 Octatonic (whole-half)
- -14 3rd Messiaen mode
- -15 4th Messiaen mode
- -16 5th Messiaen mode
- -17 6th Messiaen mode
- -18 7th Messiaen mode
- -19 Augmented

M.BX

- `M.BX i d (s)`

Multi-index version of N.B. Index `i` in the range 0-15, allows working with 16 independent scales. Scale at `i` 0 is shared with N.B.

Examples:

```
M.BX 0 0 R101011010101  ==> SET SCALE AT INDEX 0 TO C-MAJOR (DEFAULT)
CV 1 M.BX 0 1           ==>  SET CV 1 TO 1ST DEGREE OF SCALE
                             (C, VALUE CORRESPONDING TO M 0)
M.BX 1 3 R100101010010     ==>   SET SCALE AT INDEX 1 TO EB-
MINOR PENTATONIC
CV 1 M.BX 1 2           ==>  SET CV 1 TO 2ND DEGREE OF SCALE
                             (GB, VALUE CORRESPONDING TO M 6)
M.BX 2 5 -3          ==> SET SCALE AT INDEX 2 TO F-LYDIAN USING PRESET
```

QT.CS

- `QT.CS x r s d c`

Quantize 1V/OCT signal `x` to chord `c` (1-7) from scale `s` (0-8, reference N.S scales) at degree `d` (1-7) with root 1V/OCT pitch `r`.

Chords (1-7):

- 1 = Tonic
- 2 = Third
- 3 = Triad
- 4 = Seventh
- etc.

Rhythm

Mathematical calculations and tables helpful for rhythmic decisions.

OP *(set)*	*(aliases)*	Description
BPM x		milliseconds per beat in BPM x
DR.P b p s		Drum pattern helper, b is the drum bank (0-4), p is the pattern (0-215) and step is the step number (0-15), returns 0 or 1
DR.T b p q l s		Tresillo helper, b is the drum bank (0-4), p is first pattern (0-215), q is the second pattern (0-215), l is length (1-64), and step is the step number (0-length-1), returns 0 or 1
DR.V p s		Velocity helper. p is the pattern (0-19). s is the step number (0-15)
ER f l i		Euclidean rhythm, f is fill (1-32), l is length (1-32) and i is step (any value), returns 0 or 1
NR p m f s		Numeric Repeater, p is prime pattern (0-31), m is & mask (0-3), f is variation factor (0-16) and s is step (0-15), returns 0 or 1

DR.P

- DR.P b p s

The drum helper uses preset drum patterns to give 16-step gate patterns. Gates wrap after step 16. Bank 0 is a set of pseudo random gates increasing in density at higher numbered patterns, where pattern 0 is empty, and pattern 215 is 1s. Bank 1 is bass drum patterns. Bank 2 is snare drum patterns. Bank 3 is closed hi-hats. Bank 4 is open hi-hits and in some cases cymbals. Bank 1-4 patterns are related to each other (bank 1 pattern 1's bass drum pattern fits bank 2 pattern 1's snare drum pattern). The patterns are from Paul Wenzel's "Pocket Operations" book[8].

DR.T

- DR.T b p q l s

[8] https://shittyrecording.studio/

The Tresillo helper uses the preset drum patterns described in the drum pattern help function in a 3, 3, 2 rythmic formation. In the tresillo, pattern 1 will be repeated twice for a number of steps determined by the overall length of the pattern. A pattern of length 8 will play the first three steps of your selected pattern 1 twice, and the first two steps of pattern 2 once. A pattern length of 16 will play the first six steps of selected pattern 1 twice, and the first four steps of pattern 2 once. And so on. The max length is 64. Length will be rounded down to the nearest multiple of 8. The step number wraps at the given length.

DR.V

- `DR.V p s`

The velocity helper gives velocity values (0-16383) at each step. The values are intended to be used for drum hit velocities. There are 16 steps, which wrap around. Divide by 129 to convert to midi cc values.

ER

- `ER f l i`

Euclidean rhythm helper, as described by Godfried Toussaint in his 2005 paper "The Euclidean Algorithm Generates Traditional Musical Rhythms"[9][10]. From the abstract:

- `f` is fill (1-32) and should be less then or equal to length
- `l` is length (1-32)
- `i` is the step index, and will work with negative as well as positive numbers

If you wish to add rotation as well, use the following form:

`ER f l SUB i r`

where `r` is the number of step of *forward* rotation you want.

For more info, see the post on samdoshi.com[11]

MR

- `MR p m f s`

[9] http://cgm.cs.mcgill.ca/~godfried/publications/banff.pdf

[10] Toussaint, G. T. (2005, July). The Euclidean algorithm generates traditional musical rhythms. *In Proceedings of BRIDGES: Mathematical Connections in Art, Music and Science* (pp. 47-56).

[11] http://samdoshi.com/post/2016/03/teletype-euclidean/

Numeric Repeater is similar to ER, except it generates patterns using the binary arithmetic process found in “Noise Engineering’s Numeric Repetitor”[12]. From the description:

Numeric Repetitor is a rhythmic gate generator based on binary arithmetic. A core pattern forms the basis and variation is achieved by treating this pattern as a binary number and multiplying it by another. NR contains 32 prime rhythms derived by examining all possible rhythms and weeding out bad ones via heuristic.

All parameters wrap around their specified ranges automatically and support negative indexing.

Masks - 0 is no mask - 1 is 0x0F0F - 2 is 0xF003 - 3 is 0x1F0

For further detail “see the manual”[13].

[12] https://www.noiseengineering.us/shop/numeric-repetitor

[13] https://static1.squarespace.com/static/58c709192e69cf2422026fa6/t/5e6041ad4cbc0979d6d793f2/1583366574430/NR_manual.pdf

Metronome

An internal metronome executes the M script at a specified rate (in ms). By default the metronome is enabled (**M.ACT 1**) and set to 1000ms (**M 1000**). The metro can be set as fast as 25ms (**M 25**). An additional **M!** op allows for setting the metronome to experimental rates as high as 2ms (**M! 2**). **WARNING**: when using a large number of i2c commands in the M script at metro speeds beyond the 25ms teletype stability issues can occur.

Access the M script directly with alt-F10 or run the script once using F10.

OP *(set)*	*(aliases)*	Description
M (x)		get/set metronome interval to **x** (in ms), default 1000, minimum value 25
M! (x)		get/set metronome to experimental interval **x** (in ms), minimum value 2
M.ACT (x)		get/set metronome activation to **x** (0/1), default 1 (enabled)
M.RESET		hard reset metronome count without triggering

Randomness

OP *(set)*	*(aliases)*	Description
RAND x	RND	generate a random number between 0 and x inclusive
RRAND x y	RRND	generate a random number between x and y inclusive
TOSS		randomly return 0 or 1
R (x)		get a random number/set R.MIN and R.MAX to same value x (effectively allowing R to be used as a global variable)
R.MIN x		set the lower end of the range from -32768 – 32767, default: 0
R.MAX x		set the upper end of the range from -32768 – 32767, default: 16383
CHAOS x		get next value from chaos generator, or set the current value
CHAOS.R x		get or set the R parameter for the CHAOS generator
CHAOS.ALG x		get or set the algorithm for the CHAOS generator. 0 = LOGISTIC, 1 = CUBIC, 2 = HENON, 3 = CELLULAR
DRUNK (x)		changes by -1, 0, or 1 upon each read saving its state, setting will give it a new value for the next read
DRUNK.MIN (x)		set the lower bound for DRUNK, default 0
DRUNK.MAX (x)		set the upper bound for DRUNK, default 255
DRUNK.WRAP (x)		should DRUNK wrap around when it reaches it's bounds, default 0
SEED (x)		get / set the random number generator seed for all SEED ops
RAND.SEED (x)	RAND.SD , R.SD	get / set the random number generator seed for R, RRAND, and RAND ops
TOSS.SEED (x)	TOSS.SD	get / set the random number generator seed for the TOSS op
PROB.SEED (x)	PROB.SD	get / set the random number generator seed for the PROB mod
DRUNK.SEED (x)	DRUNK.SD	get / set the random number generator seed for the DRUNK op

OP *(set)*	*(aliases)*	Description
P.SEED (x)	P.SD	get / set the random number generator seed for the P.RND and PM.RND ops

DRUNK

- DRUNK (x)

Changes by -1, 0, or 1 upon each read, saving its state. Setting DRUNK will give it a new value for the next read, and drunkedness will continue on from there with subsequent reads.

Setting DRUNK.MIN and DRUNK.MAX controls the lower and upper bounds (inclusive) that DRUNK can reach. DRUNK.WRAP controls whether the value can wrap around when it reaches it's bounds.

Control flow

OP *(set)*	*(aliases)*	Description
`IF x: ...`		if `x` is not zero execute command
`ELIF x: ...`		if all previous `IF` / `ELIF` fail, and `x` is not zero, execute command
`ELSE: ...`		if all previous `IF` / `ELIF` fail, excute command
`L x y: ...`		run the command sequentially with `I` values from `x` to `y`
`W x: ...`		run the command while condition x is true
`EVERY x: ...`	`EV`	run the command every `x` times the command is called
`SKIP x: ...`		run the command every time except the `x`th time.
`OTHER: ...`		runs the command when the previous `EVERY`/`SKIP` did not run its command.
`SYNC x`		synchronizes *all* `EVERY` and `SKIP` counters to offset `x`.
`PROB x: ...`		potentially execute command with probability `x` (0-100)
`SCRIPT (x)`	`$`	get current script number, or execute script `x` (1-10), recursion allowed
`SCRIPT.POL x (p)`	`$.POL`	get script `x` trigger polarity, or set polarity `p` (1 rising edge, 2 falling, 3 both)
`$F s`		execute script `s` as a function
`$F1 s p`		execute script `s` as a function with 1 parameter `p`
`$F2 s p1 p2`		execute script `s` as a function with 2 parameters
`$L s l`		execute script `s` line `l`
`$L1 s l p`		execute script `s` line `l` as a function with 1 parameter `p`
`$L2 s l p1 p2`		execute script `s` line `l` as a function with 2 parameters
`$S l`		execute line `l` within the same script as a function
`$S1 l p`		execute line `l` within the same script as a function with 1 parameter `p`

OP *(set)*	*(aliases)*	Description
$S2 l p1 p2		execute line l within the same script as a function with 2 parameters
I1		get the first parameter when executing a script as a function
I2		get the second parameter when executing a script as a function
FR (x)		get/set the return value when a script is called as a function
SCENE (x)		get the current scene number, or load scene x (0-31)
SCENE.G x		load scene x (0-31) without loading grid control states
SCENE.P x		load scene x (0-31) without loading pattern state
KILL		clears stack, clears delays, cancels pulses, cancels slews, disables metronome
BREAK	BRK	halts execution of the current script
INIT		clears all state data
INIT.CV x		clears all parameters on CV associated with output x
INIT.CV.ALL		clears all parameters on all CV's
INIT.DATA		clears all data held in all variables
INIT.P x		clears pattern number x
INIT.P.ALL		clears all patterns
INIT.SCENE		loads a blank scene
INIT.SCRIPT x		clear script number x
INIT.SCRIPT.ALL		clear all scripts
INIT.TIME x		clear time on trigger x
INIT.TR x		clear all parameters on trigger x
INIT.TR.ALL		clear all triggers

IF

- IF x: ...

If x is not zero execute command

Advanced IF / ELIF / ELSE usage

1. Intermediate statements always run

```
SCRIPT 1:
IF 0: 0         => DO NOTHING
TR.P 1          => ALWAYS HAPPENS
ELSE: TR.P 2    => ELSE BRANCH RUNS BECAUSE OF THE PREVIOUS IF
```

2. ELSE without an IF

```
SCRIPT 1:
ELSE: TR.P 1    => NEVER RUNS, AS THERE IS NO PRECEDING IF
```

3. ELIF without an IF

```
SCRIPT 1:
ELIF 1: TR.P 1  => NEVER RUNS, AS THERE IS NO PRECEDING IF
```

4. Independent scripts

```
SCRIPT 1:
IF 1: TR.P 1    => PULSE OUTPUT 1

SCRIPT 2:
ELSE: TR.P 2    => NEVER RUNS REGARDLESS OF WHAT HAPPENS IN SCRIPT 1
                   (SEE EXAMPLE 2)
```

5. Dependent scripts

```
SCRIPT 1:
IF 0: TR.P 1    => DO NOTHING
SCRIPT 2        => WILL PULSE OUTPUT 2

SCRIPT 2:
ELSE: TR.P 2    => WILL NOT PULSE OUTPUT 2 IF CALLED DIRECTLY,
                   BUT WILL IF CALLED FROM SCRIPT 1
```

L

- L x y: ...

Run the command sequentially with I values from x to y.

For example:

```
L 1 4: TR.PULSE I   => PULSE OUTPUTS 1, 2, 3 AND 4
L 4 1: TR.PULSE I   => PULSE OUTPUTS 4, 3, 2 AND 1
```

W

- `W x: ...`

Runs the command while the condition `x` is true or the loop iterations exceed 10000.

For example, to find the first iterated power of 2 greater than 100:

```
A 2
W LT A 100: A * A A
```

A will be 256.

EVERY

- `EVERY x: ...`
- *alias*: `EV`

Runs the command every `x` times the line is executed. This is tracked on a per-line basis, so each script can have 6 different "dividers".

Here is a 1-script clock divider:

```
EVERY 2: TR.P 1
EVERY 4: TR.P 2
EVERY 8: TR.P 3
EVERY 16: TR.P 4
```

The numbers do *not* need to be evenly divisible by each other, so there is no problem with:

```
EVERY 2: TR.P 1
EVERY 3: TR.P 2
```

SKIP

- `SKIP x: ...`

This is the corollary function to `EVERY`, essentially behaving as its exact opposite.

OTHER

- `OTHER: ...`

`OTHER` can be used to do somthing alternately with a preceding `EVERY` or `SKIP` command.

-fl

For e
on-t

```
EN
OTH
```

You

```
SKIP
```

```
SYN
```

Caus
ing t

in an expression. To set the return value, either place an expression at the end of the script without assigning it to anything or assign it to the special function return variable `FR`. If you do both, `FR` will be used, and if you don't do either, zero will be returned.

Let's say you update script 1 to return the square of `X`: `* X X` (which you could also write as `FR * X X`). Then you can use it in an expression like this: `A + A $F 1`.

This op can save space if you have a calculation that is used in multiple places. Other than returning a value, a function script isn't different from a regular script and can perform other actions in addition to calculating something, including calling other scripts. The same limit of 8 maximum nested calls applies here to prevent infinite loops.

If you need to be able to pass parameters into your function, use `$F1` or `$F2` ops.

$F1

- `$F1 s p`

Same as `$F` but you can also pass a single parameter into the function. Inside the function script you can get the parameter using `I1` op.

Let's say you create a script that returns the square of the passed parameter: `FR * I1 I1`. You can then calculate the square of a number by executing `$F1 value`.

See the description of `$F` op for more details on executing scripts as functions.

$F2

- `$F2 s p1 p2`

Same as `$F` but you can also pass two parameters into a function. Inside the function script you can get them using `I1` and `I2` ops.

Let's say you create a script that returns a randomly selected value out of the two provided values: `FR ? TOSS I1 I2`. You can then save space by using `$F2 1 X Y` instead of `? TOSS X Y`. More importantly, you could use it in multiple places, and if you later want to change the calculation to something else, you just need to update the function script.

See the description of `$F` op for more details on executing scripts as functions.

$L

- `$L s l`

This op executes the specified script line. This allows you to use a script as a library of sorts, where each line does something different, so you can use the same script for multiple purposes. It also allows you to use free lines in a script to extend another script.

This op behaves similarly to **$F** op in that it can be used as a function in an expression by setting the return value with **FR**. Let's say the first line in script 1 is this: `FR * X X`. You can then get the square of **X** by executing **$L 1 1**.

If you want to use it as a function and you need to pass some parameters into it, use **$L1** / **$L2** ops.

This op is also useful if you have a loop that doesn't fit on one line - define the line later in the script and then reference it in the loop:

```
#1
L 1 6: A + A $L 1 3
BREAK
SCALE X Y C D I
```

Don't forget to add **BREAK** before the line so that it's not executed when the whole script is executed. If you use this technique, you can also save space by using **$S** op which executes a line within the same script.

$L1

- `$L1 s l p`

Execute the specified script line as a function that takes 1 parameter. See the description of **$L** and **$F1** ops for more details.

$L2

- `$L2 s l p1 p2`

Execute the specified script line as a function that takes 2 parameters. See the description of **$L** and **$F2** ops for more details.

$S

- `$S l`

This is exactly the same as **$L $ line** but saves you space on not having to specify the script number if the line you want to execute is within the same script.

See the description of **$L** for more details.

$S1

- **$S1 l p**

This is exactly the same as **$L1 $ line param** but saves you space on not having to specify the script number if the line you want to execute is within the same script.

See the description of **$L1** for more details.

$S2

- **$S2 l p1 p2**

This is exactly the same as **$L2 $ line param1 param2** but saves you space on not having to specify the script number if the line you want to execute is within the same script.

See the description of **$L2** for more details.

I1

- **I1**

This op returns the first parameter when a script is called as a function using **$F1** / **$F2** / **$L1** / **$L2** / **$S1** / **$S2** ops. If the script is called using other ops, this op will return zero.

I2

- **I2**

This op returns the second parameter when a script is called as a function using **$F2** / **$L2** / **$S2** ops. If the script is called using other ops, this op will return zero.

FR

- **FR (x)**

Use this op to get or set the return value in a script that is called as a function.

SCENE

- `SCENE (x)`

Load scene `x` (0-31).

Does *not* execute the **I** script. Will *not* execute from the **I** script on scene load. Will execute on subsequent calls to the **I** script.

WARNING: You will lose any unsaved changes to your scene.

SCENE.G

- `SCENE.G x`

Load scene `x` (0-31) without loading grid button and fader states.

WARNING: You will lose any unsaved changes to your scene.

SCENE.P

- `SCENE.P x`

Load scene `x` (0-31) without loading pattern data.

WARNING: You will lose any unsaved changes to your scene.

INIT

- `INIT`

WARNING: You will lose all settings when you initialize using `INIT` - there is NO undo!

INIT.DATA

- `INIT.DATA`

Clears the following variables and resets them to default values: A, B, C, D, CV slew, Drunk min/max, M, O, Q, R, T, TR. Does not affect the CV input (IN) or the Parameter knob (PARAM) values.

Maths

Logical operators such as `EQ`, `OR` and `LT` return `1` for true, and `0` for false.

OP *(set)*	*(aliases)*	Description
`ADD x y`	`+`	add `x` and `y` together
`SUB x y`	`-`	subtract `y` from `x`
`MUL x y`	`*`	multiply `x` and `y` together
`DIV x y`	`/`	divide `x` by `y`
`MOD x y`	`%`	find the remainder after division of `x` by `y`
`? x y z`		if condition `x` is true return `y`, otherwise return `z`
`MIN x y`		return the minimum of `x` and `y`
`MAX x y`		return the maximum of `x` and `y`
`LIM x y z`		limit the value `x` to the range `y` to `z` inclusive
`WRAP x y z`	`WRP`	limit the value `x` to the range `y` to `z` inclusive, but with wrapping
`QT x y`		round `x` to the closest multiple of `y` (quantise)
`AVG x y`		the average of `x` and `y`
`EQ x y`	`==`	does `x` equal `y`
`NE x y`	`!=` , `XOR`	`x` is not equal to `y`
`LT x y`	`<`	`x` is less than `y`
`GT x y`	`>`	`x` is greater than `y`
`LTE x y`	`<=`	`x` is less than or equal to `y`
`GTE x y`	`>=`	`x` is greater than or equal to `y`
`INR l x h`	`><`	`x` is greater than `l` and less than `h` (within range)
`OUTR l x h`	`<>`	`x` is less than `l` or greater than `h` (out of range)
`INRI l x h`	`>=<`	`x` is greater than or equal to `l` and less than or equal to `h` (within range, inclusive)
`OUTRI l x h`	`<=>`	`x` is less than or equal to `l` or greater than or equal to `h` (out of range, inclusive)
`EZ x`	`!`	`x` is 0, equivalent to logical NOT
`NZ x`		`x` is not 0
`LSH x y`	`<<`	left shift `x` by `y` bits, in effect multiply `x` by `2` to the power of `y`

OP *(set)*	*(aliases)*	Description
`RSH x y`	`>>`	right shift `x` by `y` bits, in effect divide `x` by `2` to the power of `y`
`LROT x y`	`<<<`	circular left shift `x` by `y` bits, wrapping around when bits fall off the end
`RROT x y`	`>>>`	circular right shift `x` by `y` bits, wrapping around when bits fall off the end
`| x y`		bitwise or `x`
`& x y`		bitwise and `x & y`
`^ x y`		bitwise xor `x ^ y`
`~ x`		bitwise not, i.e.: inversion of `x`
`BSET x y`		set bit `y` in value `x`
`BGET x y`		get bit `y` in value `x`
`BCLR x y`		clear bit `y` in value `x`
`BTOG x y`		toggle bit `y` in value `x`
`BREV x`		reverse bit order in value `x`
`ABS x`		absolute value of `x`
`AND x y`	`&&`	logical AND of `x` and `y`
`AND3 x y z`	`&&&`	logical AND of `x`, `y` and `z`
`AND4 x y z a`	`&&&&`	logical AND of `x`, `y`, `z` and `a`
`OR x y`	`||`	logical OR of `x` and `y`
`OR3 x y z`	`|||`	logical OR of `x`, `y` and `z`
`OR4 x y z a`	`||||`	logical OR of `x`, `y`, `z` and `a`
`SCALE a b x y i`	`SCL`	scale `i` from range `a` to `b` to range `x` to `y`, i.e. `i * (y - x) / (b - a)`
`SCALE a b i`	`SCL0`	scale `i` from range 0 to `a` to range 0 to `b`
`EXP x`		exponentiation table lookup. 0-16383 range (`V` 0-10)
`SGN x`		sign function: 1 for positive, -1 for negative, 0 for 0

MUL

- `MUL x y`
- *alias*: `*`

returns `x` times `y`, bounded to integer limits

QT

- `QT x y`

Round `x` to the closest multiple of `y`. *See also:* `QT.S`, `QT.CS`, `QT.B`, `QT.BX` *in the Pitch section*.

AND

- `AND x y`
- *alias*: `&&`

Logical AND of `x` and `y`. Returns `1` if both `x` and `y` are greater than 0, otherwise it returns 0.

AND3

- `AND3 x y z`
- *alias*: `&&&`

Logical AND of `x`, `y` and `z`. Returns `1` if both `x`, `y` and `z` are greater than 0, otherwise it returns 0.

AND4

- `AND4 x y z a`
- *alias*: `&&&&`

Logical AND of `x`, `y`, `z` and `a`. Returns `1` if both `x`, `y`, `z` and `a` are greater than 0, otherwise it returns 0.

OR

- `OR x y`
- *alias*: `||`

Logical OR of `x` and `y`. Returns `1` if either `x` or `y` are greater than 0, otherwise it returns 0.

OR3

- `OR3 x y z`
- *alias*: `|||`

Logical OR of `x`, `y` and `z`. Returns `1` if either `x`, `y` or `z` are greater than 0, otherwise it returns 0.

OR4

- `OR4 x y z a`
- *alias*: `||||`

Logical OR of `x`, `y`, `z` and `a`. Returns `1` if either `x`, `y`, `z` or `a` are greater than 0, otherwise it returns 0.

Delay

The **DEL** delay op allow commands to be sheduled for execution after a defined interval by placing them into a buffer which can hold up to 64 commands. Commands can be delayed by up to 16 seconds.

In LIVE mode, the second icon (an upside-down U) will be lit up when there is a command in the **DEL** buffer.

OP *(set)*	*(aliases)*	Description
DEL x: ...		Delay command by **x** ms
DEL.CLR		Clear the delay buffer
DEL.X x t: ...		Delay **x** commands at **t** ms intervals
DEL.R x t: ...		Trigger the command following the colon once immediately, and delay **x - 1** commands at **t** ms intervals
DEL.G x t n d: ...		Trigger the command once immediately and **x - 1** times at ms intervals of **t * (n/d)^n** where n ranges from 0 to **x - 1**.
DEL.B t b: ...		Trigger the command up to 16 times at intervals of **t** ms, with active intervals set in 16-bit bitmask **b**, LSB = immediate.

DEL

- **DEL x: ...**

Delay the command following the colon by **x** ms by placing it into a buffer. The buffer can hold up to 16 commands. If the buffer is full, additional commands will be discarded.

DEL.CLR

- **DEL.CLR**

Clear the delay buffer, cancelling the pending commands.

DEL.X

- **DEL.X x t: ...**

Delay the command following the colon **x** times at intervals of **t** ms by placing it into a buffer. The buffer can hold up to 16 commands. If the buffer is full, additional commands will be discarded.

DEL.R

- **DEL.R x t: ...**

Delay the command following the colon once immediately, and **x - 1** times at intervals of **t** ms by placing it into a buffer. The buffer can hold up to 16 commands. If the buffer is full, additional commands will be discarded.

DEL.G

- **DEL.G x t n d: ...**

Trigger the command once immediately and **x - 1** times at ms intervals of **t** * (n/d)^n where n ranges from 0 to x-1 by placing it into a buffer. The buffer can hold up to 16 commands. If the buffer is full, additional commands will be discarded.

Stack

These operators manage a last in, first out, stack of commands, allowing them to be memorised for later execution at an unspecified time. The stack can hold up to 8 commands. Commands added to a full stack will be discarded.

OP *(set)*	*(aliases)*	Description
S: ...		Place a command onto the stack
S.CLR		Clear all entries in the stack
S.ALL		Execute all entries in the stack
S.POP		Execute the most recent entry
S.L		Get the length of the stack

S

- S: ...

Add the command following the colon to the top of the stack. If the stack is full, the command will be discarded.

S.CLR

- S.CLR

Clear the stack, cancelling all of the commands.

S.ALL

- S.ALL

Execute all entries in the stack (last in, first out), clearing the stack in the process.

S.POP

- S.POP

Pop the most recent command off the stack and execute it.

S.L

- S.L

Get the number of entries in the stack.

Patterns

Patterns facilitate musical data manipulation– lists of numbers that can be used as sequences, chord sets, rhythms, or whatever you choose. Pattern memory consists four banks of 64 steps. Functions are provided for a variety of pattern creation, transformation, and playback.

New in teletype 2.0, a second version of all Pattern ops have been added. The original **P** ops (**P**, **P.L**, **P.NEXT**, etc.) act upon the 'working pattern' as defined by **P.N**. By default the working pattern is assigned to pattern 0 (**P.N 0**), in order to execute a command on pattern 1 using **P** ops you would need to first reassign the working pattern to pattern 1 (**P.N 1**).

The new set of ops, **PN** (**PN**, **PN.L**, **PN.NEXT**, etc.), include a variable to designate the pattern number they act upon, and don't effect the pattern assignment of the 'working pattern' (ex: **PN.NEXT 2** would increment pattern 2 one index and return the value at the new index). For simplicity throughout this introduction we will only refer to the **P** ops, but keep in mind that they now each have a **PN** counterpart (all of which are detailed below)

Both patterns and their arrays of numbers are indexed from 0. This makes the first pattern number 0, and the first value of a pattern is index 0. The pattern index (**P.I**) functions like a playhead which can be moved throughout the pattern and/or read using ops: **P**, **P.I**, **P.HERE**, **P.NEXT**, and **P.PREV**. You can contain pattern movements to ranges of a pattern and define wrapping behavior using ops: **P.START**, **P.END**, **P.L**, and **P.WRAP**.

Values can be edited, added, and retrieved from the command line using ops: **P**, **P.INS**, **P.RM**, **P.PUSH**, **P.HERE**, **P.NEXT**, and **P.PREV**. Some of these ops will additionally impact the pattern length upon their execution: **P.INS**, **P.RM**, **P.PUSH**, and **P.POP**.

To see your current pattern data use the tab key to cycle through live mode, edit mode, and pattern mode. In pattern mode each of the 4 patterns is represented as a column. You can use the arrow keys to navigate throughout the 4 patterns and their 64 values. For reference a key of numbers runs the down the lefthand side of the screen in pattern mode displaying 0-63.

From a blank set of patterns you can enter data by typing into the first cell in a column. Once you hit enter you will move to the cell below and the pattern length will become one step long. You can continue this process to write out a pattern of desired length. The step you are editing is always the brightest. As you add steps to a pattern by editing the value and hitting enter they become brighter than the unused cells. This provides a visual indication of the pattern length.

The start and end points of a pattern are represented by the dotted line next to the column, and the highlighted dot in this line indicates the current pattern

index for each of the patterns. See the key bindings for an extensive list of editing shortcuts available within pattern mode.

OP *(set)*	*(aliases)*	Description
P.M (x)		get/set the pattern number for the working pattern, default 0
P x (y)		get/set the value of the working pattern at index x
PM x y (z)		get/set the value of pattern x at index y
P.L (x)		get/set pattern length of the working pattern, non-destructive to data
PM.L x (y)		get/set pattern length of pattern x. non-destructive to data
P.WRAP (x)		when the working pattern reaches its bounds does it wrap (0/1), default 1 (enabled)
PM.WRAP x (y)		when pattern x reaches its bounds does it wrap (0/1), default 1 (enabled)
P.START (x)		get/set the start location of the working pattern, default 0
PM.START x (y)		get/set the start location of pattern x, default 0
P.END (x)		get/set the end location of the working pattern, default 63
PM.END x (y)		get/set the end location of the pattern x, default 63
P.I (x)		get/set index position for the working pattern.
PM.I x (y)		get/set index position for pattern x
P.HERE (x)		get/set value at current index of working pattern
PM.HERE x (y)		get/set value at current index of pattern x
P.NEXT (x)		increment index of working pattern then get/set value
PM.NEXT x (y)		increment index of pattern x then get/set value
P.PREV (x)		decrement index of working pattern then get/set value
PM.PREV x (y)		decrement index of pattern x then get/set value

OP *(set)*	*(aliases)*	Description
P.INS x y		insert value y at index x of working pattern, shift later values down, destructive to loop length
PN.INS x y z		insert value z at index y of pattern x, shift later values down, destructive to loop length
P.RM x		delete index x of working pattern, shift later values up, destructive to loop length
PN.RM x y		delete index y of pattern x, shift later values up, destructive to loop length
P.PUSH x		insert value x to the end of the working pattern (like a stack), destructive to loop length
PN.PUSH x y		insert value y to the end of pattern x (like a stack), destructive to loop length
P.POP		return and remove the value from the end of the working pattern (like a stack), destructive to loop length
PN.POP x		return and remove the value from the end of pattern x (like a stack), destructive to loop length
P.MIN		find the first minimum value in the pattern between the START and END for the working pattern and return its index
PN.MIN x		find the first minimum value in the pattern between the START and END for pattern x and return its index
P.MAX		find the first maximum value in the pattern between the START and END for the working pattern and return its index
PN.MAX x		find the first maximum value in the pattern between the START and END for pattern x and return its index
P.SHUF		shuffle the values in active pattern (between its START and END)
PN.SHUF x		shuffle the values in pattern x (between its START and END)
P.ROT n		rotate the values in the active pattern n steps (between its START and END, negative rotates backward)

OP *(set)*	*(aliases)*	Description
PN.ROT x n		rotate the values in pattern x (between its START and END, negative rotates backward)
P.REV		reverse the values in the active pattern (between its START and END)
PN.REV x		reverse the values in pattern x
P.RND		return a value randomly selected between the start and the end position
PN.RND x		return a value randomly selected between the start and the end position of pattern x
P.+ x y		increase the value of the working pattern at index x by y
PN.+ x y z		increase the value of pattern x at index y by z
P.- x y		decrease the value of the working pattern at index x by y
PN.- x y z		decrease the value of pattern x at index y by z
P.+W x y a b		increase the value of the working pattern at index x by y and wrap it to a..b range
PN.+W x y z a b		increase the value of pattern x at index y by z and wrap it to a..b range
P.-W x y a b		decrease the value of the working pattern at index x by y and wrap it to a..b range
PN.-W x y z a b		decrease the value of pattern x at index y by z and wrap it to a..b range
P.MAP: ...		apply the 'function' to each value in the active pattern, I takes each pattern value
PN.MAP x: ...		apply the 'function' to each value in pattern x, I takes each pattern value

P.N

- P.N (x)

get/set the pattern number for the working pattern, default 0. All P ops refer to this pattern.

P

- P x (y)

get/set the value of the working pattern at index x. All positive values (0-63) can be set or returned while index values greater than 63 clip to 63. Negative x values are indexed backwards from the end of the pattern length of the working pattern.

Example:

with a pattern length of 6 for the working pattern:

P 10 retrieves the working pattern value at index 6

P.I -2 retrieves the working pattern value at index 4

This applies to PN as well, except the pattern number is the first variable and a second variable specifies the index.

P.WRAP

- P.WRAP (x)

when the working pattern reaches its bounds does it wrap (0/1). With PN.WRAP enabled (1), when an index reaches its upper or lower bound using P.NEXT or P.PREV it will wrap to the other end of the pattern and you can continue advancing. The bounds of P.WRAP are defined through P.L, P.START, and P.END.

If wrap is enabled (P.WRAP 1) a pattern will begin at its start location and advance to the lesser index of either its end location or the end of its pattern length

Examples:

With wrap enabled, a pattern length of 6, a start location of 2 , and an end location of 8.

```
P.WRAP 1; P.L 6; P.START 2; P.END 8
```

The pattern will wrap between the indexes 2 and 5.

With wrap enabled, a pattern length of 10, a start location of 3, and an end location of 6.

```
P.WRAP 1; P.L 10; P.START 3; P.END 6
```

The pattern will wrap between the indexes 3 and 6.

If wrap is disabled (P.WRAP 0) a pattern will run between its start and end locations and halt at either bound.

This applies to **PN.WRAP** as well, except the pattern number is the first variable and a second variable specifies the wrap behavior (0/1).

P.I

- **P.I (x)**

get/set index position for the working pattern. all values greater than pattern length return the first step beyond the pattern length. negative values are indexed backwards from the end of the pattern length.

Example:

With a pattern length of 6 (**P.L 6**), yielding an index range of 0-5:

P.I 3

moves the index of the working pattern to 3

P.I 10

moves the index of the working pattern to 6

P.I -2

moves the index of the working pattern to 4

This applies to **PN.I**, except the pattern number is the first variable and a second variable specifics the index.

P.MAP

- **P.MAP: ...**

Replace each cell in the active pattern (between the START and END of the pattern) by assigning the variable I to the current value of the cell, evaluating the command after the mod, and assigning that pattern cell with the result. The 'map' higher-order function from functional programming, with the command giving the function of **I** to map over the pattern.

For example:

```
P.MAP: * 2 I  => double each cell in the active pattern
```

Queue

These operators manage a first in, first out, queue of values. The length of the queue can be dynamically changed up to a maximum size of 64 elements. A fixed length can be set with the `Q.M` operator, or the queue can grow and shrink automatically by setting `Q.GRW 1`. The queue contents will be preserved when the length is shortened.

Queues also offer operators that do math on the entire queue (the `Q.AVG` operator is particularly useful for smoothing input values) or copy the queue to and from a tracker pattern.

Most operators manipulates the elements up to (and including) length. Exceptions are `Q.I i x` and `Q.P2`.

Examples, only first 8 elements shown for clarity: By default all elements of the queue have a value of `0` and the length is set to 1.

```
Q.M "LENGTH" -> 1
ELEMENT NR: 1  |  2 3 4 5 6 7 8
VALUE       0  |  0 0 0 0 0 0 0
```

Using the Q OP will add values to the beginning of the queue and push the other elements to the right.

```
Q 1
1  |  0 0 0 0 0 0 0

Q 2 // ADD 2 TO QUEUE
Q 3 // ADD 3 TO QUEUE

3  |  2 1 0 0 0 0 0
```

Using the `Q` getter OP will return the last element in the queue, but not modify content or the state of the queue.

```
Q // WILL RETURN 3

3  |  2 1 0 0 0 0 0
```

Using the `Q.M` OP will either return the position of the end marker (1-indexed) or move it:

```
Q.M 2 // set the length to 2 by moving the end marker:

3 2  |  1 0 0 0 0 0

Q // get the value at the end, now `2`
```

By default grow is disabled, but it can be turned on with `Q.GRW 1`. With grow enabled, the queue will automatically expand when new elements are added with `Q x` and likewise shrink when reading with `Q`.

```
Q.GRW // ENABLE GROW
3 2 | 1 0 0 0 0 0
Q 4 // ADD TO TO QUEUE
4 3 2 | 1 0 0 0 0
Q // READ ELEMENT FROM QUEUE, WILL RETURN 2
4 3 | 2 1 0 0 0 0
```

OP *(set)*	*(aliases)*	Description
`Q (x)`		Modify the queue entries
`Q.M (x)`		The queue length
`Q.AVG (x)`		Return the average of the queue
`Q.CLR (x)`		Clear queue
`Q.GRW (x)`		Get/set grow state
`Q.SUM (x)`		Get sum of elements
`Q.MIN (x)`		Get/set minimum value
`Q.MAX (x)`		Get/set maximum value
`Q.RND (x)`		Get random element/randomize elements
`Q.SRT (SRT)`		Sort all or part of queue
`Q.REV`		Reverse queue
`Q.SH (x)`		Shift elements in queue
`Q.ADD x (i)`		Perform addition on elements in queue
`Q.SUB x (i)`		Perform subtraction on elements in queue
`Q.MUL x (i)`		Perform multiplication on elements in queue
`Q.DIV x (i)`		Perform division on elements in queue
`Q.MOD x (i)`		Perform module (%) on elements in queue
`Q.I i (x)`		Get/set value of elements at index
`Q.2P (i)`		Copy queue to current pattern/copy queue to pattern at index `i`
`Q.P2 (i)`		Copy current pattern to queue/copy pattern at index `i` to queue

Q

- `Q (x)`

Gets the output value from the queue, or places `x` into the queue.

Q.N

- `Q.N (x)`

Gets/sets the length of the queue. The length is 1-indexed.

Q.AVG

- `Q.AVG (x)`

Getting the value the average of the values in the queue. Setting `x` sets the value of each entry in the queue to `x`.

Q.CLR

- `Q.CLR (x)`

Clear queue, set all values to 0, length to 1. If parameter `x` is provided, set first elements to `x`.

Q.GRW

- `Q.GRW (x)`

If grow is set (value of 1) the queue will automatically grow and shrink when using `Q` (popping and pushing).

Q.SUM

- `Q.SUM (x)`

Get sum of all elements in queue.

Q.MIN

- `Q.MIN (x)`

Get the minimum value of elements in queue. If `x` is provided, set elements with a value less than `x` to `x`.

Q.MAX

- `Q.MAX (x)`

Get the maximum value of elements in queue. If `x` is provided, set elements with a value greater than `x` to `x`.

Q.RND

- `Q.RND (x)`

Get a random element in queue.

If `x` > 0, set all elements to a random value 0-`x`. If `x` < 0, swap two elements `-x` number of times. IF `x` == 0, do nothing.

Q.SRT

- `Q.SRT (SRT)`

Sort elements in queue. With no arguments, entire queue is sorted in accending order.

If `x` > 0, sort elements from index `i` to the end of queue. If `x` < 0, sort elements from beginning of queue to index `-i`. IF `x` == 0, sort all elements.

Index `i` is 0-indexed.

Q.REV

- `Q.REV`

Reverse order of elements in queue.

Q.SH

- `Q.SH (x)`

Shift elements `x` locations to right. Negative values of `x` shifts to the left. No value provided is equal to `x` = 1. Shifting is wrapped.

Q.ADD

- `Q.ADD x (i)`

Add **x** to all elements in queue. If index **i** is provided, only perform addition on element at index **i**.

Index **i** is 0-indexed.

Q.SUB

- `Q.SUB x (i)`

Subtract **x** from all elements in queue. If index **i** is provided, only perform subtraction on element at index **i**.

Index **i** is 0-indexed.

Q.MUL

- `Q.MUL x (i)`

Multiply all elements in queue with **x**. If index **i** is provided, only perform multiplication on element at index **i**.

Index **i** is 0-indexed.

Q.DIV

- `Q.DIV x (i)`

Divide all elements in queue by **x**. If index **i** is provided, only perform division on element at index **i**.

Index **i** is 0-indexed.

Q.MOD

- `Q.MOD x (i)`

Perform modulo of **x** (value = value % **x**) on all elements in queue. If index **i** is provided, only perform modulo operation on element at index **i**.

Index **i** is 0-indexed.

Q.I

- `Q.I i (x)`

Get value of element at index **i** or set value of element **i** to value **x**. Indexing works on entire lenght of queue, and is not limited to elements below queue end point.

Index **i** is 0-indexed.

Q.2P

- `Q.2P (i)`

Copy entire queue to current pattern or (if **i** provided) pattern at index **i**.

Index **i** is 0-indexed.

Q.P2

- `Q.P2 (i)`

Copy current pattern to queue or (if **i** provided) copy pattern at index **i** to queue.

Index **i** is 0-indexed.

Turtle

A 2-dimensional, movable index into the pattern values as displayed on the TRACKER screen.

OP *(set)*	*(aliases)*	Description
`@ (x)`		get or set the current pattern value under the turtle
`@X (x)`		get the turtle X coordinate, or set it to `x`
`@Y (x)`		get the turtle Y coordinate, or set it to `x`
`@MOVE x y`		move the turtle `x` cells in the X axis and `y` cells in the Y axis
`@F x1 y1 x2 y2`		set the turtle's fence to corners `x1`, `y1` and `x2`, `y2`
`@FX1 (x)`		get the left fence line or set it to `x`
`@FX2 (x)`		get the right fence line or set it to `x`
`@FY1 (x)`		get the top fence line or set it to `x`
`@FY2 (x)`		get the bottom fence line or set it to `x`
`@SPEED (x)`		get the speed of the turtle's @STEP in cells per step or set it to `x`
`@DIR (x)`		get the direction of the turtle's @STEP in degrees or set it to `x`
`@STEP`		move `@SPEED`/100 cells forward in `@DIR`, triggering `@SCRIPT` on cell change
`@BUMP (1)`		get whether the turtle fence mode is `BUMP`, or set it to `BUMP` with `1`
`@WRAP (1)`		get whether the turtle fence mode is `WRAP`, or set it to `WRAP` with `1`
`@BOUNCE (1)`		get whether the turtle fence mode is `BOUNCE`, or set it to `BOUNCE` with `1`
`@SCRIPT (x)`		get which script runs when the turtle changes cells, or set it to `x`
`@SHOW (1)`		get whether the turtle is displayed on the TRACKER screen, or turn it on or off

Grid

Grid operators allow creating scenes that can interact with grid connected to teletype (important: grid must be powered externally, do not connect it directly to teletype!). You can light up individual LEDs, draw shapes and create controls (such as buttons and faders) that can be used to trigger and control scripts. You can take advantage of grid operators even without an actual grid by using the built in Grid Visualizer.

For more information on grid integration see Advanced section and Grid Studies[14].

As there are many operators let's review some naming conventions that apply to the majority of them. All grid ops start with `G.`. For control related ops this is followed by 3 letters specifying the control: `G.BTN` for buttons, `G.FDR` for faders. To define a control you use the main ops `G.BTN` and `G.FDR`. To define multiple controls replace the last letter with `X`: `G.BTX`, `G.FDX`.

All ops that initialize controls use the same list of parameters: id, coordinates, width, height, type, level, script. When creating multiple controls there are two extra parameters: the number of columns and the number of rows. Controls are created in the current group (set with `G.GRP`). To specify a different group use the group versions of the 4 above ops - `G.GBT`, `G.GFD`, `G.GBX`, `G.GFX` and specify the desired group as the first parameter.

All controls share some common properties, referenced by adding a `.` and:

- `EN`: `G.BTN.EN`, `G.FDR.EN` - enables or disables a control
- `V`: `G.BTN.V`, `G.FDR.V` - value, 1/0 for buttons, range value for faders
- `L`: `G.BTN.L`, `G.FDR.L` - level (brightness level for buttons and coarse faders, max value level for fine faders)
- `X`: `G.BTN.X`, `G.FDR.X` - the X coordinate
- `Y`: `G.BTN.Y`, `G.FDR.Y` - the Y coordinate

To get/set properties for individual controls you normally specify the control id as the first parameter: `G.FDR.V 5` will return the value of fader 5. Quite often the actual id is not important, you just want to work with the latest control pressed. As these are likely the ops to be used most often they are offered as shortcuts without a .: `G.BTNV` returns the value of the last button pressed, `G.FDRL 4` will set the level of the last fader pressed etc etc.

OP *(set)*	*(aliases)*	Description
`G.RST`		full grid reset
`G.CLR`		clear all LEDs

[14] https://github.com/scanner-darkly/teletype/wiki/GRID-INTEGRATION

OP *(set)*	*(aliases)*	Description
`G.DIM level`		set dim level
`G.ROTATE x`		set grid rotation
`G.KEY x y action`		emulate grid press
`G.GRP (id)`		get/set current group
`G.GRP.EN id (x)`		enable/disable group or check if enabled
`G.GRP.RST id`		reset all group controls
`G.GRP.SW id`		switch groups
`G.GRP.SC id (script)`		get/set group script
`G.GRPI`		get last group
`G.LED x y (level)`		get/set LED
`G.LED.C x y`		clear LED
`G.REC x y w h fill border`		draw rectangle
`G.RCT x1 y1 x2 y2 fill border`		draw rectangle
`G.BTN id x y w h type level script`		initialize button
`G.GBT group id x y w h type level script`		initialize button in group
`G.BTX id x y w h type level script columns rows`		initialize multiple buttons
`G.GBX group id x y w h type level script columns rows`		initialize multiple buttons in group
`G.BTN.EN id (x)`		enable/disable button or check if enabled
`G.BTN.X id (x)`		get/set button x coordinate
`G.BTN.Y id (y)`		get/set button y coordinate
`G.BTN.V id (value)`		get/set button value
`G.BTN.L id (level)`		get/set button level
`G.BTNI`		id of last pressed button
`G.BTNX (x)`		get/set x of last pressed button
`G.BTNY (y)`		get/set y of last pressed button
`G.BTNV (value)`		get/set value of last pressed button
`G.BTNL (level)`		get/set level of last pressed button
`G.BTN.SW id`		switch button
`G.BTN.PR id action`		emulate button press/release

OP *(set)*	*(aliases)*	Description
G.GBTN.V group value		set value for group buttons
G.GBTN.L group odd_level even_level		set level for group buttons
G.GBTN.C group		get count of currently pressed
G.GBTN.I group index		get id of pressed button
G.GBTN.W group		get button block width
G.GBTN.H group		get button block height
G.GBTN.X1 group		get leftmost pressed x
G.GBTN.X2 group		get rightmost pressed x
G.GBTN.Y1 group		get highest pressed y
G.GBTN.Y2 group		get lowest pressed y
G.FDR id x y w h type level script		initialize fader
G.GFD grp id x y w h type level script		initialize fader in group
G.FDX id x y w h type level script columns rows		initialize multiple faders
G.GFX group id x y w h type level script columns rows		initialize multiple faders in group
G.FDR.EN id (x)		enable/disable fader or check if enabled
G.FDR.X id (x)		get/set fader x coordinate
G.FDR.Y id (y)		get/set fader y coordinate
G.FDR.N id (value)		get/set fader value
G.FDR.V id (value)		get/set scaled fader value
G.FDR.L id (level)		get/set fader level
G.FDRI		id of last pressed fader
G.FDRX (x)		get/set x of last pressed fader
G.FDRY (y)		get/set y of last pressed fader
G.FDRN (value)		get/set value of last pressed fader
G.FDRV (value)		get/set scaled value of last pressed fader
G.FDRL (level)		get/set level of last pressed fader
G.FDR.PR id value		emulate fader press
G.GFDR.N group value		set value for group faders
G.GFDR.V group value		set scaled value for group faders

OP *(set)*	*(aliases)*	Description
`G.GFDR.L group odd_level even_level`		set level for group faders
`G.GFDR.RM group min max`		set range for group faders

G.RST

- `G.RST`

Full grid reset - hide all controls and reset their properties to the default values, clear all LEDs, reset the dim level and the grid rotation.

G.CLR

- `G.CLR`

Clear all LEDs set with `G.LED`, `G.REC` or `G.RCT`.

G.DIM

- `G.DIM level`

Set the dim level (0..14, higher values dim more). To remove set to 0.

G.ROTATE

- `G.ROTATE x`

Set the grid rotation (0 - no rotation, 1 - rotate by 180 degrees).

G.KEY

- `G.KEY x y action`

Emulate a grid key press at the specified coordinates (0-based). Set `action` to 1 to emulate a press, 0 to emulate a release. You can also emulate a button press with `G.BTN.PR` and a fader press with `G.FDR.PR`.

G.GRP

- `G.GRP (id)`

Get or set the current group. Grid controls created without specifying a group will be assigned to the current group. This op doesn't enable/disable groups - use G.GRP.EN for that. The default current group is 0. 64 groups are available.

G.GRP.EN

- G.GRP.EN id (x)

Enable or disable the specified group or check if it's currently enabled. 1 means enabled, 0 means disabled. Enabling or disabling a group enables / disables all controls assigned to that group (disabled controls are not shown and receive no input). This allows groups to be used as pages - initialize controls in different groups, and then simply enable one group at a time.

G.GRP.RST

- G.GRP.RST id

Reset all controls associated with the specified group. This will disable the controls and reset their properties to the default values. This will also reset the fader scale range to 0..16383.

G.GRP.SW

- G.GRP.SW id

Switch groups. Enables the specified group, disables all others.

G.GRP.SC

- G.GRP.SC id (script)

Assign a script to the specified group, or get the currently assigned script. The script gets executed whenever a control associated with the group receives input. It is possible to have different scripts assigned to a control and the group it belongs to. Use 9 for Metro and 10 for Init. To unassign, set it to 0.

G.GRPI

- G.GRPI

Get the id of the last group that received input. This is useful when sharing a script between multiple groups.

G.LED

- `G.LED x y (level)`

Set the LED level or get the current level at the specified coordinates. Possible level range is 0..15 (on non varibright grids anything below 8 is 'off', 8 or above is 'on').

Grid controls get rendered first, and LEDs are rendered last. This means you can use LEDs to accentuate certain areas of the UI. This also means that any LEDs that are set will block whatever is underneath them, even with the level of 0. In order to completely clear an LED set its level to -3. There are two other special values for brightness: -1 will dim, and -2 will brighten what's underneath. They can be useful to highlight the current sequence step, for instance.

G.LED.C

- `G.LED.C x y`

Clear the LED at the specified coordinates. This is the same as setting the brightness level to -3. To clear all LEDs use `G.CLR`.

G.REC

- `G.REC x y w h fill border`

Draw a rectangle with the specified width and height. `x` and `y` are the coordinates of the top left corner. Coordinates are 0-based, with the 0,0 point located at the top left corner of the grid. You can draw rectangles that are partially outside of the visible area, and they will be properly cropped.

`fill` and `border` specify the brightness levels for the inner area and the one-LED-wide border respectively, 0..15 range. You can use the three special brightness levels: -1 to dim, -2 to brighten and -3 for transparency (you could draw just a frame by setting `fill` to -3, for instance).

To draw lines, set the width or the height to 1. In this case only `border` brightness level is used.

G.RCT

- `G.RCT x1 y1 x2 y2 fill border`

Same as `G.REC` but instead of specifying the width and height you specify the coordinates of the top left corner and the bottom right corner.

G.BTN

- G.BTN id x y w h type level script

Initializes and enables a button with the specified id. 256 buttons are available (ids are 0-based so the possible id range is 0..255. The button will be assigned to the current group (set with G.GRP). Buttons can be reinitialized at any point.

x and y specify the coordinates of the top left corner, and w and h specify width and height respectively. type determines whether the button is latching (1) or momentary (0). level sets the "off" brightness level, possible rand is -3..15 (the brightness level for pressed buttons is fixed at 13).

script specifies the script to be executed when the button is pressed or released (the latter only for momentary buttons). Use 9 for Metro and 10 for Init. Use 0 if you don't need a script assigned.

G.GBT

- G.GBT group id x y w h type level script

Initialize and enable a button. Same as G.BTN but you can also choose which group to assign the button too.

G.BTX

- G.BTX id x y w h type level script columns rows

Initialize and enable a block of buttons in the current group with the specified number of columns and rows . Ids are incremented sequentially by columns and then by rows.

G.GBX

- G.GBX group id x y w h type level script columns rows

Initialize and enable a block of buttons. Same as G.BTX but you can also choose which group to assign the buttons too.

G.BTN.EN

- G.BTN.EN id (x)

Enable (set x to 1) or disable (set x to 0) a button with the specified id, or check if it's currently enabled. Disabling a button hides it and stops it from receiving input but keeps all the other properties (size/location etc) intact.

G.BTN.X

- `G.BTN.X id (x)`

Get or set `x` coordinate for the specified button's top left corner.

G.BTN.Y

- `G.BTN.Y id (y)`

Get or set `y` coordinate for the specified button's top left corner.

G.BTN.V

- `G.BTN.V id (value)`

Get or set the specified button's value. For buttons the value of 1 means the button is pressed and 0 means it's not. If there is a script assigned to the button it will not be triggered if you change the value - use `G.BTN.PR` for that.

Button values don't change when a button is disabled. Button values are stored with the scene (both to flash and to USB sticks).

G.BTN.L

- `G.BTN.L id (level)`

Get or set the specified button's brightness level (-3..15). Please note you can only set the level for unpressed buttons, the level for pressed buttons is fixed at 13.

G.BTNI

- `G.BTNI`

Get the id of the last pressed button. This is useful when multiple buttons are assigned to the same script.

G.BTNX

- `G.BTNX (x)`

Get or set `x` coordinate of the last pressed button's top left corner. This is the same as `G.BTN.X G.BTNI`.

G.BTNY

- `G.BTNY (y)`

Get or set `y` coordinate of the last pressed button's top left corner. This is the same as `G.BTN.Y G.BTNI`.

G.BTNV

- `G.BTNV (value)`

Get or set the value of the last pressed button. This is the same as `G.BTN.V G.BTNI`. This op is especially useful with momentary buttons when you want to react to presses or releases only - just put `IF EZ G.BTNV: BREAK` in the beginning of the assigned script (this will ignore releases, to ignore presses replace `NZ` with `EZ`).

G.BTNL

- `G.BTNL (level)`

Get or set the brightness level of the last pressed button. This is the same as `G.BTN.L G.BTNI`.

G.BTN.SW

- `G.BTN.SW id`

Set the value of the specified button to 1 (pressed), set it to 0 (not pressed) for all other buttons within the same group (useful for creating radio buttons).

G.BTN.PR

- `G.BTN.PR id action`

Emulate pressing/releasing the specified button. Set `action` to `1` for press, `0` for release (`action` is ignored for latching buttons).

G.GBTN.V

- `G.GBTN.V group value`

Set the value for all buttons in the specified group.

G.GBTN.L

- `G.GBTN.L group odd_level even_level`

Set the brightness level (0..15) for all buttons in the specified group. You can use different values for odd and even buttons (based on their index within the group, not their id) - this can be a good way to provide some visual guidance.

G.GBTN.C

- `G.GBTN.C group`

Get the total count of all the buttons in the specified group that are currently pressed.

G.GBTN.I

- `G.GBTN.I group index`

Get the id of a currently pressed button within the specified group by its index (0-based). The index should be between 0 and C-1 where C is the total count of all pressed buttons (you can get it using `G.GBTN.C`).

G.GBTN.W

- `G.GBTN.W group`

Get the width of the rectangle formed by pressed buttons within the specified group. This is basically the distance between the leftmost and the rightmost pressed buttons, inclusive. This op is useful for things like setting a loop's length, for instance. To do so, check if there is more than one button pressed (using `G.GBTN.C`) and if there is, use `G.GBTN.W` to set the length.

G.GBTN.H

- `G.GBTN.H group`

Get the height of the rectangle formed by pressed buttons within the specified group (see `G.GBTN.W` for more details).

G.GBTN.X1

- `G.GBTN.X1 group`

Get the X coordinate of the leftmost pressed button in the specified group. If no buttons are currently pressed it will return -1.

G.GBTN.X2

- `G.GBTN.X2 group`

Get the X coordinate of the rightmost pressed button in the specified group. If no buttons are currently pressed it will return -1.

G.GBTN.Y1

- `G.GBTN.Y1 group`

Get the Y coordinate of the highest pressed button in the specified group. If no buttons are currently pressed it will return -1.

G.GBTN.Y2

- `G.GBTN.Y2 group`

Get the Y coordinate of the lowest pressed button in the specified group. If no buttons are currently pressed it will return -1.

G.FDR

- `G.FDR id x y w h type level script`

Initializes and enables a fader with the specified id. 64 faders are available (ids are 0-based so the possible id range is 0..63). The fader will be assigned to the current group (set with `G.GRP`). Faders can be reinitialized at any point.

`x` and `y` specify the coordinates of the top left corner, and `w` and `h` specify width and height respectively.

`type` determines the fader type and orientation. Possible values are:

- 0 - coarse, horizontal bar
- 1 - coarse, vertical bar
- 2 - coarse, horizontal dot
- 3 - coarse, vertical dot
- 4 - fine, horizontal bar
- 5 - fine, vertical bar
- 6 - fine, horizontal dot
- 7 - fine, vertical dot

Coarse faders have the possible range of 0..N-1 where N is width for horizontal faders or height for vertical faders. Pressing anywhere within the fader area sets the fader value accordingly. Fine faders allow selecting a bigger range of values by mapping the range to the fader's height or width and dedicating the edge buttons for incrementing/decrementing. Fine faders employ varibrightness to reflect the current value.

level has a different meaning for coarse and fine faders. For coarse faders it selects the background brightness level (similar to buttons). For fine faders this is the maximum value level (the minimum level being 0). In order to show each value distinctly using varibright the maximum level possible is the number of available buttons multiplied by 16 minus 1 (since range is 0-based). Remember that 2 buttons are always reserved for increment/decrement. Using a larger number is allowed - it will be automatically adjusted to what's possible.

script specifies the script to be executed when the fader value is changed. Use 9 for Metro and 10 for Init. Use 0 if you don't need a script assigned.

G.GFD

- `G.GFD grp id x y w h type level script`

Initialize and enable a fader. Same as **G.FDR** but you can also choose which group to assign the fader too.

G.FDX

- `G.FDX id x y w h type level script columns rows`

Initialize and enable a block of faders with the specified number of columns and rows in the current group. Ids are incremented sequentially by columns and then by rows.

G.GFX

- `G.GFX group id x y w h type level script columns rows`

Initialize and enable a block of faders. Same as **G.FDX** but you can also choose which group to assign the faders too.

G.FDR.EM

- `G.FDR.EM id (x)`

Enable (set `x` to 1) or disable (set `x` to 0) a fader with the specified id, or check if it's currently enabled. Disabling a fader hides it and stops it from receiving input

G.FDR.L

- `G.FDR.L id (level)`

Get or set the specified fader's brightness level (for coarse faders), or the maximum value level (for fine faders).

G.FDRI

- `G.FDRI`

Get the id of the last pressed fader. This is useful when multiple faders are assigned to the same script.

G.FDRX

- `G.FDRX (x)`

Get or set `x` coordinate of the last pressed fader's top left corner. This is the same as `G.FDR.X G.FDRI`.

G.FDRY

- `G.FDRY (y)`

Get or set `y` coordinate of the last pressed fader's top left corner. This is the same as `G.BTN.Y G.BTNI`.

G.FDRM

- `G.FDRM (value)`

Get or set the value of the last pressed fader. This is the same as `G.FDR.M G.FDRI`. See `G.FDR.M` for more details.

G.FDRV

- `G.FDRV (value)`

Get or set the scaled value of the last pressed fader. This is the same as `G.FDR.V G.FDRI`. See `G.FDR.V` for more details.

G.FDRL

- `G.FDRL (level)`

Get or set the brightness level (for coarse faders), or the maximum value level (for fine faders) of the last pressed fader. This is the same as **G.FDR.L G.BTNI**. For more details on levels see **G.FDR**.

G.FDR.PR

- `G.FDR.PR id value`

Emulate pressing the specified fader. Fader value will be set to the specified value, and if there is a script assigned it will be executed.

G.GFDR.M

- `G.GFDR.M group value`

Set the value for all faders in the specified group. This can be useful for resetting all faders in a group. See **G.FDR.M** for more details.

G.GFDR.V

- `G.GFDR.V group value`

Set the scaled value for all faders in the specified group. This can be useful for resetting all faders in a group. See **G.FDR.V** for more details.

G.GFDR.L

- `G.GFDR.L group odd_level even_level`

Set the brightness level (0..15) for all faders in the specified group. You can use different values for odd and even faders (based on their index within the group, not their id) - this can be a good way to provide some visual guidance.

G.GFDR.RM

- `G.GFDR.RM group min max`

Set the range to be used for **V** fader values (**G.FDR.V**, **G.FDRV**, **G.GFDR.V**). While the **.M** ops provide the actual fader value sometimes it's more convenient to map it to a different range so it can be used directly for something like a CV without having to scale it each time.

An example: let's say you create a coarse fader with the width of 8 which will be used to control a CV output where the voltage must be in the 2V..5V range. Using `G.FDR.M` you would need to do this: `CV 1 SCL 0 7 V 2 V 5 G.FDR.M 0`. Instead you can set the range for scaling once: `G.GFDR.RM 0 V 2 V 5` (assuming the fader is in group 0) and then simply do `CV 1 G.FDR.V 0`.

The range is shared by all faders within the same group. If you need to use a different range use a different group when initializing a fader.

The default range is 0..16383. `G.RST` and `G.GRP.RST` reset ranges to the default value.

MIDI in

MIDI in ops allow the Teletype to react to MIDI events. MIDI is received via the USB port - simply plug a MIDI controller or sequencer into the USB port. Unless your MIDI device is powered externally, make sure your power supply can provide sufficient power! Please note that not all devices are supported.

To use the MIDI in ops, you need to assign MIDI events to one of the scripts with `MI.$` op. You can assign different event types to different scripts (so, script 1 could react to Note On events and script 2 to Note Off, for instance). You can assign multiple event types to the same script too. Various ops allow you to get detailed information about the event type and any additional data. It's possible that more than one event happens before a script is called (say, if you turn multiple knobs at once or play chords). To properly process them all, use indexed ops to get each event data instead of only processing the last event. The indexed ops use variable `I` as the index to allow easy use in loops.

OP *(set)*	*(aliases)*	Description
`MI.$ x (y)`		assign MIDI event type `x` to script `y`
`MI.LE`		get the latest event type
`MI.LCH`		get the latest channel (1..16)
`MI.LN`		get the latest Note On (0..127)
`MI.LNV`		get the latest Note On scaled to teletype range (shortcut for `N MI.LN`)
`MI.LV`		get the latest velocity (0..127)
`MI.LVV`		get the latest velocity scaled to 0..16383 range (0..+10V)
`MI.LO`		get the latest Note Off (0..127)
`MI.LC`		get the latest controller number (0..127)
`MI.LCC`		get the latest controller value (0..127)
`MI.LCCV`		get the latest controller value scaled to 0..16383 range (0..+10V)
`MI.NL`		get the number of Note On events
`MI.NCH`		get the Note On event channel (1..16) at index specified by variable `I`
`MI.N`		get the Note On (0..127) at index specified by variable `I`
`MI.NV`		get the Note On scaled to 0..+10V range at index specified by variable `I`
`MI.V`		get the velocity (0..127) at index specified by variable `I`

OP *(set)*	*(aliases)*	Description
MI.VV		get the velocity scaled to 0..10V range at index specified by variable I
MI.OL		get the number of Note Off events
MI.OCH		get the Note Off event channel (1..16) at index specified by variable I
MI.O		get the Note Off (0..127) at index specified by variable I
MI.CL		get the number of controller events
MI.CCH		get the controller event channel (1..16) at index specified by variable I
MI.C		get the controller number (0..127) at index specified by variable I
MI.CC		get the controller value (0..127) at index specified by variable I
MI.CCV		get the controller value scaled to 0..+10V range at index specified by variable I
MI.CLKD (x)		set clock divider to x (1-24) or get the current divider
MI.CLKR		reset clock counter

MI.$

- MI.$ x (y)

Assign a script to be triggered when a MIDI event of the specified type is received. The following types are supported: 0 - all events 1 - note on 2 - note off 3 - controller change 4 - clock 5 - start 6 - stop 7 - continue

Calibration

OP *(set)*	*(aliases)*	Description
DEVICE.FLIP		Flip the screen/inputs/outputs
IN.CAL.MIN		Reads the input CV and assigns the voltage to the zero point
IN.CAL.MAX		Reads the input CV and assigns the voltage to the max point
IN.CAL.RESET		Resets the input CV calibration
PARAM.CAL.MIN		Reads the Parameter Knob minimum position and assigns a zero value
PARAM.CAL.MAX		Reads the Parameter Knob maximum position and assigns the maximum point
PARAM.CAL.RESET		Resets the Parameter Knob calibration
CV.CAL n mv1v mv3v		Calibrate CV output n
CV.CAL.RESET n		Reset calibration data for CV output n

DEVICE.FLIP

- DEVICE.FLIP

Flip the screen, the inputs and the outputs. This op is useful if you want to mount your Teletype upside down. The new state will be saved to flash.

IN.CAL.MIN

- IN.CAL.MIN

1. Connect a patch cable from a calibrated voltage source
2. Set the voltage source to 0 volts
3. Execute IN.CAL.MIN from the live terminal
4. Call IN and confirm the 0 result

IN.CAL.MAX

- IN.CAL.MAX

5. Set the voltage source to target maximum voltage (10V)
6. Execute IN.CAL.MAX from the live terminal
7. Call IN and confirm that the result is 16383

PARAM.CAL.MIN

- `PARAM.CAL.MIN`

1. Turn the PARAM knob all the way to the left
2. Execute PARAM.CAL.MIN from the live terminal
3. Call PARAM and confirm the 0 result

PARAM.CAL.MAX

- `PARAM.CAL.MAX`

4. Turn the knob all the way to the right
5. Execute PARAM.CAL.MAX from the live terminal
6. Call PARAM and verify that the result is 16383

CV.CAL

- `CV.CAL n mv1v mv3v`

Following a short calibration procedure, you can use `CV.CAL` to more precisely match your CV outputs to each other or to an external reference. A digital multimeter (or other voltage measuring device) is required.

To calibrate CV 1, first set it to output one volt with `CV 1 V 1`. Using a digital multimeter with at least millivolt precision (three digits after the decimal point), record the measured output of CV 1 between tip and sleeve on a patch cable. Then set CV 1 to three volts with `CV 1 V 3` and measure again.

Once you have both measurements, use the observed 1V and 3V values in millivolts as the second and third arguments to `CV.CAL`. For example, if you measured 0.990V and 2.984V, enter `CV.CAL 1 990 2984`. (If both your measurements are within 1 or 2 millivolts already, there's no need to run `CV.CAL`.)

Measure the output with `CV 1 V 1` and `CV 1 V 3` again and confirm the values are closer to the expected 1.000V and 3.000V.

Repeat the above steps for CV 2-4, if desired. The calibration data is stored in flash memory so you only need to go through this process once.

Note: The calibration adjustment is made after `CV.SLEW` and `CV.OFF` are applied, and does not affect `CV.GET` or any other scene-visible values. It only affects the levels coming out of the DAC.

CV.CAL.RESET

- `CV.CAL.RESET n`

Clear the calibration data for CV output n and return it to its default behavior, with no calibration adjustment.

Generic I2C

Generic I2C ops allow querying and sending commands to any I2C enabled devices connected to teletype. Before you can send or query you need to set the I2C address of the device using **II.A** (you might want to place that in your INIT script so that the address is set when you load a scene).

You can send up to 3 additional parameters, which can be either byte values or full range teletype values (for something like velocity), which will get sent as 2 bytes (MSB followed by LSB). All parameters must be of the same type - if you need to send both byte and word values, use the bitshift ops to combine/split bytes.

No validation or transformation is applied to any of the parameters - they are send as is. As dedicated ops are often 1-based, you might want to subtract 1 when reproducing them with the generic ops.

There are 2 sets of query ops - one for getting regular (word) values and one for getting byte values. If the address is not set, or if it's set but there are no follower devices listening at that address, query ops will return zero.

OP *(set)*	*(aliases)*	Description
`II.A (address)`		Set I2C address or get the currently selected address
`II.S cmd`		Execute the specified command
`II.S1 cmd value`		Execute the specified command with 1 parameter
`II.S2 cmd value1 value2`		Execute the specified command with 2 parameters
`II.S3 cmd value1 value2 value3`		Execute the specified command with 3 parameters
`II.SB1 cmd value`		Execute the specified command with 1 byte parameter
`II.SB2 cmd value1 value2`		Execute the specified command with 2 byte parameters
`II.SB3 cmd value1 value2 value3`		Execute the specified command with 3 byte parameters
`II.Q cmd`		Execute the specified query and get a value back
`II.Q1 cmd value`		Execute the specified query with 1 parameter and get a value back
`II.Q2 cmd value1 value2`		Execute the specified query with 2 parameters and get a value back

OP *(set)*	*(aliases)*	Description
`IIQ3 cmd value1 value2 value3`		Execute the specified query with 3 parameters and get a value back
`IIQB1 cmd value`		Execute the specified query with 1 byte parameter and get a value back
`IIQB2 cmd value1 value2`		Execute the specified query with 2 byte parameters and get a value back
`IIQB3 cmd value1 value2 value3`		Execute the specified query with 3 byte parameters and get a value back
`IIB cmd`		Execute the specified query and get a byte value back
`IIB1 cmd value`		Execute the specified query with 1 parameter and get a byte value back
`IIB2 cmd value1 value2`		Execute the specified query with 2 parameters and get a byte value back
`IIB3 cmd value1 value2 value3`		Execute the specified query with 3 parameters and get a byte value back
`IIBB1 cmd value`		Execute the specified query with 1 byte parameter and get a byte value back
`IIBB2 cmd value1 value2`		Execute the specified query with 2 byte parameters and get a byte value back
`IIBB3 cmd value1 value2 value3`		Execute the specified query with 3 byte parameters and get a byte value back

IIA

- `IIA (address)`

Set the I2C address to be used by the generic I2C ops. The address is -1 when not selected or when it's set to a value outside of the supported range.

Ansible

OP *(set)*	*(aliases)*	Description
AMS.G.LED x y		get grid LED buffer at position x, y
AMS.G x y (z)		get/set grid key on/off state (z) at position x, y
AMS.G.P x y		simulate grid key press at position (x, y)
AMS.A.LED n x		read arc LED buffer for ring n, LED x clockwise from north
AMS.A (d)		send arc encoder event for ring n, delta d
AMS.APP (x)		get/set active app
KR.PRE (x)		return current preset / load preset x
KR.PERIOD (x)		get/set internal clock period
KR.PAT (x)		get/set current pattern
KR.SCALE (x)		get/set current scale
KR.POS x y (z)		get/set position z for track x, parameter y
KR.L.ST x y (z)		get loop start for track x, parameter y / set to z
KR.L.LEN x y (z)		get length of track x, parameter y / set to z
KR.RES x y		reset position to loop start for track x, parameter y
KR.CV x		get the current CV value for channel x
KR.MUTE x (y)		get/set mute state for channel x (1 = muted, 0 = unmuted)
KR.TMUTE x		toggle mute state for channel x
KR.CLK x		advance the clock for channel x (channel must have teletype clocking enabled)
KR.PG (x)		get/set the active page
KR.CUE (x)		get/set the cued pattern
KR.DIR (x)		get/set the step direction
KR.DUR x		get the current duration value for channel x
ME.PRE (x)		return current preset / load preset x
ME.SCALE (x)		get/set current scale
ME.PERIOD (x)		get/set internal clock period
ME.STOP x		stop channel x (0 = all)
ME.RES x		reset channel x (0 = all), also used as “start”

OP *(set)*	*(aliases)*	Description
ME.CV x		get the current CV value for channel x
LV.PRE (x)		return current preset / load preset x
LV.RES x		reset, 0 for soft reset (on next ext. clock), 1 for hard reset
LV.POS (x)		get/set current position
LV.L.ST (x)		get/set loop start
LV.L.LEN (x)		get/set loop length
LV.L.DIR (x)		get/set loop direction
LV.CV x		get the current CV value for channel x
CY.PRE (x)		return current preset / load preset x
CY.RES x		reset channel x (0 = all)
CY.POS x (y)		get / set position of channel x (x = 0 to set all), position between 0-255
CY.REV x		reverse channel x (0 = all)
CY.CV x		get the current CV value for channel x
MID.SLEW t		set pitch slew time in ms (applies to all allocation styles except FIXED)
MID.SHIFT o		shift pitch CV by standard Teletype pitch value (e.g. M 6, V -1, etc)
ARP.HLD h		0 disables key hold mode, other values enable
ARP.STY y		set base arp style [0-7]
ARP.GT v g		set voice gate length [0-127], scaled/synced to course divisions of voice clock
ARP.SLEW v t		set voice slew time in ms
ARP.RPT v n s		set voice pattern repeat, n times [0-8], shifted by s semitones [-24, 24]
ARP.DIV v d		set voice clock divisor (euclidean length), range [1-32]
ARP.FIL v f		set voice euclidean fill, use 1 for straight clock division, range [1-32]
ARP.ROT v r		set voice euclidean rotation, range [-32, 32]
ARP.ER v f d r		set all euclidean rhythm
ARP.RES v		reset voice clock/pattern on next base clock tick

ARP.SHIFT v o	shift voice cv by standard tt pitch value (e.g. N 6, V -1, etc)

AMS.APP

- AMS.APP (x)

Get or change the app that is active on Ansible. Numbering:

- 0 = levels
- 1 = cycles
- 2 = kria
- 3 = meadowphysics
- 4 = midi standard
- 5 = midi arp
- 6 = teletype expander

KR.POS

- KR.POS x y (z)

Set position to z for track x, parameter y.

A value of 0 for x means all tracks.

A value of 0 for y means all parameters

Parameters:

- 0 = all
- 1 = trigger
- 2 = note
- 3 = octave
- 4 = length

KR.PG

- KR.PG (x)

Get or change the current parameter page. Numbering:

- 0 = trigger
- 1 = ratchet
- 2 = note
- 3 = alt note
- 4 = octave

- 5 = glide
- 6 = duration
- 7 = TBD
- 8 = scale
- 9 = pattern

KR.CUE

- **KR.CUE (x)**

Get or change the cued pattern. Numbered from 0.

KR.DIR

- **KR.DIR (x)**

Get or change the step direction. Numbered from 0.

Just Type

More extensively covered in the Just Friends Documentation[15].

Copied below, modified and stripped of crow commands and unimplemented (proposed) teletype commands.

monome's *Teletype* (and since, *crow*) excites us. Writing simple scripts away from a computer; executing tiny morsels of musical composition. Just Type is a suggestion for how these ideas can be extended & deeply integrated with elements of synthesis.

At it's most basic, Just Type is a set of invisible patch cords. But more than a cloaking device, it extends the base functionality of Just Friends into more complex territory. Every output can be driven with varying velocity, and the INTONE relationship can be altered away from the default harmonic structure.

Beyond these general-purpose modifications, Just Type brings two entirely new modalities. *Synthesis* allows explicit or automatic polyphonic control over each channel. *Geode* instead pursues rhythmic manipulation of striated repetitions, creating polymetric bursts with dynamic decay.

Enough! Just. Type.

Install?

Just Type has shipped on every new module made since late 2017. A substantial update was released in July 2020, opening up a host of new features, and fixing many shortcomings of the original formulation.

Get the latest release[16] and you'll be all set to type!

On reading this doc

Each command has a description of the names & syntax for usage. Both Teletype & crow syntax is displayed and will take the following form:

`JF.COMMAND value` (set)
`JF.COMMAND` (get)

[15] https://github.com/whimsicalraps/Just-Friends/blob/main/Just-Type.md
[16] https://github.com/whimsicalraps/Just-Friends/releases/latest

Each command has 1 or 2 forms listed. These can be of 2 types:

- 'setters' tell Just Friends to do something without expecting a response.
- 'getters' ask Just Friends to return a value.

Some commands are 'set' only, while others and 'get' only, but many have both functionalities. What's important is to recognize which option you need in your script.

Furthermore, getters work quite differently on Teletype vs crow. For Teletype, the getter will query the value and return it directly. On crow, the response to a **ii.jf.get()** call will come through the **ii.jf.event(event, value)** function which you must add to your script. For an example, you can call **ii.jf.help()** and crow will show you what this event function should look like.

Remote control

These commands allow remote control over Just Friends. Imagine a set of invisible patch cables connected to the TRIGGERS and RUN jacks.

Triggers

JF.TR CHANNEL STATE

Create a TRIGGER event on the **CHANNEL** with the provided **STATE**.

- **CHANNEL**
  - 1 is IDENTITY, and 6 is 6N
  - 0 creates a TRIGGER on all 6 channels (hardware normalization is ignored)
- **STATE**
  - 1 is *high* (5V). All non-zero values are treated as high
  - 0 is *low* (0V)
- Only *sustain* cares about the *low* triggers (the others modes will simply ignore this message).

Run Mode

JF.RMODE MODE

Set the RUN state of Just Friends when no physical jack is present.

- **MODE**
  - 1 activates RUN mode. All non-zero values are treated as high
  - 0 deactivates RUN. If a physical jack is present, RUN stays high.

Run Voltage

JF.RUN VOLTS

Send a virtual voltage to the RUN input.

- VOLTS
  - The voltage to be virtually sent to the RUN jack. Adds to the physical state
  - On TT use JF.RUN V x to set to x volts.
  - Range is -5 to +5
- *Requires* JF.RMODE 1 *to have been executed*.

Extended behaviour

This collection of commands extend the base capabilities of Just Friends, without totally changing the mode of interaction. They let you interact with the module in subtly different ways, and consider alternative approaches to creating compositional structure.

Transposition

JF.SHIFT PITCH

Shifts the transposition of Just Friends, regardless of speed setting. Shifting by V 1 doubles the frequency in *sound*, or doubles the rate in *shape*.

- PITCH
  - Amount to shift base pitch by
  - Use M x for semitones, or V y for octaves
  - Microtonal transpositions are allowed (especially useful for tuning)

Velocity

JF.VTR CHANNEL LEVEL

Trigger *channel* with velocity set by *level*. Like Trigger but with added volume control. Velocity is scaled with volts, so try V 5 for an output trigger of 5 volts. Channels remember their latest velocity setting and apply it regardless of whether the TRIGGER comes digitally or via CV.

- CHANNEL
  - channel to trigger
  - 0 sets all channels to the same velocity
- LEVEL
  - amplitude of output in volts

- 0 is treated as a 'low' state, and doesn't change the saved velocity (same as `JF.TR CHANNEL 0`)

Tuning and Intonation

`JF.TUNE CHANNEL n d`

Adjust the tuning ratios used by the INTONE control. The default for this is just the first 6 elements of the harmonic series. Instead you can retune this to your needs or desires. Think a guitar with open-G tuning – lends itself to an entirely different style of play.

Tuning is defined as a pitch ratio: numerator `n` / denominator `d`. 1/1 is IDENTITY, whereas 4/1 would be 4N's default setting. Read a little about just intonation for some ideas how you might utilise this feature.

- `CHANNEL`
  - select which channel's tuning to redefine (1 through 6)
- `n` (numerator)
  - set the multiplier for the tuning ratio
- `d` (denominator)
  - set the divisor for the tuning ratio
- *Reset to default tuning with* `JF.TUNE 0 0 0`
- If you want to retain your custom tuning permanently (across power-cycles), send the special command `JF.TUNE -1 0 0` which will store the setup to memory. When you restart the module, it will automatically recall the custom tuning.

Address (communicating with two Just Friends simultaneously)

`JF.ADDR index`

This is only useful when configuring an ii network with two Just Friends device. Note that all devices default to index 1.

1. Power down your case
2. Disconnect the Just Friends that will be index #1 from the i2c bus
3. Make sure your second device is connected
4. Power on the case
5. Run the above address command with an index of 2: `JF.ADDR 2`
6. Test you can now talk to the second device: `JF2.TR 1 1`
7. Power down the case
8. Reconnect the Just Friends from step 2.
9. Power on the case

Now you can refer to your two devices like so. Teletype will use the `JF2` prefix instead of `JF`:

```
JF.TR 1 1
JF2.TR 1 1
```

Panel queries

The physical panel settings are able to be queried too. With some outside-the-box thinking, you can use the Just Friends panel to manipulate parameters inside your script. This could augment the controls (eg. the *CURVE* value could change `ptrigger` level), or introduce additional dimensions (eg. *FM* could select different `TUNE` ratios).

While the lower 3 jacks (*RAMP*, *FM*, *CURVE*) send only the knob position, *TIME* and *INTONE* send a combination of the knob with any received CV. These signals are also mapped to the same scaling as Just Type in *Synthesis* mode (see below). That means, a value of `V 0` is equal to C3. As such you could rapidly query the *TIME* control and convert it to a control-voltage with Teletype or crow - allowing for Just Friends to control the base pitch of multiple oscillators.

Finally the parameters can be used entirely tangentially to Just Friends' functionality. *RAMP* could control the rate of a METRO, while *sound*/*shape* choose between major and minor arpeggios.

`JF.SPEED`

- Returns the current *shape* (0) or *sound* (1) switch position

`JF.TSC`

- Returns the current *MODE* switch state
  - 1 = *transient*
  - 2 = *sustain*
  - 3 = *cycle*

`JF.RAMP`

- Returns the current state of the *RAMP* knob in volts (-5,5)

`JF.CURVE`

- Returns the current state of the *CURVE* knob in volts (-5,5)

`JF.FM`

- Returns the current state of the *FM* knob in volts (-5,5)

`JF.TIME`

- Returns the current state of the *TIME* knob + cv in volts (-5,5)

JF.INTONE

- Returns the current state of the *INTONE* knob + cv in volts
  - 0 = C3
  - 1V/octave scaled

Modal personality

Until now, we've only been speaking of modifying or extending the base Just Friends behaviours. Conversely, it is also possible to change some fundamentals of the JF system, leaning more heavily on the Teletype / crow integration for configuration and control.

These alternate personalities are *Synthesis*, a polyphonic synthesizer; and *Geode*, a rhythm machine. `JF.MODE 1` will take you to these modes depending on the *sound*/*shape* setting. Beware that whilst in Just Type's alternate modes, things will behave differently to normal & will remain there until power-cycling or exiting with `JF.MODE 0`.

JF.MODE STATE

Activates *Synthesis* or *Geode* modalities.

- `STATE`
  - 1 activates JT alternate modes. Any non-zero value is treated as 1.
  - 0 returns to standard functionality

You'll likely want to put `JF.MODE x` in your INIT script.

Synthesis mode

Synthesis is, as its name boringly suggests, a synthesizer. Further, it is a polyphonic synthesizer of six independent voices. Control is either explicitly per voice, or can be dynamically assigned in a traditional polysynth fashion.

Enter *Synthesis* with `JF.MODE 1` and switching to *sound*.

The voices are centered around Just Friends' manifold generators, with pitch controlled digitally rather than with the panel controls. Each generator is shaped by *RAMP* & *CURVE* as per normal, then passed to a Vactrol Low-Pass Gate model to impart dynamics. The Vactrol model implements rudimentary envelope shaping of the velocity, controlled by *TIME*, for envelope speed, and *INTONE*, for attack-release shaping. These envelopes are controlled by the *transient* / *sustain* / *cycle* switch, and may be excited either digitally or via the hardware *TRIGGERS*.

Internally each voice contains a linked sinewave oscillator providing frequency modulation over the function generator. FM index (ie. amount), is controlled

with the *FM* knob & CV input. The knob functions as normal with INTONE-style modulation CCW, and uniform modulation CW. *FM* CV input is a traditional CV-offset where positive voltage increases, and negative decreases modulation. The frequency relationship between the modulation & carrier oscillators is set via the *RUN* jack, though is matched at 1:1 with no cable attached. Positive voltages move toward 2:1 at 5V, while negative sweeps down to 1:2 at -5V giving many grumbles.

The **VOX** and **NOTE** commands are designed to create complete notes in the General MIDI sense. They simultaneously set the pitch of a voice & begin / end an envelope cycle. Physical **TRIGGERS** on the other hand, will only trigger the envelope, using whatever pitch & velocity are currently set for that voice, encouraging combinations of digital & voltage control. Pitch can be set directly with **PITCH** to slew between tones without triggering notes.

Individual voice control (6 monosynth voices)

JF.VOX CHANNEL PITCH LEVEL

Play a note on the specified *channel* at the defined *pitch* and *level*.

- **CHANNEL**
  - Assign to channel 1 through 6
  - 0 sets all channels simultaneously.
- **PITCH**
  - set the pitch in 1V/octave
  - **V 0** is C3
- **LEVEL**
  - set the volume as in **VTR**
  - **V 5** gives 5V peak to peak (ie. standard modular level)

Assigning notes with voice control is great if you want to sequence independent synthesizer lines on the different channels. By using the *FM* control in the counter-clockwise direction, you can give each voice its own character, as the higher-numbered voices will have greater frequency modulation applied.

Dynamic voice allocation (6-voice polysynth)

JF.NOTE PITCH LEVEL

Polyphonically allocated note sequencing. Works like **VOX** but with *channel* selected automatically. In *sustain*, free voices will be prioritized. If all voices are currently sustaining, the oldest note will be stolen to play the new note.

- **PITCH**
  - set the pitch in 1V/octave

    - V 0 is C3
- LEVEL
    - set the volume as in VTR
    - V 5 gives 5V peak to peak (ie. standard modular level)

VOX and NOTE are interactive, meaning you can create interesting splits of mono and poly voices.

Try turning the FM knob counter-clockwise and sequencing a fixed cycle of tones– The timbre of the voices will cycle through 6 levels of FM depth. This can be great with 6 tones where the note order chooses timbre, but even more interesting if your sequence is 5 or 7 steps long, creating long phasing patterns for movement in a simple arpeggio.

Pitch Control (portamento)

JF.PITCH CHANNEL PITCH

Control the *pitch* of a chosen *channel* without triggering the envelope (like JF.VOX). - CHANNEL - Assign to channel 1 through 6 - 0 sets all channels simultaneously.

- PITCH
    - set the pitch in volts-per-octave
    - V 0 is C3

This command is useful along with VOX & NOTE control to introduce pitch changes while a note is decaying (*transient*), or without retriggering the cyclic envelope (*cycle*). Additionally it can be very useful where you are *TRIGGER*ing the channels with CV pulses, but want to choose scales or chords digitally.

Attuned Vibrations

JF.GOD STATE

Redefines C3 to align with the 'God' note. See: https://attunedvibrations.com/432hz/ or http://www.roelhollander.eu/en/tuning-frequency/goebbels-and-440/.

- STATE
    - 0 is A=440Hz
    - 1 is A=432Hz

Geode

In *shape*, Just Type inherits it's functionality from the standard mode. However, atop it sits a rhythmic engine for polymetric & -phasic patterns. Fundamentally this is a 'clocked' mode, whether internally so or via a continuous *tick*. The TIME & INTONE controls maintain their standard free-running influence, speeding up and slowing down *envelopes*, while the rhythms are controlled remotely.

Notes in Geode are a combination of a standard trigger along with a number of *repeats* & a rhymthic *division*. The former sets the number of envelope events to create, while the latter chooses the rhythmic relation of those repeats to the core timebase. The *MODE* switch selects how the amplitudes of repeated elements change over time. These changes are further modified by the RUN jack for fluid rhythmic variation under voltage control. These undulations are highly interactive with the *TIME* & *INTONE* controls, where the different MODE settings will handle overlapping repeats in drastically different ways. Start with *TIME* set very fast, then dial it back to hear how the repeats entangle.

Once these rhythmic streams are moving, their pattern can be corralled into a set of *quantized* steps. Using odd-subdivisions for notes, with even quantize, will enable patterns to break out of the evenly-spaced-repeats model. Try prime numbers (5/7/11) for divisions, but 4/8/16 for quantize to create traditional syncopated rhythms.

Geode: Transient

When set to *transient*, each repeat will have the full velocity (or a reduced one set by VTR).

RUN voltages will introduce a rhythmic variation every *n* repeats. At 0V, every note is emphasized (hence sounding static). Increasing a little and every 2nd note is emphasized. Further and a cycle of 3 velocities is introduced, and so on up to a cycle of 10 notes. The velocities decrease in a 'sawtooth' pattern.

With negative voltages, the same cycles are introduced, however the pattern is reversed, dropping the volume at first, then rising up over *n* repeats.

Regular syncopated rhythms are great here. Try 8 *repeats*, triggered every 8 clocks and choose a rhythm with RUN.

Geode: Sustain

In *sustain* repeats decay to silence over the duration of *repeats*.

By adding a RUN voltage the rate of decay can be modified. At 0V it takes exactly *repeats* to fade away. As RUN increases the repeats fade more quickly, however they will reflect back up when hitting the minimum. With around 1V the repeats

will decay to near zero, then back to full volume by the last repeat, creating a triangle shape. Further increasing this level 'folds' into multiple waves per set of repeats.

Negative values slow the decay rate, making the fade out effect more and more subtle. At -5V the amplitudes are almost uniform.

Creative use of this behaviour can introduce a third temporal element to the Geode equation: *TIME* & *INTONE* set a base envelope rate, *divisions* sets the rhythm of notes, and *RUN* sets the amplitude cycle relative to *repeats*.

This mode is useful for creating pseudo-delay envelopes.

Geode: Cycle

cycle introduces a complex, *repeats* sensitive amplitude cycling. The rhythm is generated similarly to *transient*, however the variation is applied continuously rather than in single steps of beats.

Applying RUN voltage emphasizes every 2nd then 3rd then 4th event, however all the in-between beats are available too. Subtle CV shifts allow for a variation of *groove* with nothing but volume manipulation.

Negative RUN levels emphasize every fraction of a beat, which is a hard thing to think about, let alone control. As the voltage becomes lower the rhythmic cycles change more rapidly, starting to feel random. This zone is great to explore if you want to introduce some unpredictability into a rhythmic pattern.

This mode is a source of endless subtle movement and works extra well with **QT** active.

Percussive Timebase

Geode needs a timebase from which to calculate the rates for the envelope sequences. This base can be set with a continuous stream of events (ie a clock) useful for when you need to synchronize the events to other elements. Alternatively a simple beats-per-minute value can be used if Just Friends is free running and doesn't need to play in time with others.

JF.TICK DIV5

Clock *Geode* with a stream of **DIV** ticks per measure.

- **DIV**
  - Tells Just Type how many **tick** messages will be received per measure, where a measure is 4 beats.
  - 1 to 48 ticks per measure are allowed
  - 4 means 1 tick per beat

    - 0 acts a reset to synchronize to the start of the measure

Typically `JF.TICK` will be called in a METRO. For 60 bpm, you can send `JF.TICK 4` once per second, or `JF.TICK 8` twice per second etc. Once you are comfortable using it in a standard way, the `DIV` value can be modulated to create rhythmically related clock multiplications and divisions of the *repeats*.

`JF.TICK BPM`

Set timebase for *Geode* with a static `BPM`.

- `BPM`
    - Number of beats per minute where a measure is 4 beats.
    - Must be between 49 and 255 bpm.
    - 0 acts a reset to synchronize to the start of the measure

Individual Rhythms

`JF.VOX CHANNEL DIV REPEATS`

Create a stream of rhythmic envelopes on the named `CHANNEL`. The stream will continue for the count of `REPEATS` at a rhythm defined by `DIV`.

- `CHANNEL`
    - select the channel to assign this rhythmic stream
    - 0 sets all channels
- `DIV`
    - Divides the measure into this many segments
    - 4 creates quarter notes
    - 15 creates 15 equally spaced notes per bar (weird!)
- `REPEATS`
    - Number of times to retrigger the envelope
    - -1 repeats indefinitely
    - 0 will still create the initial trigger but no repeats
    - 1 will make 2 events total (the initial trigger, and 1 repeat)

Round-Robin Rhythms

`JF.NOTE DIV REPEATS`

Works as `JF.VOX` but with dynamic allocation of channel. Assigns the rhythmic stream to the next channel.

- `DIV`
    - Divides the measure into this many segments
    - 4 creates quarter notes
    - 15 creates 15 equally spaced notes per bar (weird!)

- REPEATS
  - Number of times to retrigger the envelope
  - -1 repeats indefinitely
  - 0 will still create the initial trigger but no repeats
  - 1 will make 2 events total (the initial trigger, and 1 repeat)

Time Quantization

JF.QT DIV

Quantize *Geode* events to DIV of a measure.

When non-zero, all events are queued & delayed until the next quantize event occurs. Using values that don't align with the division of rhythmic streams will cause irregular patterns to unfold.

- DIV
  - delay all events until this division of the timebase
  - 0 deactivates quantization
  - 1 to 32 sets the subdivision & activates quantization

If you need your rhythms to stay on a regular grid, activate that grid with Quantization. By setting a regular quantization (try 8 or 16) you can experiment with irregular DIV when triggering POX or NOTE (try 7, 11, 13, 15) and those repeats will be locked into the quantized grid. Couple this with dynamic control over RUN and you have a very powerful groove generator with a few high level controls. Instant percussion inspiration!

While it uses a different implementation, this functionality can create Euclidean rhythms[17], though 'rotating' the rhythms requires delaying the POX or NOTE calls.

Teletype Reference

Set values or call actions:

JF.TR CHANNEL STATE: Set trigger channel to STATE
JF.RMODE MODE: Set RUN state to MODE
JF.RUN VOLTS: Set RUN voltage to VOLTS
JF.SHIFT VOLTS: Transpose frequency / speed by VOLTS (v/8)
JF.VTR CHANNEL LEVEL: Trigger CHANNEL with velocity set by LEVEL
JF.TUNE CHANNEL NUMERATOR DENOMINATOR: Alter the INTONE relationship to IDENTITY
JF.MODE STATE: Activates *Synthesis* or *Geode* JF.ADDR INDEX Set all connected

[17] https://splice.com/blog/euclidean-rhythms/

Just Friends to ii address **INDEX**

Synthesis:

JF.NOTE PITCH LEVEL: Play a note, dynamically allocated to a voice
JF.VOX CHANNEL PITCH LEVEL: Play a note on a specific voice
JF.GOD STATE: If **STATE**, retune to A=432Hz (default A=440Hz) **JF.PITCH CHANNEL PITCH**: Same as **JF.VOX** but doesn't trigger the envelope

Geode:

JF.NOTE DIVS REPEATS: Play a sequence, dynamically allocated to a channel
JF.VOX CHANNEL DIVS REPEATS: Play a sequence on a specific **CHANNEL**
JF.TICK DIVS: Clock *Geode* with a stream of ticks at **DIVS** per measure
JF.TICK BPM: Set timebase for *Geode* with a static **BPM**
JF.QT DIVS: Quantize *Geode* events to **DIVS** of the timebase

Get values from Just Friends:

Proposed getters, yet to be implemented as of Teletype 3.2

JF.TR CHANNEL: Returns true if the **CHANNEL** is in motion
JF.RMODE: Returns the state of run_mode (ignores the jack)
JF.RUN: Returns the current RUN value (ignores the jack)
JF.SHIFT: Returns the current transpose setting
JF.MODE: Returns 1 if *Synthesis* or *Geode* are active
JF.TICK: Returns the current *Geode* tempo in beats per minute
JF.GOD: Returns 1 if god mode is active
JF.QT: Returns the number of *divisions* quantize is set to
JF.SPEED: Returns the current *shape* (0) or *sound* (1) switch position
JF.TSC: Returns the current *MODE* switch state (1/2/3)
JF.RAMP: Returns the current state of the *RAMP* knob
JF.CURVE: Returns the current state of the *CURVE* knob
JF.FM: Returns the current state of the *FM* knob
JF.TIME: Returns the current state of the *TIME* knob + cv
JF.INTONE: Returns the current state of the *INTONE* knob + cv

Just Friends reference

OP *(set)*	*(aliases)*	Description
`JF.ADDR x`		Sets JF II address (`1` = primary, `2` = secondary). Use with only one JF on the bus! Saves to JF internal memory, so only one-time config is needed.
`JF.SEL x`		Sets target JF unit (`1` = primary, `2` = secondary).
`JF0: ...`		Send following JF OPs to both units starting with selected unit.
`JF1: ...`		Send following JF OPs to unit 1 ignoring the currently selected unit.
`JF2: ...`		Send following JF OPs to unit 2 ignoring the currently selected unit.
`JF.RAMP`		Gets value of RAMP knob.
`JF.CURVE`		Gets value of CURVE knob.
`JF.FM`		Gets value of FM knob.
`JF.INTONE`		Gets value of INTONE knob and CV offset.
`JF.TIME`		Gets value of TIME knob and CV offset.
`JF.SPEED`		Gets value of SPEED switch (`1` = sound, `0` = shape).
`JF.TSC`		Gets value of MODE switch (`0` = transient, `1` = sustain, `2` = cycle).
`JF.TR x y`		Simulate a TRIGGER input. `x` is channel (`0` = all primary JF channels, `1..6` = primary JF, `7..12` = secondary JF, `-1` = all channels both JF) and y is state (`0` or `1`)
`JF.RMODE x`		Set the RUN state of Just Friends when no physical jack is present. (`0` = run off, non-zero = run on)
`JF.RUN x`		Send a 'voltage' to the RUN input. Requires `JF.RMODE 1` to have been executed, or a physical cable in JF's input. Thus Just Friend's RUN modes are accessible without needing a physical cable & control voltage to set the RUN parameter. use `JF.RUN V x` to set to `x` volts. The expected range is V -5 to V 5

OP *(set)*	*(aliases)*	Description
`JF.SHIFT x`		Shifts the transposition of Just Friends, regardless of speed setting. Shifting by V 1 doubles the frequency in sound, or doubles the rate in shape. `x` = pitch, use `M x` for semitones, or `V y` for octaves.
`JF.VTR x y`		Like `JF.TR` with added volume control. Velocity is scaled with volts, so try `V 5` for an output trigger of 5 volts. Channels remember their latest velocity setting and apply it regardless of TRIGGER origin (digital or physical). `x` = channel, `0` sets all channels. `y` = velocity, amplitude of output in volts. eg `JF.VTR 1 V 4`.
`JF.TUNE x y z`		Adjust the tuning ratios used by the INTONE control. `x` = channel, `y` = numerator (set the multiplier for the tuning ratio), `z` = denominator (set the divisor for the tuning ratio). `JF.TUNE 0 0 0` resets to default ratios.
`JF.MODE x`		Set the current choice of standard functionality, or Just Type alternate modes (Speed switch to Sound for Synth, Shape for Geode). You'll likely want to put JF.MODE x in your Teletype INIT scripts. `x` = nonzero activates alternative modes. `0` restores normal.
`JF.VOX x y z`		Synth mode: create a note at the specified channel, of the defined pitch & velocity. All channels can be set simultaneously with a chan value of 0. `x` = channel, `y` = pitch relative to C3, `z` = velocity (like `JF.VTR`). Geode mode: Create a stream of rhythmic envelopes on the named channel. `x` = channel, `y` = division, `z` = number of repeats.

OP *(set)*	*(aliases)*	Description
`]F.NOTE x y`		Synth: polyphonically allocated note sequencing. Works as JF.VOX with chan selected automatically. Free voices will be taken first. If all voices are busy, will steal from the voice which has been active the longest. `x` = pitch relative to C3, `y` = velocity. Geode: works as JF.VOX with dynamic allocation of channel. Assigns the rhythmic stream to the oldest unused channel, or if all are busy, the longest running channel. `x` = division, `y` = number of repeats.
`]F.POLY x y`		As `]F.NOTE` but across dual JF. Switches between primary and secondary units every 6 notes or until reset using `]F.POLY.RESET`.
`]F.POLY.RESET`		Resets `]F.POLY` note count.
`]F.PITCH x y`		Change pitch without retriggering. `x` = channel, `y` = pitch relative to C3.
`]F.GOD x`		Redefines C3 to align with the 'God' note. `x` = `0` sets A to 440, `x` = `1` sets A to 432.
`]F.TICK x`		Sets the underlying timebase of the Geode. `x` = clock. 0 resets the timebase to the start of measure. 1 to 48 shall be sent repetitively. The value representing ticks per measure. 49 to 255 sets beats-per-minute and resets the timebase to start of measure.
`]F.QT x`		When non-zero, all events are queued & delayed until the next quantize event occurs. Using values that don't align with the division of rhythmic streams will cause irregular patterns to unfold. Set to 0 to deactivate quantization. `x` = division, 0 deactivates quantization, 1 to 32 sets the subdivision & activates quantization.

16n

The 16n Faderbank is an open-source controller that can be polled by the Teletype to read the positions of its 16 sliders.

OP *(set)*	*(aliases)*	Description
`FADER x`	`FB`	Reads the value of the `FADER` slider `x`; default return range is from 0 to 16383. Up to four Faderbanks can be addressed; `x` value between 1 and 16 correspond to Faderbank 1, `x` between 17 and 32 to Faderbank 2, etc…
`FADER.SCALE x y z`	`FB.S`	Set static scaling of the `FADER x` to between `min` and `max`.
`FADER.CAL.MIN x`	`FB.C.MIN`	Reads `FADER x` minimum position and assigns a zero value
`FADER.CAL.MAX x`	`FB.C.MAX`	Reads `FADER x` maximum position and assigns the maximum point
`FADER.CAL.RESET x`	`FB.C.R`	Resets the calibration for FADER x

FADER.CAL.MIN

- `FADER.CAL.MIN x`
- *alias*: `FB.C.MIN`

1. Slide FADER x all the way down to the bottom
2. Execute `FADER.CAL.MIN x` from the live terminal
3. Call `FADER x` and confirm the 0 result

FADER.CAL.MAX

- `FADER.CAL.MAX x`
- *alias*: `FB.C.MAX`

1. Slide FADER x all the way up to the top
2. Execute `FADER.CAL.MAX x` from the live terminal
3. Call `FADER x` and verify that the result is 16383

TELEXi

The TELEXi (or TXi) is an input expander that adds 4 IN jacks and 4 PARAM knobs to the Teletype. There are jumpers on the back so you can hook more than one TXi to your Teletype simultaneously.

Inputs added to the system by the TELEX modules are addressed sequentially: 1-4 are on your first module of any type, 5-8 are on the second, 9-12 on the third, and so on. A few of the commands reference the module by its unit number – but those are rare.

OP *(set)*	*(aliases)*	Description
TI.PARAM x	TI.PRM	reads the value of PARAM knob x; default return range is from 0 to 16383; return range can be altered by the TI.PARAM.MAP command
TI.PARAM.QT x	TI.PRM.QT	return the quantized value for PARAM knob x using the scale set by TI.PARAM.SCALE; default return range is from 0 to 16383
TI.PARAM.N x	TI.PRM.N	return the quantized note number for PARAM knob x using the scale set by TI.PARAM.SCALE
TI.PARAM.SCALE x	TI.PRM.SCALE	select scale # y for PARAM knob x; scales listed in full description
TI.PARAM.MAP x y z	TI.PRM.MAP	maps the PARAM values for input x across the range y - z (defaults 0-16383)
TI.IN x		reads the value of IN jack x; default return range is from -16384 to 16383 - representing -10V to +10V; return range can be altered by the TI.IN.MAP command
TI.IN.QT x		return the quantized value for IN jack x using the scale set by TI.IN.SCALE; default return range is from -16384 to 16383 - representing -10V to +10V
TI.IN.N x		return the quantized note number for IN jack x using the scale set by TI.IN.SCALE
TI.IN.SCALE x		select scale # y for IN jack x; scales listed in full description
TI.IN.MAP x y z		maps the IN values for input jack x across the range y - z (default range is -16384 to 16383 - representing -10V to +10V)

OP *(set)*	*(aliases)*	Description
TI.PARAM.INIT x	TI.PRM.INIT	initializes PARAM knob x back to the default boot settings and behaviors; neutralizes mapping (but not calibration)
TI.IN.INIT x		initializes IN jack x back to the default boot settings and behaviors; neutralizes mapping (but not calibration)
TI.INIT d		initializes all of the PARAM and IN inputs for device number d (1-8)
TI.PARAM.CALIB x y	TI.PRM.CALIB	calibrates the scaling for PARAM knob x; y of 0 sets the bottom bound; y of 1 sets the top bound
TI.IN.CALIB x y		calibrates the scaling for IN jack x; y of -1 sets the -10V point; y of 0 sets the 0V point; y of 1 sets the +10V point
TI.STORE d		stores the calibration data for TXi number d (1-8) to its internal flash memory
TI.RESET d		resets the calibration data for TXi number d (1-8) to its factory defaults (no calibration)

TI.PARAM.SCALE

- TI.PARAM.SCALE x
- *alias*: TI.PRM.SCALE

Quantization Scales

0. Equal Temperament [DEFAULT]
1. 12-tone Pythagorean scale
2. Vallotti & Young scale (Vallotti version) also known as Tartini-Vallotti (1754)
3. Andreas Werckmeister's temperament III (the most famous one, 1681)
4. Wendy Carlos' Alpha scale with perfect fifth divided in nine
5. Wendy Carlos' Beta scale with perfect fifth divided by eleven
6. Wendy Carlos' Gamma scale with third divided by eleven or fifth by twenty
7. Carlos Harmonic & Ben Johnston's scale of 'Blues' from Suite f.micr.piano (1977) & David Beardsley's scale of 'Science Friction'
8. Carlos Super Just
9. Kurzweil "Empirical Arabic"
10. Kurzweil "Just with natural b7th", is Sauveur Just with 7/4

11. Kurzweil “Empirical Bali/Java Harmonic Pelog”
12. Kurzweil “Empirical Bali/Java Slendro, Siam 7”
13. Kurzweil “Empirical Tibetian Ceremonial”
14. Harry Partch’s 43-tone pure scale
15. Partch’s Indian Chromatic, Exposition of Monophony, 1933.
16. Partch Greek scales from “Two Studies on Ancient Greek Scales” on black/white

TI.PARAM.MAP

- **TI.PARAM.MAP x y z**
- *alias*: **TI.PRM.MAP**

If you would like to have a **PARAM** knob values over a specific range, you can offload the processing for this to the TXo by mapping the range of the potentiometer using the **MAP** command. It works a lot like the **MAP** operator, but does the heavy lifting on the TXi, saving you space in your code and cycles on your processor.

For instance, let’s have the first knob return a range from 0 to 100.

```
TI.PARAM.MAP 1 0 100
```

You can reset the mapping by either calling the map command with the default range or by using the **INIT** command (**TO.PARAM.INIT 1**).

TI.IN.SCALE

- **TI.IN.SCALE x**

Quantization Scales

0. Equal Temperament [DEFAULT]
1. 12-tone Pythagorean scale
2. Vallotti & Young scale (Vallotti version) also known as Tartini-Vallotti (1754)
3. Andreas Werckmeister’s temperament III (the most famous one, 1681)
4. Wendy Carlos’ Alpha scale with perfect fifth divided in nine
5. Wendy Carlos’ Beta scale with perfect fifth divided by eleven
6. Wendy Carlos’ Gamma scale with third divided by eleven or fifth by twenty
7. Carlos Harmonic & Ben Johnston’s scale of ‘Blues’ from Suite f.micr.piano (1977) & David Beardsley’s scale of ‘Science Friction’
8. Carlos Super Just
9. Kurzweil “Empirical Arabic”

10. Kurzweil “Just with natural b7th”, is Sauveur Just with 7/4
11. Kurzweil “Empirical Bali/Java Harmonic Pelog”
12. Kurzweil “Empirical Bali/Java Slendro, Siam 7”
13. Kurzweil “Empirical Tibetian Ceremonial”
14. Harry Partch’s 43-tone pure scale
15. Partch’s Indian Chromatic, Exposition of Monophony, 1933.
16. Partch Greek scales from “Two Studies on Ancient Greek Scales” on black/white

TI.PARAM.CALIB

- **TI.PARAM.CALIB x y**
- *alias*: **TI.PRM.CALIB**

You can calibrate your **PARAM** knob by using this command. The steps for full calibration are as follows:

1. Turn the PARAM knob **x** all the way to the left
2. Send the command ‘TI.PARAM.CALIBRATE x 0’
3. Turn the PARAM knob **x** all the way to the right
4. Send the command ‘TI.PARAM.CALIBRATE x 1’

Don’t forget to call the **TI.STORE** command to save your calibration between sessions.

TI.IN.CALIB

- **TI.IN.CALIB x y**

You can calibrate your **IN** jack to external voltages by using this command. The steps for full calibration are as follows:

1. Send a **-10V** signal to the input **x**
2. Send the command ‘TI.IN.CALIBRATE x -1’
3. Send a **0V** signal to the input **x**
4. Send the command ‘TI.IN.CALIBRATE x 0’
5. Send a **10V** signal to the input **x**
6. Send the command ‘TI.IN.CALIBRATE x 1’

Don’t forget to call the **TI.STORE** command to save your calibration between sessions.

TELEXo

The TELEXo (or TXo) is an output expander that adds an additional 4 Trigger and 4 CV jacks to the Teletype. There are jumpers on the back so you can hook more than one TXo to your Teletype simultaneously.

Outputs added to the system by the TELEX modules are addressed sequentially: 1-4 are on your first module of any type, 5-8 are on the second, 9-12 on the third, and so on. A few of the commands reference the module by its unit number – but those are rare.

Unlike the Teletype's equivalent operators, the TXo does not have get commands for its functions. This was intentional as these commands eat up processor and bus-space. While they may be added in the future, as of now you cannot poll the TXo for the current state of its various operators.

OP *(set)*	*(aliases)*	Description
`TO.TR x y`		sets the `TR` value for output `x` to `y` (0/1)
`TO.TR.TOG x`		toggles the `TR` value for output `x`
`TO.TR.PULSE x`	`TO.TR.P`	pulses the `TR` value for output `x` for the duration set by `TO.TR.TIME/S/M`
`TO.TR.PULSE.DIV x y`	`TO.TR.P.DIV`	sets the clock division factor for `TR` output `x` to `y`
`TO.TR.PULSE.MUTE x y`	`TO.TR.P.MUTE`	mutes or un-mutes `TR` output `x`; `y` is 1 (mute) or 0 (un-mute)
`TO.TR.TIME x y`		sets the time for `TR.PULSE` on output `n`; `y` in milliseconds
`TO.TR.TIME.S x y`		sets the time for `TR.PULSE` on output `n`; `y` in seconds
`TO.TR.TIME.M x y`		sets the time for `TR.PULSE` on output `n`; `y` in minutes
`TO.TR.WIDTH x y`		sets the time for `TR.PULSE` on output `n` based on the width of its current metronomic value; `y` in percentage (0-100)
`TO.TR.POL x y`		sets the polarity for `TR` output `n`
`TO.TR.M.ACT x y`		sets the active status for the independent metronome for output `x` to `y` (0/`1`); default 0 (disabled)
`TO.TR.M x y`		sets the independent metronome interval for output `x` to `y` in milliseconds; default `1000`

OP *(set)*	*(aliases)*	Description
TO.TR.M.S x y		sets the independent metronome interval for output x to y in seconds; default **1**
TO.TR.M.M x y		sets the independent metronome interval for output x to y in minutes
TO.TR.M.BPM x y		sets the independent metronome interval for output x to y in Beats Per Minute
TO.TR.M.COUNT x y		sets the number of repeats before deactivating for output x to y; default 0 (infinity)
TO.TR.M.MUL x y		multiplies the **M** rate on **TR** output x by y; y defaults to **1** - no multiplication
TO.TR.M.SYNC x		synchronizes the **PULSE** for metronome on **TR** output number x
TO.M.ACT d y		sets the active status for the 4 independent metronomes on device d (1-8) to y (0/**1**); default 0 (disabled)
TO.M d y		sets the 4 independent metronome intervals for device d (1-8) to y in milliseconds; default **1000**
TO.M.S d y		sets the 4 independent metronome intervals for device d to y in seconds; default **1**
TO.M.M d y		sets the 4 independent metronome intervals for device d to y in minutes
TO.M.BPM d y		sets the 4 independent metronome intervals for device d to y in Beats Per Minute
TO.M.COUNT d y		sets the number of repeats before deactivating for the 4 metronomes on device d to y; default 0 (infinity)
TO.M.SYNC d		synchronizes the 4 metronomes for device number d (1-8)
TO.CV x		CV target output x; y values are bipolar (-16384 to +16383) and map to -10 to +10
TO.CV.SLEW x y		set the slew amount for output x; y in milliseconds
TO.CV.SLEW.S x y		set the slew amount for output x; y in seconds
TO.CV.SLEW.M x y		set the slew amount for output x; y in minutes

OP *(set)*	*(aliases)*	Description
`TO.CV.SET x y`		set the CV for output `x` (ignoring `SLEW`); `y` values are bipolar (-16384 to +16383) and map to -10 to +10
`TO.CV.OFF x y`		set the CV offset for output `x`; `y` values are added at the final stage
`TO.CV.QT x y`		CV target output `x`; `y` is quantized to output's current `CV.SCALE`
`TO.CV.QT.SET x y`		set the CV for output `x` (ignoring `SLEW`); `y` is quantized to output's current `CV.SCALE`
`TO.CV.M x y`		target the CV to note `y` for output `x`; `y` is indexed in the output's current `CV.SCALE`
`TO.CV.M.SET x y`		set the CV to note `y` for output `x`; `y` is indexed in the output's current `CV.SCALE` (ignoring `SLEW`)
`TO.CV.SCALE x y`		select scale # `y` for CV output `x`; scales listed in full description
`TO.CV.LOG x y`		translates the output for `CV` output `x` to logarithmic mode `y`; `y` defaults to 0 (off); mode 1 is for 0-16384 (0V-10V), mode 2 is for 0-8192 (0V-5V), mode 3 is for 0-4096 (0V-2.5V), etc.
`TO.CV.CALIB x`		Locks the current offset (`CV.OFF`) as a calibration offset and saves it to persist between power cycles for output `x`.
`TO.CV.RESET x`		Clears the calibration offset for output `x`
`TO.OSC x y`		Targets oscillation for CV output `x` to `y`
`TO.OSC.SET x y`		set oscillation for CV output `x` to `y` (ignores slew)
`TO.OSC.QT x y`		targets oscillation for CV output `x` to `y`
`TO.OSC.QT.SET x y`		set oscillation for CV output `x` to `y`, quantized to the current scale (ignores slew)
`TO.OSC.M x y`		targets oscillation for CV output `x` to note `y`
`TO.OSC.M.SET x y`		sets oscillation for CV output `x` to note `y` (ignores slew)
`TO.OSC.FQ x y`		targets oscillation for CV output `x` to frequency `y` in Hertz
`TO.OSC.FQ.SET x y`		targets oscillation for CV output `x` to frequency `y` in Hertz (ignores slew)

OP *(set)*	*(aliases)*	Description
`TO.OSC.LFO x y`		Targets oscillation for CV output `x` to LFO frequency `y` in millihertz
`TO.OSC.LFO.SET x y`		Targets oscillation for CV output `x` to LFO frequency `y` in millihertz (ignores slew)
`TO.OSC.CYC x y`		targets the oscillator cycle length to `y` for CV output `x` with the portamento rate determined by the `TO.OSC.SLEW` value; `y` is in milliseconds
`TO.OSC.CYC.SET x y`		sets the oscillator cycle length to `y` for CV output `x` (ignores `CV.OSC.SLEW`); `y` is in milliseconds
`TO.OSC.CYC.S x y`		targets the oscillator cycle length to `y` for CV output `x` with the portamento rate determined by the `TO.OSC.SLEW` value; `y` is in seconds
`TO.OSC.CYC.S.SET x y`		sets the oscillator cycle length to `y` for CV output `x` (ignores `CV.OSC.SLEW`); `y` is in seconds
`TO.OSC.CYC.M x y`		targets the oscillator cycle length to `y` for CV output `x` with the portamento rate determined by the `TO.OSC.SLEW` value; `y` is in minutes
`TO.OSC.CYC.M.SET x y`		sets the oscillator cycle length to `y` for CV output `x` (ignores `CV.OSC.SLEW`); `y` is in minutes
`TO.OSC.SCALE x y`		select scale # `y` for CV output `x`; scales listed in full description
`TO.OSC.WAVE x y`		set the waveform for output `x` to `y`; `y` range is 0-4500, blending between 45 waveforms
`TO.OSC.RECT x y`		rectifies the polarity of the oscillator for output `x` to `y`; 0 is no rectification, +/-1 is partial rectification, +/-2 is full rectification
`TO.OSC.WIDTH x y`		sets the width of the pulse wave on output `x` to `y`; `y` is a percentage of total width (0 to 100); only affects waveform 3000
`TO.OSC.SYNC x`		resets the phase of the oscillator on `CV` output `x` (relative to `TO.OSC.PHASE`)
`TO.OSC.PHASE x y`		sets the phase offset of the oscillator on CV output `x` to `y` (0 to 16383); `y` is the range of one cycle

OP *(set)*	*(aliases)*	Description
`TO.OSC.SLEW x y`		sets the frequency slew time (portamento) for the oscillator on CV output `x` to `y`; `y` in milliseconds
`TO.OSC.SLEW.S x y`		sets the frequency slew time (portamento) for the oscillator on CV output `x` to `y`; `y` in seconds
`TO.OSC.SLEW.M x y`		sets the frequency slew time (portamento) for the oscillator on CV output `x` to `y`; `y` in minutes
`TO.OSC.CTR x y`		centers the oscillation on CV output `x` to `y`; `y` values are bipolar (-16384 to +16383) and map to -10 to +10
`TO.ENV.ACT x y`		activates/deactivates the AD envelope generator for the CV output `x`; `y` turns the envelope generator off (0 - default) or on (1); `CV` amplitude is used as the peak for the envelope and needs to be > 0 for the envelope to be perceivable
`TO.ENV x y`		trigger the attack stage of output `x` when `y` changes to 1, or decay stage when it changes to 0
`TO.ENV.TRIG x`		triggers the envelope at `CV` output `x` to cycle; `CV` amplitude is used as the peak for the envelope and needs to be >0 for the envelope to be perceivable
`TO.ENV.ATT x y`		set the envelope attack time to `y` for `CV` output `x`; `y` in milliseconds (default 12 ms)
`TO.ENV.ATT.S x y`		set the envelope attack time to `y` for `CV` output `x`; `y` in seconds
`TO.ENV.ATT.M x y`		set the envelope attack time to `y` for `CV` output `x`; `y` in minutes
`TO.ENV.DEC x y`		set the envelope decay time to `y` for `CV` output `x`; `y` in milliseconds (default 250 ms)
`TO.ENV.DEC.S x y`		set the envelope decay time to `y` for `CV` output `x`; `y` in seconds
`TO.ENV.DEC.M x y`		set the envelope decay time to `y` for `CV` output `x`; `y` in minutes
`TO.ENV.EOR x n`		at the end of rise of `CV` output `x`, fires a `PULSE` to the trigget output `n`

OP *(set)*	*(aliases)*	Description
TO.ENV.EOC x n		at the end of cycle of CV output x, fires a PULSE to the trigget output n
TO.ENV.LOOP x y		causes the envelope on CV output x to loop for y times
TO.TR.INIT x		initializes TR output x back to the default boot settings and behaviors; neutralizes metronomes, dividers, pulse counters, etc.
TO.CV.INIT x		initializes CV output x back to the default boot settings and behaviors; neutralizes offsets, slews, envelopes, oscillation, etc.
TO.INIT d		initializes all of the TR and CV outputs for device number d (1-8)
TO.KILL d		cancels all TR pulses and CV slews for device number d (1-8)

TO.TR.PULSE.DIV

- TO.TR.PULSE.DIV x y
- *alias*: TO.TR.P.DIV

The pulse divider will output one trigger pulse every y pulse commands. For example, setting the DIV factor to 2 like this:

```
TO.TR.P.DIV 1 2
```

Will cause every other TO.TR.P 1 command to emit a pulse.

Reset it to one (TO.TR.P.DIV 1 1) or initialize the output (TO.TR.INIT 1) to return to the default behavior.

TO.TR.WIDTH

- TO.TR.WIDTH x y

The actual time value for the trigger pulse when set by the WIDTH command is relative to the current value for TO.TR.M. Changes to TO.TR.M will change the duration of TR.PULSE when using the WIDTH mode to set its value. Values for y are set in percentage (0-100).

For example:

```
TO.TR.M 1 1000
TO.TR.WIDTH 1 50
```

The length of a **TR.PULSE** is now 500ms.

```
TO.TR.M 1 500
```

The length of a **TR.PULSE** is now 250ms. Note that you don't need to use the width command again as it automatically tracks the value for **TO.TR.M**.

TO.TR.M.ACT

- **TO.TR.M.ACT** x y

Each **TR** output has its own independent metronome that will execute a **TR.PULSE** at a specified interval. The **ACT** command enables (1) or disables (0) the metronome.

TO.TR.M.COUNT

- **TO.TR.M.COUNT** x y

This allows for setting a limit to the number of times **TO.TR.M** will **PULSE** when active before automatically disabling itself. For example, let's set it to pulse 5 times with 500ms between pulses:

```
TO.TR.M 1 500
TO.TR.M.COUNT 1 5
```

Now, each time we activate it, the metronome will pulse 5 times - each a half-second apart.

```
TO.TR.M.ACT 1 1
```

PULSE ... **PULSE** ... **PULSE** ... **PULSE** ... **PULSE**.

The metronome is now disabled after pulsing five times. If you call **ACT** again, it will emit five more pulses.

To reset, either set your **COUNT** to zero (**TO.TR.M.COUNT 1 0**) or call init on the output (**TO.TR.INIT 1 1**).

TO.TR.M.MUL

- **TO.TR.M.MUL** x y

The following example will cause 2 against 3 patterns to pulse out of **TO.TR** outputs **3** and **4**.

```
TO.TR.M.MUL 3 2
TO.TR.M.MUL 4 3
L 3 4: TO.TR.M.ACT | 1
```

TO.M.SYNC

- `TO.M.SYNC d`

This command causes the TXo at device `d` to synchronize all of its independent metronomes to the moment it receives the command. Each will then continue to pulse at its own independent `M` rate.

TO.CV.SCALE

- `TO.CV.SCALE x y`

Quantization Scales

0. Equal Temperament [DEFAULT]
1. 12-tone Pythagorean scale
2. Vallotti & Young scale (Vallotti version) also known as Tartini-Vallotti (1754)
3. Andreas Werckmeister's temperament III (the most famous one, 1681)
4. Wendy Carlos' Alpha scale with perfect fifth divided in nine
5. Wendy Carlos' Beta scale with perfect fifth divided by eleven
6. Wendy Carlos' Gamma scale with third divided by eleven or fifth by twenty
7. Carlos Harmonic & Ben Johnston's scale of 'Blues' from Suite f.micr.piano (1977) & David Beardsley's scale of 'Science Friction'
8. Carlos Super Just
9. Kurzweil "Empirical Arabic"
10. Kurzweil "Just with natural b7th", is Sauveur Just with 7/4
11. Kurzweil "Empirical Bali/Java Harmonic Pelog"
12. Kurzweil "Empirical Bali/Java Slendro, Siam 7"
13. Kurzweil "Empirical Tibetian Ceremonial"
14. Harry Partch's 43-tone pure scale
15. Partch's Indian Chromatic, Exposition of Monophony, 1933.
16. Partch Greek scales from "Two Studies on Ancient Greek Scales" on black/white

TO.CV.LOG

- `TO.CV.LOG x y`

The following example creates an envelope that ramps to 5V over a logarithmic curve:

```
TO.CV.SET 1 V 5
TO.CV.LOG 1 2
```

```
TO.ENV.ATT 1 500
TO.ENV.DEC.S 1 2
TO.ENV.ACT 1 1
```

When triggered (`TO.ENV.TRIG 1`), the envelope will rise to 5V over a half a second and then decay back to zero over two seconds. The curve used is 2' which covers 0V-5V.

If a curve is too small for the range being covered, values above the range will be limited to the range's ceiling. In the above example, voltages above 5V will all return as 5V.

TO.CV.CALIB

- `TO.CV.CALIB x`

To calibrate your TXo outputs, follow these steps. Before you start, let your expander warm up for a few minutes. It won't take long - but you want to make sure that it is calibrated at a more representative temperature.

Then, first adjust your offset (CV.OFF) until the output is at zero volts (0). For example:

```
CV.OFF 1 8
```

Once that output measures at zero volts, you want to lock it in as the calibration by calling the following operator:

```
CV.CALIB 1
```

You will find that the offset is now zero, but the output is at the value that you targeted during your prior adjustment. To reset to normal (and forget this calibration offset), use the `TO.CV.RESET` command.

TO.OSC

- `TO.OSC x y`

Targets oscillation for CV output `x` to `y` with the portamento rate determined by the `TO.OSC.SLEW` value. `y` is 1V/oct translated from the standard range (1-16384). A value of 0 disables oscillation; `CV` amplitude is used as the peak for oscillation and needs to be > 0 for it to be perceivable.

Setting an `OSC` frequency greater than zero for a `CV` output will start that output oscillating. It will swing its voltage between to the current `CV` value and its polar opposite. For example:

```
TO.CV 1 V 5
```

```
T0.OSC 1 M 69
```

This will emit the audio-rate note A (at 440Hz) swinging from '+5V' to '-5V'. The `CV` value acts as an amplitude control. For example:

```
T0.CV.SLEW.M 1 1
T0.CV 1 V 10
```

This will cause the oscillations to gradually increase in amplitude from 5V to 10V over a period of one minute.

IMPORANT: if you do not set a `CV` value, the oscillator will not emit a signal.

If you want to go back to regular `CV` behavior, you need to set the oscillation frequency to zero. E.g. `T0.OSC 1 0`. You can also initialize the `CV` output with `T0.CV.INIT 1`, which resets all of its settings back to start-up default.

T0.OSC.SET

- `T0.OSC.SET x y`

Set oscillation for CV output `x` to `y` (ignores `CV.OSC.SLEW`.) `y` is 1V/oct translated from the standard range (1-16384); a value of 0 disables oscillation. `CV` amplitude is used as the peak for oscillation and needs to be > 0 for it to be perceivable.

T0.OSC.QT

- `T0.OSC.QT x y`

Targets oscillation for CV output `x` to `y` with the portamento rate determined by the `T0.OSC.SLEW` value. `y` is 1V/oct translated from the standard range (1-16384) and quantized to current `OSC.SCALE`. A value of 0 disables oscillation; `CV` amplitude is used as the peak for oscillation and needs to be > 0 for it to be perceivable.

T0.OSC.QT.SET

- `T0.OSC.QT.SET x y`

Set oscillation for CV output `x` to the 1V/oct value `y` (ignores `CV.OSC.SLEW`.) `y` is 1v/oct translated from the standard range (1-16384) and quantized to current `OSC.SCALE`. A value of 0 disables oscillation; `CV` amplitude is used as the peak for oscillation and needs to be >0 for it to be perceivable.

TO.OSC.M

- `TO.OSC.M x y`

Targets oscillation for CV output `x` to note `y` with the portamento rate determined by the `TO.OSC.SLEW` value. See quantization scale reference for `y`; `CV` amplitude is used as the peak for oscillation and needs to be >0 for it to be perceivable.

TO.OSC.M.SET

- `TO.OSC.M.SET x y`

Sets oscillation for CV output `x` to note `y` (ignores `CV.OSC.SLEW`.) See quantization scale reference for `y`; `CV` amplitude is used as the peak for oscillation and needs to be >0 for it to be perceivable.

TO.OSC.FQ

- `TO.OSC.FQ x y`

Targets oscillation for CV output `x` to frequency `y` with the portamento rate determined by the `TO.OSC.SLEW` value. `y` is in Hz; a value of 0 disables oscillation. `CV` amplitude is used as the peak for oscillation and needs to be >0 for it to be perceivable.

TO.OSC.FQ.SET

- `TO.OSC.FQ.SET x y`

Sets oscillation for CV output `x` to frequency `y` (ignores `CV.OSC.SLEW`.) `y` is in Hz; a value of 0 disables oscillation. `CV` amplitude is used as the peak for oscillation and needs to be >0 for it to be perceivable.

TO.OSC.LFO

- `TO.OSC.LFO x y`

Targets oscillation for CV output `x` to LFO frequency `y` with the portamento rate determined by the `TO.OSC.SLEW` value. `y` is in mHz (millihertz: 10^-3 Hz); a value of 0 disables oscillation. `CV` amplitude is used as the peak for oscillation and needs to be >0 for it to be perceivable.

TO.OSC.LFO.SET

- `TO.OSC.LFO.SET x y`

Sets oscillation for CV output `x` to LFO frequency `y` (ignores `CV.OSC.SLEW`.) `y` is in mHz (millihertz: 10^-3 Hz); a value of 0 disables oscillation. `CV` amplitude is used as the peak for oscillation and needs to be >0 for it to be perceivable.

TO.OSC.SCALE

- `TO.OSC.SCALE x y`

Quantization Scales

0. Equal Temperament [DEFAULT]
1. 12-tone Pythagorean scale
2. Vallotti & Young scale (Vallotti version) also known as Tartini-Vallotti (1754)
3. Andreas Werckmeister's temperament III (the most famous one, 1681)
4. Wendy Carlos' Alpha scale with perfect fifth divided in nine
5. Wendy Carlos' Beta scale with perfect fifth divided by eleven
6. Wendy Carlos' Gamma scale with third divided by eleven or fifth by twenty
7. Carlos Harmonic & Ben Johnston's scale of 'Blues' from Suite f.micr.piano (1977) & David Beardsley's scale of 'Science Friction'
8. Carlos Super Just
9. Kurzweil "Empirical Arabic"
10. Kurzweil "Just with natural b7th", is Sauveur Just with 7/4
11. Kurzweil "Empirical Bali/Java Harmonic Pelog"
12. Kurzweil "Empirical Bali/Java Slendro, Siam 7"
13. Kurzweil "Empirical Tibetian Ceremonial"
14. Harry Partch's 43-tone pure scale
15. Partch's Indian Chromatic, Exposition of Monophony, 1933.
16. Partch Greek scales from "Two Studies on Ancient Greek Scales" on black/white

TO.OSC.WAVE

- `TO.OSC.WAVE x y`

There are 45 different waveforms, values translate to sine (0), triangle (100), saw (200), pulse (300) all the way to random/noise (4500). Oscillator shape between values is a blend of the pure waveforms.

TO.OSC.RECT

- `TO.OSC.RECT x y`

The rectification command performs a couple of levels of rectification based on how you have it set. The following values for y work as follows:

- y = 2: “full-positive” - inverts negative values, making them positive
- y = 1: “half-positive” - omits all negative values (values below zero are set to zero)
- y = 0: no rectification (default)
- y = -1: “half-negative” - omits all positive values (values above zero are set to zero)
- y = -2: “full-negative” - inverts positive values, making them negative

TO.OSC.SLEW

- TO.OSC.SLEW x y

This parameter acts as a frequency slew for the targeted CV output. It allows you to gradually slide from one frequency to another, creating a portamento like effect. It is also great for smoothing transitions between different LFO rates on the oscillator. For example:

```
TO.CV 1 V 5
TO.OSC.SLEW 1 30000
TO.OSC.LFO.SET 1 1000
TO.OSC.LFO 1 100
```

This will start an LFO on CV 1 with a rate of 1000mHz. Then, over the next 30 seconds, it will gradually decrease in rate to 100mHz.

TO.OSC.CTR

- TO.OSC.CTR x y

For example, to create a sine wave that is centered at 2.5V and swings up to +5V and down to 0V, you would do this:

```
TO.CV 1 VV 250
TO.OSC.CTR 1 VV 250
TO.OSC.LFO 1 500
```

TO.ENV.ACT

- TO.ENV.ACT x y

This setting activates (1) or deactivates (0) the envelope generator on CV output y. The envelope generator is dependent on the current voltage setting for the output. Upon activation, the targeted output will go to zero. Then, when triggered

(TO.ENV.TRIG), it will ramp the voltage from zero to the currently set peak voltage (TO.CV) over the attack time (TO.ENV.ATT) and then decay back to zero over the decay time (TO.ENV.DEC). For example:

```
TO.CV.SET 1 V 8
TO.ENV.ACT 1 1
TO.ENV.ATT.S 1 1
TO.ENV.DEC.S 1 30
```

This will initialize the CV 1 output to have an envelope that will ramp to +8V over one second and decay back to zero over thirty seconds. To trigger the envelope, you need to send the trigger command TO.ENV.TRIG 1. Envelopes currently re-trigger from the start of the cycle.

To return your CV output to normal function, either deactivate the envelope (TO.ENV.ACT 1 0) or reinitialize the output (TO.CV.INIT 1).

TO.ENV

- TO.ENV x y

This parameter essentially allows output x to act as a gate between the 0 and 1 state. Changing this value from 0 to 1 causes the envelope to trigger the attack phase and hold at the peak CV value; changing this value from 1 to 0 causes the decay stage of the envelope to be triggered.

TO.ENV.EOR

- TO.ENV.EOR x n

Fires a PULSE at the End of Rise to the unit-local trigger output n for the envelope on CV output x; n refers to trigger output 1-4 on the same TXo as CV output x.

The most important thing to know with this operator is that you can only cause the EOR trigger to fire on the same device as the TXo with the envelope. For this command, the outputs are numbered LOCALLY to the unit with the envelope.

For example, if you have an envelope running on your second TXo, you can only send the EOR pulse to the four outputs on that device:

```
TO.ENV.EOR 5 1
```

This will cause the first output on TXo #2 (TO.TR 5) to pulse after the envelope's attack segment.

TO.ENV.EOC

- **TO.ENV.EOC x n**

Fires a **PULSE** at the End of Cycle to the unit-local trigger output **n** for the envelope on **CV** output **x**. **n** refers to trigger output 1-4 on the same TXo as CV output 'y'.

The most important thing to know with this operator is that you can only cause the EOC trigger to fire on the same device as the TXo with the envelope. For this command, the outputs are numbered LOCALLY to the unit with the envelope.

For example, if you have an envelope running on your second TXo, you can only send the EOC pulse to the four outputs on that device:

TO.ENV.EOC 5 1

This will cause the first output on TXo #2 (**TO.TR 5**) to pulse after the envelope's decay segment.

TO.ENV.LOOP

- **TO.ENV.LOOP x y**

Causes the envelope on **CV** output **x** to loop for **y** times. A **y** of 0 will cause the envelope to loop infinitely; setting **y** to 1 (default) disables looping and (if currently looping) will cause it to finish its current cycle and cease.

Crow

OP *(set)*	*(aliases)*	Description
`CROW.SEL x`		Sets target crow unit (1 (default), to 4).
`CROWM: ...`		Send following CROW OPs to all units starting with selected unit.
`CROW1: ...`		Send following CROW OPs to unit 1 ignoring the currently selected unit.
`CROW2: ...`		Send following CROW OPs to unit 2 ignoring the currently selected unit.
`CROW3: ...`		Send following CROW OPs to unit 3 ignoring the currently selected unit.
`CROW4: ...`		Send following CROW OPs to unit 4 ignoring the currently selected unit.
`CROW.V x y`		Sets output `x` to value `y`. Use `V y` for volts.
`CROW.SLEW x y`		Sets output `x` slew rate to `y` milliseconds.
`CROW.C1 x`		Calls the function 'ii.self.call1(x)' on crow.
`CROW.C2 x y`		Calls the function 'ii.self.call2(x, y)' on crow.
`CROW.C3 x y z`		Calls the function 'ii.self.call3(x, y, z)' on crow.
`CROW.C4 x y z t`		Calls the function 'ii.self.call4(x, y, z, t)' on crow.
`CROW.RST`		Calls the function `crow.reset()` returning crow to default state.
`CROW.PULSE x y z t`		Creates a trigger pulse on output `x` with duration `y` (ms) to voltage `z` with polarity `t`.
`CROW.AR x y z t`		Creates an envelope on output `x`, rising in `y` ms, falling in `z` ms, and reaching height `t`.
`CROW.LFO x y z t`		Starts an envelope on output `x` at rate `y` where 0 = 1Hz with 1v/octave scaling. `z` sets amplitude and `t` sets skew for assymetrical triangle waves.
`CROW.IN x`		Gets voltage at input `x`.
`CROW.OUT x`		Gets voltage of output `x`.
`CROW.Q0`		Returns the result of calling the function 'crow.self.query0()'.
`CROW.Q1 x`		Returns the result of calling the function 'crow.self.query1(x)'.

OP *(set)*	*(aliases)*	Description
`CROW.Q2 x y`		Returns the result of calling the function ‘crow.self.query2(x, y)’.
`CROW.Q3 x y z`		Returns the result of calling the function ‘crow.self.query3(x, y, z)’.

W/

More extensively covered in the W/ Documentation[18].

OP *(set)*	*(aliases)*	Description
`WS.PLAY x`		Set playback state and direction. 0 stops playback. 1 sets forward motion, while -1 plays in reverse
`WS.REC x`		Set recording mode. 0 is playback only. 1 sets overdub mode for additive recording. -1 sets overwrite mode to replace the tape with your input
`WS.CUE x`		Go to a cuepoint relative to the playhead position. 0 retriggers the current location. 1 jumps to the next cue forward. -1 jumps to the previous cue in the reverse. These actions are relative to playback direction such that 0 always retriggers the most recently passed location
`WS.LOOP x`		Set the loop state on/off. 0 is off. Any other value turns loop on

[18] https://www.whimsicalraps.com/pages/w-type

W/2.0

More extensively covered in the W/ Documentation[19].

There are separate ops for each supported algorithm: delay, synth, tape. Two units can be connected using a different i2c address (refer to the official documentation for more details). The following section describes ops that control which unit is selected. These ops apply to all algorithms.

OP *(set)*	*(aliases)*	Description
`W/.SEL x`		Sets target W/2.0 unit (1 = primary, 2 = secondary).
`W/1: ...`		Send following W/2.0 OPs to unit 1 ignoring the currently selected unit.
`W/2: ...`		Send following W/2.0 OPs to unit 2 ignoring the currently selected unit.

[19] https://www.whimsicalraps.com/pages/w-type

W/2.0 tape

Tape mode of W/ eurorack module.

More extensively covered in the W/ Documentation[20].

OP *(set)*	*(aliases)*	Description
`W/T.REC x`		Sets recording state (0/1)
`W/T.PLAY playback`		Set the playback state. -1 will flip playback direction
`W/T.REV`		Reverse the direction of playback
`W/T.SPEED n d`		Set speed as a rate, calculated by `n` / `d` (numerator/denominator). Negative values are reverse (s16V)
`W/T.FREQ f`		Set speed as a 'frequency' (s16V) style value `f`. Maintains reverse state
`W/T.ERASE.LVL level`		Strength of erase head when recording. 0 is overdub, 1 is overwrite. Opposite of feedback (s16V)
`W/T.MONITOR.LVL gain`		Level of input passed directly to output (s16V)
`W/T.REC.LVL gain`		Level of input material recorded to tape (s16V)
`W/T.ECHOMODE x`		Set to 1 to playback before erase. 0 (default) erases first
`W/T.LOOP.START`		Set the current time as the beginning of a loop
`W/T.LOOP.END`		Set the current time as the loop end, and jump to start
`W/T.LOOP.ACTIVE x`		Set the state of looping (0/1)
`W/T.LOOP.SCALE x`		Multiply (positive) or divide(negative) loop brace by `x`. Zero resets to original window (s8)
`W/T.LOOP.NEXT dir`		Move loop brace forward/backward by length of loop. Pos = fwd, neg = bkwd. 0 jumps to loop start (eg. retriggers). (s8)

[20] https://www.whimsicalraps.com/pages/w-type

OP *(set)*	*(aliases)*	Description
`W/T.TIME sec sub`		Move playhead to an arbitrary location on tape. `sec` is a whole number of seconds, `sub` is subseconds where 1V = 1 sec. Uses 2 args because TT only supports 16bit args. (s16V)
`W/T.SEEK sec sub`		Move playhead relative to current position. `sec` is a whole number of seconds, `sub` is subseconds where 1V = 1 sec. Uses 2 args because TT only supports 16bit args. (s16V)
`W/T.CLEARTAPE`		WARNING! Erases all recorded audio on the tape!

W/2.0 delay

Delay mode of W/ eurorack module.

More extensively covered in the W/ Documentation[21].

OP *(set)*	*(aliases)*	Description
`W/D.FBK lvl`		amount of feedback from read head to write head (s16V)
`W/D.MIX fade`		fade from dry to delayed signal
`W/D.FILT cutoff`		centre frequency of filter in feedback loop (s16V)
`W/D.FREEZE is_active`		deactivate record head to freeze the current buffer (s8)
`W/D.TIME seconds`		set delay buffer length in `seconds` (s16V), when rate == 1
`W/D.LEN count divisions`		set buffer loop length as a fraction of buffer time (u8)
`W/D.POS count divisions`		set loop start location as a fraction of buffer time (u8)
`W/D.CUT count divisions`		jump to loop location as a fraction of loop length (u8)
`W/D.FREQ.RNG freq_range`		TBD (s8)
`W/D.RATE mul`		direct `mul`tiplier (s16V) of tape speed
`W/D.FREQ volts`		manipulate tape speed with musical values (s16V)
`W/D.CLK`		sends clock pulse for synchronization
`W/D.CLK.RATIO mul div`		set clock pulses per buffer time, with clock `mul`/`div` (s8)
`W/D.PLUCK volume`		pluck the delay line with noise at volume (s16V)
`W/D.MOD.RATE rate`		set the multiplier for the modulation rate (s16V)
`W/D.MOD.AMT amount`		set the `amount` (s16V) of delay line modulation to be applied

[21] https://www.whimsicalraps.com/pages/w-type

W/2.0 synth

Synth mode of W/ eurorack module.

More extensively covered in the W/ Documentation[22].

OP *(set)*	*(aliases)*	Description
`W/S.PITCH voice pitch`		set `voice` (s8) to `pitch` (s16V) in volts-per-octave
`W/S.VEL voice velocity`		strike the vactrol of `voice` (s8) at `velocity` (s16V) in volts
`W/S.VOX voice pitch velocity`		set `voice` (s8) to `pitch` (s16V) and strike the vactrol at `velocity` (s16V)
`W/S.NOTE pitch level`		dynamically assign a voice, set to `pitch` (s16V), strike with `velocity` (s16V)
`W/S.POLY pitch level`		As `W/S.NOTE` but across dual W/. Switches between primary and secondary units every 4 notes or until reset using `W/S.POLY.RESET`.
`W/S.POLY.RESET`		Resets `W/S.POLY` note count.
`W/S.AR.MODE is_ar`		in attack-release mode, all notes are `plucked` and no `release` is required'
`W/S.LPG.TIME time`		vactrol `time` (s16V) constant. -5=drones, 0=vtl5c3, 5=blits
`W/S.LPG.SYM symmetry`		vactrol attack-release ratio. -5=fastest attack, 5=long swells (s16V)
`W/S.CURVE curve`		cross-fade waveforms: -5=square, 0=triangle, 5=sine (s16V)
`W/S.RAMP ramp`		waveform symmetry: -5=rampwave, 0=triangle, 5=sawtooth (NB: affects FM tone)
`W/S.FM.INDEX index`		amount of FM modulation. -5=negative, 0=minimum, 5=maximum (s16V)
`W/S.FM.RATIO n d`		ratio of the FM modulator to carrier as a ratio, `n`umerator / `d`enominator. floating point values up to 20.0 supported (s16V)
`W/S.FM.ENV amount`		amount of vactrol envelope applied to fm index, -5 to +5 (s16V)
`W/S.PATCH jack param`		patch a hardware `jack` (s8) to a `param` (s8) destination

[22] https://www.whimsicalraps.com/pages/w-type

OP *(set)*	*(aliases)*	Description
`W/S.VOICES count`		set number of polyphonic voices to allocate. use 0 for unison mode (s8)

i2c2midi

i2c2midi is a DIY open source 2 HP Teletype Expander that speaks I2C and MIDI. It bridges the gap between monome Teletype and external MIDI-enabled devices, using I2C: It receives I2C messages from Teletype and converts them to MIDI notes, MIDI CC messages and other MIDI messages to control external devices like synths and effects; it receives MIDI messages from external MIDI controllers and stores the values internally, which can be requested at any time by Teletype via I2C. For more information: https://github.com/attowatt/i2c2midi

OP *(set)*	*(aliases)*	Description
`I2M.CH (x)`	`I2M.#`	Get currently set MIDI channel / Set MIDI channel `x` (1..16 for TRS, 17..32 for USB) for MIDI out
`I2M.TIME (x)`	`I2M.T`	Get current note duration / Set note duration of MIDI notes to `x` ms (0..32767) for current channel
`I2M.T# ch (x)`		Get current note duration / Set note duration of MIDI notes to `x` ms (0..32767) for channel `ch` (0..32).
`I2M.SHIFT (x)`	`I2M.S`	Get current transposition / Set transposition of MIDI notes to `x` semitones (-127..127) for current channel
`I2M.S# ch (x)`		Get current transposition / Set transposition of MIDI notes to `x` semitones (-127..127) for channel `ch` (0..32)
`I2M.MIN x y`		Set minimum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for current channel
`I2M.MIN# ch x y`		Set minimum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for channel `ch` (0..32)
`I2M.MAX x y`		Set maximum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for current channel
`I2M.MAX# ch x y`		Set maximum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for channel `ch` (0..32)
`I2M.REP (x)`		Get current repetition / Set repetition of MIDI notes to `x` repetitions (1..127) for current channel

OP *(set)*	*(aliases)*	Description
I2M.REP# ch x		Get current repetition / Set repetition of MIDI notes to **x** repetitions (1..127) for channel **ch** (0..32)
I2M.RAT (x)		Get current ratcheting / Set ratcheting of MIDI notes to **x** ratchets (1..127) for current channel
I2M.RAT# ch x		Get current ratcheting / Set ratcheting of MIDI notes to **x** ratchets (1..127) for channel **ch** (0..32)
I2M.MUTE (x)		Get mute state / Set mute state of current MIDI channel to **x** (0..1)
I2M.MUTE# (x)		Get mute state / Set mute state of MIDI channel **ch** to **x** (0..1)
I2M.SOLO (x)		Get solo state / Set solo state of current MIDI channel to **x** (0..1)
I2M.SOLO# (x)		Get solo state / Set solo state of MIDI channel **ch** to **x** (0..1)
I2M.NOTE x y	**I2M.N**	Send MIDI Note On message for note number **x** (0..127) with velocity **y** (1..127) on current channel
I2M.N# ch x y		Send MIDI Note On message for note number **x** (0..127) with velocity **y** (1..127) on channel **ch** (1..32)
I2M.NOTE.O x	**I2M.NO**	Send a manual MIDI Note Off message for note number **x** (0..127)
I2M.NO# ch x		Send a manual MIDI Note Off message for note number **x** (0..127) on channel **ch** (1..32)
I2M.NT x y z		Send MIDI Note On message for note number **x** (0..127) with velocity **y** (1..127) and note duration **z** ms (0..32767)
I2M.NT# ch x y z		Send MIDI Note On message for note number **x** (0..127) with velocity **y** (1..127) and note duration **z** ms (0..32767) on channel **ch** (1..32)
I2M.CC x y		Send MIDI CC message for controller **x** (0..127) with value **y** (0..127)
I2M.CC# ch x y		Send MIDI CC message for controller **x** (0..127) with value **y** (0..127) on channel **ch** (1..32)

OP *(set)*	*(aliases)*	Description
`I2M.CC.SET x y`		Send MIDI CC message for controller `x` (0..127) with value `y` (0..127), bypassing any slew settings
`I2M.CC.SET# ch x y`		Send MIDI CC message for controller `x` (0..127) with value `y` (0..127) on channel `ch` (1..32), bypassing any slew settings
`I2M.CCV x y`		Send MIDI CC message for controller `x` (0..127) with volt value `y` (0..16383, 0..+10V)
`I2M.CCV# ch x y`		Send MIDI CC message for controller `x` (0..127) with volt value `y` (0..16383, 0..+10V) on channel `ch` (1..32)
`I2M.CC.OFF x (y)`		Get current offset / Set offset of values of controller `x` (0..127) to `y` (-127..127)
`I2M.CC.OFF# ch x (y)`		Get current offset / Set offset of values of controller `x` (0..127) to `y` (-127..127) for channel `ch` (1..32)
`I2M.CC.SLEW x (y)`		Get current slew time for controller `x` / Set slew time for controller `x` (0..127) to `y` ms (0..32767)
`I2M.CC.SLEW# ch x (y)`		Get current slew time for controller `x` / Set slew time for controller `x` (0..127) to `y` ms (0..32767) for channel `ch` (1..32)
`I2M.NRPN x y z`		Send MIDI NRPN message (high-res CC) for parameter MSB `x` and LSB `y` with value `y` (0..16383)
`I2M.NRPN# ch x y z`		Send MIDI NRPN message (high-res CC) for parameter MSB `x` and LSB `y` with value `y` (0..16383) on channel `ch` (1..32)
`I2M.NRPN.OFF x y (z)`		Get current offset / Set offset of values of NRPN messages to `z` (-16384..16383)
`I2M.NRPN.OFF# ch x y (z)`		Get current offset / Set offset of values of NRPN messages to `z` (-16384..16383) for channel `ch` (1..32)
`I2M.NRPN.SLEW x y (z)`		Get current slew time / Set slew time for NRPN messages to `z` ms (0..32767)
`I2M.NRPN.SLEW# ch x y (z)`		Get current slew time / Set slew time for NRPN messages to `z` ms (0..32767) for channel `ch` (1..32)

OP *(set)*	*(aliases)*	Description
I2M.NRPN.SET x y z		Send MIDI NRPN message for parameter MSB x and LSB y with value y (0..16383), bypassing any slew settings
I2M.NRPN.SET# ch x y z		Send MIDI NRPN message for parameter MSB x and LSB y with value y (0..16383) on channel ch (1..32), bypassing any slew settings
I2M.PRG x		Send MIDI Program Change message for program x (0..127)
I2M.PB x		Send MIDI Pitch Bend message with value x (-8192..8191)
I2M.AT x		Send MIDI After Touch message with value x (0..127)
I2M.CLK		Send MIDI Clock message, this still needs improvement …
I2M.START		Send MIDI Clock Start message
I2M.STOP		Send MIDI Clock Stop message
I2M.CONT		Send MIDI Clock Continue message
I2M.CHORD x y z	I2M.C	Play chord x (1..8) with root note y (-127..127) and velocity z (1..127)
I2M.C# ch x y z		Play chord x (1..8) with root note y (-127..127) and velocity z (1..127) on channel ch (1..32)
I2M.C.ADD x y	I2M.C+	Add relative note y (-127..127) to chord x (0..8), use x = 0 to add to all chords
I2M.C.RM x y	I2M.C-	Remove note y (-127..127) from chord x (0..8), use x = 0 to remove from all chords
I2M.C.INS x y z		Add note z (-127..127) to chord x (0..8) at index y (0..7), with z relative to the root note; use x = 0 to insert into all chords
I2M.C.DEL x y		Delete note at index y (0..7) from chord x (0..8), use x = 0 to delete from all chords
I2M.C.SET x y z		Set note at index y (0..7) in chord x (0..8) to note z (-127..127), use x = 0 to set in all chords
I2M.C.B x y		Clear and define chord x (0..8) using reverse binary notation (B...)
I2M.C.CLR x		Clear chord x (0..8), use x = 0 to clear all chords

OP *(set)*	*(aliases)*	Description
I2M.C.L x (y)		Get current length / Set length of chord x (0..8) to y (1..8), use x = 0 to set length of all chords
I2M.C.SC x y		Set scale for chord x (0..8) based on chord y (0..8), use x = 0 to set for all chords, use y = 0 to remove scale
I2M.C.REV x y		Set reversal of notes in chord x (0..8) to y. y = 0 or an even number means not reversed, y = 1 or an uneven number means reversed. Use x = 0 to set for all chords.
I2M.C.ROT x y		Set rotation of notes in chord x (0..8) to y steps (-127..127), use x = 0 to set for all chords
I2M.C.TRP x y		Set transposition of chord x (0..8) to y (-127..127), use x = 0 to set for all chords
I2M.C.DIS x y z		Set distortion of chord x (0..8) to y (-127..127) with anchor point z (0..16), use x = 0 to set for all chords
I2M.C.REF x y z		Set reflection of chord x (0..8) to y iterations (-127..127) with anchor point z (0..16), use x = 0 to set for all chords
I2M.C.INV x y		Set inversion of chord x (0..8) to y (-32..32), use x = 0 to set for all chords
I2M.C.STR x y		Set strumming of chord x (0..8) to x ms (0..32767), use x = 0 to set for all chords
I2M.C.VCUR w x y z	I2M.C.V~	Set velocity curve for chord w (0..8) with curve type x (0..5), start value y% (0..32767) and end value z% (0..32767), use w = 0 to set for all chords, use x = 0 to turn off
I2M.C.TCUR w x y z	I2M.C.T~	Set time curve to strumming for chord w (0..8) with curve type x (0..5), start value y% (0..32767) and end value z% (0..32767), use w = 0 to set for all chords, use x = 0 to turn off
I2M.C.DIR x y		Set play direction for chord x (0..8) to direction y (0..8)
I2M.C.QN x y z		Get the transformed note number of a chord note for chord x (1..8) with root note y (-127..127) at index z (0..7)

OP *(set)*	*(aliases)*	Description
I2M.C.QV x y z		Get the transformed note velocity of a chord note for chord x (1..8) with root velocity y (1..127) at index z (0..7)
I2M.B.R x		Turn recording of notes into the buffer on or off
I2M.B.L x		Set the length of the buffer to x ms (0..32767)
I2M.B.START x		Add an offset of x ms (0..32767) to the start of the buffer
I2M.B.END x		Add a negative offset of x ms (0..32767) to the end of the buffer
I2M.B.DIR x		Set the play direction x (0..2) of the buffer
I2M.B.SPE x		Set the playing speed x (1..32767) of the buffer. x = 100 is equivalent to 'normal speed', x = 50 means double the speed, x = 200 means half the speed, etc.
I2M.B.FB x		Set the feedback length x (0..255) of the buffer
I2M.B.NSHIFT x		Set the note shift of recorded notes to x semitones (-127..127)
I2M.B.VSHIFT x		Set the velocity shift of recorded notes to x (-127..127)
I2M.B.TSHIFT x		Set the note duration shift ('time shift') of recorded notes to x ms (-16384..16383)
I2M.B.NOFF x		Set the note offset of recorded notes to x semitones (-127..127)
I2M.B.VOFF x		Set the velocity offset of recorded notes to x (-127..127)
I2M.B.TOFF x		Set the note duration offset ('time offset') of recorded notes to x ms (-16384..16383)
I2M.B.CLR		Clear the buffer, erasing all recorded notes in the buffer
I2M.B.MODE x		Set the buffer mode to x (0..1). 1) Digital 2) Tape
I2M.Q.CH (x)	I2M.Q.#	Get currently set MIDI channel / Set MIDI channel x (1..16) for MIDI in
I2M.Q.LATCH x		Turn on or off 'latching' for MIDI notes received via MIDI in
I2M.Q.NOTE x	I2M.Q.N	Get x (0..7) last note number (0..127) received via MIDI in

OP *(set)*	*(aliases)*	Description
I2M.Q.VEL x	I2M.Q.V	Get x (0..7) last note velocity (1..127) received via MIDI in
I2M.Q.CC x		Get current value (0..127) of controller x (0..127) received via MIDI in
I2M.Q.LCH		Get the latest channel (1..16) received via MIDI in
I2M.Q.LN		Get the note number (0..127) of the latest Note On received via MIDI in
I2M.Q.LV		Get the velocity (1..127) of the latest Note On received via MIDI in
I2M.Q.LO		Get the note number (0..127) of the latest Note Off received via MIDI in
I2M.Q.LC		Get the latest controller number (0..127) received via MIDI in
I2M.Q.LCC		Get the latest controller value (0..127) received via MIDI in
I2M.PANIC		Send MIDI Note Off messages for all notes on all channels, and reset note duration, shift, repetition, ratcheting, min/max

I2M.CH

- I2M.CH (x)
- *alias*: I2M.#

Get currently set MIDI channel / Set MIDI channel x (1..16 for TRS, 17..32 for USB) for MIDI out. Use MIDI channels 1-16 for TRS output, 17-32 for USB output. Default is x = 1.

I2M.TIME

- I2M.TIME (x)
- *alias*: I2M.T

Get current note duration / Set note duration of MIDI notes to x ms (0..32767) for current channel. Based on note duration, i2c2midi will send a MIDI Note Off message automatically. Set x = 0 to deactivate automatic Note Off messages. Default is x = 100.

I2M.T#

- `I2M.T# ch (x)`

Get current note duration / Set note duration of MIDI notes to `x` ms (0..32767) for channel `ch` (0..32). Use `ch = 0` to set for all channels.

I2M.SHIFT

- `I2M.SHIFT (x)`
- *alias*: `I2M.S`

Get current transposition / Set transposition of MIDI notes to `x` semitones (-127..127) for current channel. Default is `x = 0`.

I2M.S#

- `I2M.S# ch (x)`

Get current transposition / Set transposition of MIDI notes to `x` semitones (-127..127) for channel `ch` (0..32). Use `ch = 0` to set for all channels."

I2M.MIN

- `I2M.MIN x y`

Set minimum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for current channel. Default is `x = 0` and `y = 0`. The following modes are available for notes lower than the minimum: 0) Ignore notes 1) Clamp notes 2) Fold back notes by one octave 3) Fold back notes by multiple octaves.

I2M.MIN#

- `I2M.MIN# ch x y`

Set minimum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for channel `ch` (0..32). Use `ch = 0` to set for all channels.

I2M.MAX

- `I2M.MAX x y`

Set maximum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for current channel. Default is `x = 0` and `y = 0`. The following modes are available

for notes higher than the maximum: 0) Ignore notes 1) Clamp notes 2) Fold back notes by one octave 3) Fold back notes by multiple octaves.

I2M.MAX#

- `I2M.MAX# ch x y`

Set maximum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for channel `ch` (0..32). Use `ch = 0` to set for all channels.

I2M.REP

- `I2M.REP (x)`

Get current repetition / Set repetition of MIDI notes to `x` repetitions (1..127) for current channel. Set `x = 1` for no repetitions. Default is `x = 1`.

I2M.REP#

- `I2M.REP# ch x`

Get current repetition / Set repetition of MIDI notes to `x` repetitions (1..127) for channel `ch` (0..32). Use `ch = 0` to set for all channels.

I2M.RAT

- `I2M.RAT (x)`

Get current ratcheting / Set ratcheting of MIDI notes to `x` ratchets (1..127) for current channel. Set `x = 1` for no ratcheting. Default is `x = 1`.

I2M.RAT#

- `I2M.RAT# ch x`

Get current ratcheting / Set ratcheting of MIDI notes to `x` ratchets (1..127) for channel `ch` (0..32). Use `ch = 0` to set for all channels.

I2M.NOTE

- `I2M.NOTE x y`
- *alias*: `I2M.N`

Send MIDI Note On message for note number `x` (0..127) with velocity `y` (1..127) on current channel. A velocity of `0` will be treated as a MIDI Note Off message.

I2M.NOTE.O

- I2M.NOTE.O x
- *alias*: I2M.NO

Send a manual MIDI Note Off message for note number x (0..127). This can be used either before i2c2midi sends the automatic Note Off message (to stop the note from playing before its originally planned ending), or in combination with I2M.TIME set to 0 (in which case i2c2midi does not send automatic Note Off messages).

I2M.NT

- I2M.NT x y z

Send MIDI Note On message for note number x (0..127) with velocity y (1..127) and note duration z ms (0..32767).

I2M.CC

- I2M.CC x y

Send MIDI CC message for controller x (0..127) with value y (0..127).

I2M.CC.SET

- I2M.CC.SET x y

Send MIDI CC message for controller x (0..127) with value y (0..127), bypassing any slew settings.

I2M.CCV

- I2M.CCV x y

Send MIDI CC message for controller x (0..127) with volt value y (0..16383, 0..+10V).

I2M.CC.OFF

- I2M.CC.OFF x (y)

Get current offset / Set offset of values of controller x (0..127) to y (-127..127). Default is y = 0.

I2M.CC.SLEW

- `I2M.CC.SLEW x (y)`

Get current slew time for controller `x` / Set slew time for controller `x` (0..127) to `y` ms (0..32767). i2c2midi will ramp from the controller's last value to a new value within the given time `x`, sending MIDI CCs at a maximum rate of 30 ms. If the slewing is still ongoing when a new value is set, the slewing uses its current position as the last value. Is 8 CC controller values can be slewed simoultaneously before the oldest currently slewing value is overwritten by the newest. Default is `y = 0`.

I2M.NRPN

- `I2M.NRPN x y z`

Send MIDI NRPN message (high-res CC) for parameter MSB `x` and LSB `y` with value `y` (0..16383).

I2M.NRPN.OFF

- `I2M.NRPN.OFF x y (z)`

Get current offset / Set offset of values of NRPN messages to `z` (-16384..16383). Default is z = 0.

I2M.NRPN.SLEW

- `I2M.NRPN.SLEW x y (z)`

Get current slew time / Set slew time for NRPN messages to `z` ms (0..32767). Default is z = 0.

I2M.NRPN.SET

- `I2M.NRPN.SET x y z`

Send MIDI NRPN message for parameter MSB `x` and LSB `y` with value `y` (0..16383), bypassing any slew settings.

I2M.CHORD

- `I2M.CHORD x y z`
- *alias*: `I2M.C`

Play chord `x` (1..8) with root note `y` (-127..127) and velocity `z` (1..127). A chord consists of up to eight notes defined relative to the root note via `I2M.C.ADD`, `I2M.C.RM`, `I2M.C.INS`, `I2M.C.DEL` or `I2M.C.SET`, which are sent out as MIDI Note On messages in the order they are defined in the chord. If no note has been defined in the chord yet, no note will be played. 8 chords can be defined using their respective index 1..8.

I2M.C.ADD

- `I2M.C.ADD x y`
- *alias*: `I2M.C+`

Add note `y` (-127..127) to chord `x` (0..8), with `y` relative to the root note specified when playing a chord. E.g. add `0`, `4` and `7` to define a major triad. Or go more experimental, e.g. `-2`, `13`, `2`, `13`. Up to eight chords can be defined, with eight notes each. Use `x = 0` to add the note to all chords.

I2M.C.RM

- `I2M.C.RM x y`
- *alias*: `I2M.C-`

Remove note `y` (-127..127) from chord `x` (0..8). If the chord contains note `y` multiple times, the latest instance is removed. If the chord does not contain the note the message is simply ignored. Use `x = 0` to remove the note from all chords.

I2M.C.INS

- `I2M.C.INS x y z`

Add note `z` (-127..127) to chord `x` (0..8) at index `y` (0..7), with `z` relative to the root note. Already defined notes at index `y` and higher are pushed to the right. Use `x = 0` to insert the note to all chords.

I2M.C.DEL

- `I2M.C.DEL x y`

Delete note at index `y` (0..7) from chord `x` (0..8). Notes at index `y + 1` and higher are pushed to the left. If `y` is higher than the length of the chord, the message is ignored. Use `x = 0` to delete the note from all chords.

I2M.C.SET

- `I2M.C.SET x y z`

Set note at index `y` (0..7) in chord `x` (0..8) to note `z` (-127..127), replacing what was defined earlier at this index. If `y` is higher than the length of the chord, the message is ignored. Use `x = 0` to set the note in all chords.

I2M.C.B

- `I2M.C.B x y`

Clear and define chord `x` (0..8) using reverse binary notation (`R...`). Use `1` or `0` in order to include or exclude notes from the chord. E.g. use `x = R10001001` for `0,4,7` (major triad) or `x = R1000000100000001` for `0,7,15`. `y` can be a maximum of 16 digit long. Use `x = 0` to clear and define all chords.

I2M.C.CLR

- `I2M.C.CLR x`

Clear chord `x` (0..8). Use `x = 0` to clear all chords.

I2M.C.L

- `I2M.C.L x (y)`

Get current length / Set length of chord `x` (0..8) to `y` (1..8). The length of a chord changes automatically each time a note is added or removed. Values of `x` higher than number of actual defined notes are ignored. Already defined notes are not affected by setting the chord length, but won't be played if their index is outside of the set chord length. Use `x = 0` to set the length of all chords.

I2M.C.SC

- `I2M.C.SC x y`

Set scale for chord `x` (0..8) based on chord `y` (0..8). Setting a scale for a chord comes in handy when using chord transformations that introduce new notes, like `I2M.C.TRP`, `I2M.C.DIS` or `I2M.C.REF`. Use `y = 0` to remove the scale. Use `x = 0` to set reversal for all chords.

I2M.C.REV

- `I2M.C.REV x y`

Set reverse of notes in chord x (0..8) to y. y = 0 or an even number means not reversed, y = 1 or an uneven number means reversed. E.g. y = 1 for chord 0,3,7 will lead to 7,3,0. Default is y = 0. Use x = 0 to reverse all chords.

I2M.C.ROT

- I2M.C.ROT x y

Set rotation of notes in chord x (0..8) to y steps (-127..127). E.g. y = 1 of chord 0,3,7 will lead to 3,7,0, y = 2 will lead to 7,0,3, y = -1 will lead to 7,0,3. Default is y = 0. Use x = 0' to set rotation for all chords.

I2M.C.TRP

- I2M.C.TRP x y

Set transposition of chord x (0..8) to y (-127..127). Transposition adds y to the note number of each note in the chord.Default is y = 0. Use x = 0 to set transposition for all chords. This transformation introduces new notes to the chord – try it in combination with setting a scale.

I2M.C.DIS

- I2M.C.DIS x y z

Set distortion of chord x (0..8) to width y (-127..127) with anchor point z (0..16). Distortion adds $y*n$ to the note number of each note in the chord. The anchor point influences the direction and amount (n) of the transformation. Default is y = 0. Use x = 0 to set distortion for all chords. This transformation introduces new notes to the chord – try it in combination with setting a scale.

I2M.C.REF

- I2M.C.REF x y z

Set reflection of chord x (0..8) to y (-127..127) with anchor point z (0..16). The anchor point defines at which axis the chord gets reflected. Default is y = 0. Use x = 0 to set distortion for all chords. This transformation introduces new notes to the chord – try it in combination with setting a scale.

I2M.C.INV

- I2M.C.INV x y

Set inversion of chord `x` (0..8) to `y` (-32..32). Default is `y = 0`. Use `x = 0` to set inversion for all chords.

I2M.C.STR

- `I2M.C.STR x y`

Set strumming of chord `x` (0..8) to `x` ms (0..32767). Strumming plays the notes of a chord arpeggiated, with an interval of `y` ms in between notes. Default is `y = 0`. Use `x = 0` to set strumming for all chords.

I2M.C.VCUR

- `I2M.C.VCUR w x y z`
- *alias*: `I2M.C.V~`

Set velocity curve for chord `w` (0..8) with curve type `x` (0..5), start value `y`% (0..32767) and end value `z`% (0..32767). This will affect the velocity of the notes in the order they are defined in the chord. Start and end percentages refer to the velocity with which the chord is played via `I2M.C`. Use `x = 0` to turn velocity curve off. The following curves are available: 0) Off 1) Linear 2) Exponential 3) Triangle 4) Square 5) Random. Use `w = 0` to set velocity curve for all chords. Try a random curve with subtle values for a humanizing effect.

I2M.C.TCUR

- `I2M.C.TCUR w x y z`
- *alias*: `I2M.C.T~`

Set time curve for chord `w` (0..8) with curve type `x` (0..5), start value `y`% (0..32767) and end value `z`% (0..32767). This will affect the time interval between the notes in the order they are defined in the chord. Start and end percentages refer to the current strumming setting of the chord, set via `I2M.C.STR`. Use `x = 0` to turn time curve off. The following curves are available: 0) Off 1) Linear 2) Exponential 3) Triangle 4) Square 5) Random. Use `w = 0` to set time curve for all chords. Try a square curve with similar values to create swing. Try a random curve with subtle values for a humanizing effect.

I2M.C.DIR

- `I2M.C.DIR x y`

Set play direction for chord `x` (0..8) to direction `y` (0..8). This will affect the order in which chord notes are played. Make sure to set strumming via `I2M.C.STR`.

The following directions are available: 0) Forward (0,1,2,3,4) 1) Backward (4,3,2,1,0) 2) Inside out (2,1,3,0,4) 3) Outside in (0,4,1,3,2) 4) Random (2,3,1,0,4) 5) Bottom repeat (0,1,0,2,0,3,0,4) 6) Top repeat (0,4,1,4,2,4,3,4) 7) Pingpong (0,1,2,3,4,3,2,1,0) 8) Ping & pong (0,1,2,3,4,4,3,2,1,0). Default is `y = 0`.

I2M.C.QN

- `I2M.C.QN x y z`

Get the transformed note number of a chord note for chord `x` (1..8) with root note `y` (-127..127) at index `z` (0..7). The response is the absolute note number (0..127). Use this OP to send the transformed note number to other devices within eurorack, e.g. via V/OCT to any oscillator or via I2C to I2C-enabled devices like Just Friends or disting EX.

I2M.C.QV

- `I2M.C.QV x y z`

Get the transformed note velocity of a chord note for chord `x` (1..8) with root velocity `y` (1..127) at index `z` (0..7). The response is the absolute note velocity (0..127). Use this OP to send the transformed note velocity to other devices within eurorack, e.g. via CV to a VCA or via I2C to I2C-enabled devices like Just Friends or disting EX.

I2M.B.R

- `I2M.B.R x`

Turn recording of notes into the buffer on or off. `x = 1` is on, `x = 0` is off. If recording is turned on, all outgoing MIDI notes are recorded into the buffer, storing note number, note velocity, note duration and MIDI channel.

I2M.B.L

- `I2M.B.L x`

Set the length of the buffer to `x` ms (0..32767). Default is `x = 1000`.

I2M.B.START

- `I2M.B.START x`

Add an offset of **x** ms (0..32767) to the start of the buffer. The offset time is non-distructively added to the start of the looping buffer. E.g. if the buffer length is set to **1000** ms and start offset is set to **200**

the buffer). Set `x = 0` to turn off the automatic decrease in velocity, keeping notes in the buffer indefinitely. Use this setting in combination with **I2M.B.VSHIFT** or **I2M.B.CLR**. Default is `x = 0`.

I2M.B.NSHIFT

- `I2M.B.NSHIFT x`

Set the note shift of recorded notes to `x` semitones (-127..127). With each buffer iteration, this value gets added accumulatively to the original note number. E.g. with a note shift setting of `x = 12`, a recorded note `60` will be played as note `72` during the first buffer iteration, as note `84` during the second iteration, etc. Default is `x = 0`.

I2M.B.VSHIFT

- `I2M.B.VSHIFT x`

Set the velocity shift of recorded notes to `x` (-127..127). With each buffer iteration, this value gets added accumulatively to the original note velocity. E.g. with a velocity shift setting of `x = -10`, a recorded note with velocity `110` will be played with velocity `100` during the first buffer iteration, with velocity `90` during the second iteration, etc. Default is `x = 0`. Please note: This setting is the twin sibling of **I2M.B.FB**: While **I2M.B.FB** defines the number of iterations determining the amount of change in velocity per iteration, **I2M.B.VSHIFT** defines the amount of change in velocity per iteration determining the number of iterations.

I2M.B.TSHIFT

- `I2M.B.TSHIFT x`

Set the note duration shift ('time shift') of recorded notes to `x` ms (-16384..16383). With each buffer iteration, this value gets added accumulatively to the original note duration. E.g. with a duration shift setting of `x = 100`, a recorded note with duration `200` will be played with duration `300` during the first buffer iteration, with duration `400` during the second iteration, etc. Default is `x = 0`.

I2M.B.NOFF

- `I2M.B.NOFF x`

Set the note offset of recorded notes to `x` semitones (-127..127). This value gets added once to the original note number and is then kept for all buffer iterations.

E.g. with a note offset setting of `x = 7`, a recorded note `60` will be played as note `67` for all buffer iterations. Default is `x = 0`.

I2M.B.VOFF

- `I2M.B.VOFF x`

Set the velocity offset of recorded notes to `x` (-127..127). This value gets added once to the original note velocity and is then kept for all buffer iterations. E.g. with a velocity offset setting of `x = -50`, a recorded note with velocity `120` will be played with velocity `70` for all buffer iterations. Default is `x = 0`.

I2M.B.TOFF

- `I2M.B.TOFF x`

Set the note duration offset ('time offset') of recorded notes to `x` ms (-16384..16383). This value gets added once to the original note duration and is then kept for all buffer iterations. E.g. with a duration offset setting of `x = -50`, a recorded note with duration `200` will be played with duration `150` for all buffer iterations. Default is `x = 0`.

I2M.B.MODE

- `I2M.B.MODE x`

Set the buffer mode to `x` (0..1). The buffer can work in two different modes: 1) Digital 2) Tape. In Digital mode, the buffer speed (set via `I2M.B.SPE`) works independent of note number and note duration: If the buffer speed changes, the note number and note duration of a recorded note stays unaffected. In Tape mode on the other hand, the buffer speed affects the note number and note duration of recorded notes in the buffer, mimicking the behaviour of real tape. If the buffer speed gets doubled, the note number is pitched up by one octave and the note duration gets divided in half. Similarly, if the buffer speed gets divided in half, the note number is pitched down an octave and the note duration gets doubled, etc. Default is `x = 0`.

I2M.Q.CH

- `I2M.Q.CH (x)`
- *alias*: `I2M.Q.#`

Get currently set MIDI channel / Set MIDI channel `x` (1..16) for MIDI in. Default is `x = 1`.

I2M.Q.LATCH

- `I2M.Q.LATCH x`

Turn on or off ‘latching’ for MIDI notes received via MIDI in. `x = 0` means Note Off messages are recorded in the note history, so only notes with keys currently held down on the MIDI controller are stored. `x = 1` means Note Off messages are not recorded in the note history, so notes are still stored after releasing the respective key on the MIDI controller. Default is `x = 1`.

Advanced

Teletype terminology

Here is a picture to help understand the naming of the various parts of a Teletype command:

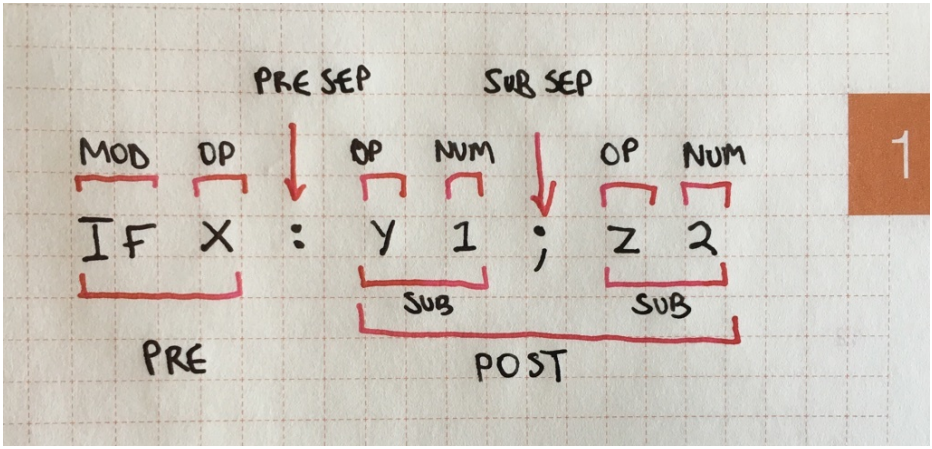


Figure 2: Teletype command terminology

COMMAND The entire command, e.g. **IF X: Y 1; Z 2;**.

PRE The (optional) part before the **PRE SEP**, e.g. **IF X**.

POST The part after the *PRE SEP*, e.g. **Y 1; Z 2**.

SUB A sub command (only allowed in the **POST**), e.g. **Y 1**, or **Z 2**.

PRE SEP A *colon*, only one is allowed.

SUB SEP A *semi-colon*, that separates sub commands (if used), only allowed in the **POST**.

NUM A number between **-32768** and **32767**.

OP An *operator*, e.g. **X**, **TR.PULSE**

MOD A *modifier*, e.g. **IF**, or **L**.

Sub commands

Sub commands allow you to run multiple commands on a single line by utilising a semi-colon to separate each command, for example the following script:

```
X 0
Y 1
Z 2
```

Can be rewritten using sub commands as:

```
X 0; Y 1; Z 2
```

On their own sub commands allow for an increased command density on the Teletype. However when combined with **PRE** statements, certain operations become a lot easier.

Firstly, sub commands cannot be used before a **MOD** or in the **PRE** itself. For example, the following is **not allowed**:

```
X 1; IF X: TR.PULSE 1
```

We can use them in the **POST** though, particularly with an **IF**, for example:

```
IF X: CV 1 M 60; TR.P 1
IF Y: TR.P 1; TR.P 2; TR.P 3
```

Sub commands can also be used with **L**.

Aliases

In general, aliases are a simple concept to understand. Certain `OP`s have been given shorted names to save space and the amount of typing, for example:

```
TR.PULSE 1
```

Can be replaced with:

```
TR.P 1
```

Where confusion may arise is with the symbolic aliases that have been given to some of the maths `OP`s. For instance `+` is given as an alias for `ADD` and it *must* be used as a direct replacement:

```
X ADD 1 1
X + 1 1
```

The key to understanding this is that the Teletype uses *prefix notation*[23] always, even when using mathematical symbols.

The following example (using *infix notation*) **will not work**:

```
X 1 + 1
```

Aliases are entirely optional, most `OP`s do not have aliases. Consult the `OP` tables and documentation to find them.

[23] Also know as *Polish notation*.

Avoiding non-determinism

Although happy accidents in the modular world are one of it's many joys, when writing computer programs they can be incredibly frustrating. Here are some small tips to help keep things predictable (when you want them to be):

1. **Don't use variables unless you need to.**

   This is not to say that variables are not useful, rather it's the opposite and they are extremely powerful. But it can be hard to keep a track of what each variable is used for and on which script it is used. Rather, try to save using variables for when you do want non-deterministic (i.e. *variable*) behaviour.

2. **Consider using `I` as a temporary variable.**

   If you do find yourself needing a variable, particularly one that is used to continue a calculation on another line, consider using the variable `I`. Unlike the other variables, `I` is overwritten whenever `L` is used, and as such, is implicitly transient in nature. One should never need to worry about modifying the value of `I` and causing another script to malfunction, as no script should ever assume the value of `I`.

3. **Use `PM` versions of `OP`s.**

   Most `P OP`s are now available as `PM` versions that ignore the value of `P.I`. (e.g. `PM.START` for `P.START`). Unless you explicitly require the non-determinism of `P` versions, stick to the `PM` versions (space allowing).

4. **Avoid using `A`, `B`, `C` and `D` to refer to the trigger outputs, instead use the numerical values directly.**

   As `A-D` are variables, they may no longer contain the values `1-4`, and while this was the recommend way to name triggers, it is no longer consider ideal. Newer versions of the Teletype hardware have replaced the labels on the trigger outputs, with the numbers `1` to `4`.

Grid integration

Grid integration can be described very simply: it allows you to use grid with teletype. However, there is more to it than just that. You can create custom grid interfaces that can be tailored individually for each scene. Since it's done with scripts you can dynamically change these interfaces at any point - you could even create a dynamic interface that reacts to the scene itself or incoming triggers or control voltages.

You can simply use grid as an LED display to visualize your scene. Or make it into an earthsea style keyboard. You can create sequencers, or control surfaces to control other sequencers. The grid operators simplify building very complex interfaces, while something simple like a bank of faders can be done with just two lines of scripts.

Grid integration consists of 3 main features: grid operators, Grid Visualizer, and Grid Control mode. Grid operators allow you to draw on grid or create grid controls, such as buttons and faders, that can trigger scripts when pressed. As with any other operators you can execute them in Live screen or use them in any of your scripts.

Grid Visualizer provides a virtual grid within the Teletype itself:

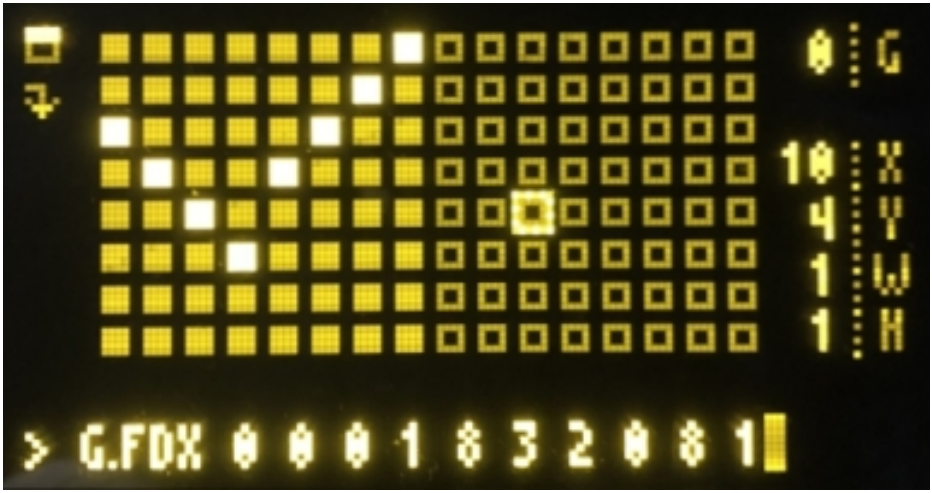


Figure 3: Grid Visualizer

It can be very useful while developing a script as you don't have to switch between the grid and the keyboard as often. To turn it on navigate to Live screen and press **ALT-G** (press again to switch to Full View / turn it off). You can also emulate button presses, which means it can even be used as an alternative to grid if you don't have one, especially in full mode - try it with one of the many grid scenes already developed. For more information on how to use it see the 'Grid Visualizer' section.

Grid Control Mode is a built in grid interface that allows you to use grid to trigger and mute scripts, edit variables and tracker values, save and load scenes, and more. It's available in addition to whatever grid interface you develop - simply press the front panel button while the grid is attached. It can serve as a simple way to use grid to control any scene even without using grid ops, but it can also be very helpful when used together with a scripted grid interface. For more information and diagrams please refer to the 'Grid control mode' section.

If you want to try and build your own grid interfaces see the 'Grid Studies' section.

There is a repository of teletype scenes on github[24].

[24] https://github.com/scanner-darkly/teletype/wiki/CODE-EXCHANGE

Grid operators

Usage

Controls are referenced by ids. Buttons can use ids from `0` to `255`, faders can use ids from `0` to `63`. Controls can overlap, and presses are processed by all the controls that include that grid button. `G.LED` and `G.REC` will be applied on top of everything else.

Defining a control enables it. Disabling a control will hide it. Disabling a group disables all controls within that group. `G.LED` and `G.REC` cannot be disabled, instead they have to be cleared.

`x` and `y` specify the top left corner, `x` is the horizontal coordinate between `0` (left) and `15` (right), `y` is the vertical coordinate between `0` (top) and `15` (bottom). `w` and `h` are width and height. `level` is the brightness level between `0` (completely dark) and `15` (the brightest). There are two special brightness levels: `-1` (dim) and `-2` (brighten). Level -3 clears (makes it transparent again).

A control can only belong to one group at a time. Operators that don't take groups as a parameter will use the currently selected group which is set with `G.GRP`. There are 64 groups available (ids: `0` to `63`).

For script parameter possible values can be `1-10`, where `9` is Metro and `10` is Init. Same script can be shared by multiple controls, in which case take advantage

Groups

there are also button and fader specific group ops - see BUTTONS/FADERS sections below

`G.GRP` `G.GRP id`
get or set the current group

`G.GRP.EN id` `G.GRP.EN id 1/0`
check if a group is enabled or enable/disable a group

`G.GRP.RST id`
reset all controls within a group to defaults

`G.GRP.SW id`
switch a group (enable specified group, disable all others)

`G.GRP.SC id` `G.GRP.SC id script`
get assigned script or assign a script to a group

`G.GRPI`
get id of the last group that received input

LEDS / Rectangles

to dim set level to -1
to brighten set level to -2
to clear set level to -3

`G.LED x y` `G.LED x y level`
get LED level or set LED to level

`G.LED.C x y`
clear LED (same as using `G.LED` with level `-3`)

`G.REC x y w h fill border`
draw a rectangle (use width or height of `1` for lines)

`G.RCT x1 y1 x2 y2 fill border`
draw a rectangle using start/end coordinates

Buttons

`G.BTN id x y w h latch level script`
initialize a button in the current group and assign a script (`0` for no script)

set `latch` to `0` for momentary, any other value for latching

`G.GBT group id x y w h latch level script`
same as above but with group specified

`G.BTX id x y w h latch level script columns rows`
create a block of buttons with the specified number of columns and rows
ids are incremented sequentially

`G.GBX group id x y w h latch level script columns rows`
same as above but with group specified

`G.BTN.EN id` `G.BTN.EN id 1/0`
check if a button is enabled or enable/disable a button

`G.BTN.X id` `G.BTN.X id x`
get or set x coordinate

`G.BTN.Y id` `G.BTN.Y id y`
get or set y coordinate

`G.BTN.V id` `G.BTN.V id value`
get or set value. `1` means the button is pressed, `0` not pressed

`G.BTN.L id` `G.BTN.L id level`
get or set brightness level

`G.BTNI`
get id of the last pressed

`G.BTNX` `G.BTNX x`
get or set x coordinate of the last pressed

`G.BTNY` `G.BTNY y`
get or set y coordinate of the last pressed

`G.BTNV` `G.BTNV value`
get or set value of the last pressed

`G.BTNL` `G.BTNL level`
get or set brightness level of the last pressed

`G.BTN.SW id`
set value for specified button to `1`, set it to `0` for all others within the same group

`G.BTN.PR id action`
emulate button press. set `action` to 1 for press, 0 for release
(`action` is ignored for latching buttons)

`G.GBTN.V group value`
set value for all buttons in a group

`G.GBTN.L group odd_level even_level`
set brightness level for all buttons in a group

`G.GBTN.C group`
get the count of all currently pressed buttons in a group

`G.GBTN.I group index`
get the id of a currently pressed button by index (index is 0-based)

`G.GBTN.W group`
get the width of a block represented by currently pressed buttons in a group

`G.GBTN.H group`
get the height of a block represented by currently pressed buttons in a group

`G.GBTN.X1 group`
get x coordinate for the leftmost pressed button in a group

`G.GBTN.X2 group`
get x coordinate for the rightmost pressed button in a group

`G.GBTN.Y1 group`
get y coordinate for the highest pressed button in a group

`G.GBTN.Y2 group`
get y coordinate for the lowest pressed button in a group

Faders

`G.FDR id x y w h type level script`
initialize fader in the current group and assign a script (`0` for no script)
`type` selects fader type:
0 - coarse, horizontal bar
1 - coarse, vertical bar
2 - coarse, horizontal dot
3 - coarse, vertical dot
4 - fine, horizontal bar
5 - fine, vertical bar
6 - fine, horizontal dot
7 - fine, vertical dot
`level` is brightness level for coarse faders, max value level for fine faders

`G.GFD group id x y w h type level script`
same as above but with group specified

`G.FDX id x y w h type level script columns rows`
create a block of faders with the specified number of columns and rows

ids are incremented sequentially

G.GFX group id x y w h type level script columns rows
same as above but with group specified

G.FDR.EN id G.FDR.EN id 1/0
check if a fader is enabled or enable/disable a fader

G.FDR.X id G.FDR.X id x
get/set x coordinate

G.FDR.Y id G.FDR.Y id y
get/set y coordinate

G.FDR.V id G.FDR.V id value
get/set value scaled to fader min max (set range with **G.GFDR.RM**)

G.FDR.N id G.FDR.N id value
get/set value in grid units

G.FDR.L id G.FDR.L id level
get or set level (brightness level for coarse faders, max value level for fine faders)

G.FDRI
get id of the last pressed

G.FDRX G.FDRX x
get or set x coordinate of the last pressed

G.FDRY G.FDRY y
get or set y coordinate of the last pressed

G.FDRV G.FDRV value
get or set value of the last pressed scaled to fader min max

G.FDRN G.FDRN value
get or set value of the last pressed

G.FDRL G.FDRL level
get or set level of the last pressed

G.FDR.PR id value
emulate fader press

G.GFDR.V group value
set value for all faders in group

G.GFDR.N group value
set value for all faders in group

G.GFDR.L group odd_level even_level
set level for all faders in group

G.GFDR.RN group min max
set range for fader values (by default 0..16383)
applies to all faders within that group

X/Y Pad (work in progress)

G.XYP
G.GXYP
G.XYPX

G.XYP.EN
G.XYP.LAT
G.XYP.L

G.XYP.X id index
G.XYP.Y id index

G.XYPI
G.XYPN
G.XYPX
G.XYPY
G.XYPL

Game of Life (work in progress)

G.GOL
G.GGOL
G.GOLX

G.GOL.EN
G.GOL.L
G.GOL.P
G.GOL.DENS

G.GOL.CLK
G.GOL.RND
G.GOL.NEW
G.GOL.RULES
G.GOL.SHF

Grid studies

Basic visualizations

Let's start with some basic drawing. Open the Metro script and enter the following command:

```
G.REC 3 3 4 4 0 RRAND 1 15
```

Make sure Metro is enabled (execute `M.ACT 1`), unplug the keyboard and connect the grid. You should see a 4x4 rectangle with a border that changes brightness on each metro tick.

Let's reconnect the keyboard (at this point you are realizing you'll be doing this a lot - not to worry, the grid visualization page which we'll cover in a bit will help to minimize the amount of switching). Edit the line and change it to the following:

```
G.REC 1 1 4 4 0 RRAND 1 15
```

If you plug in the grid again you should see another 4x4 rectangle partially covering the previous one. Even though we edited the first command the first rectangle is still there. That's because using `G.REC` is like painting on grid canvas - whatever was previously painted will remain until something is painted over, or until we clear the canvas.

Let's examine all the parameters of `G.REC`:

```
G.REC x y w h fill border
```

`x` and `y` are the coordinates for the rectangle's top left corner. `x` is the horizontal coordinate, starting with the leftmost column, and `y` is the vertical coordinate, starting with the top row. coordinates are 0-based, so the top left corner is 0 0. `w` and `h` are width and height. `fill` and `border` specify the brightness levels. As you can tell by the range we used for `RRAND` the possible range for brightness starts with 1 (the dimmest) and extends to 15 (the brightest). 0 will turn the corresponding LED completely off.

How can we erase the first rectangle now? We could paint a couple of rectangles with the brightness level of 0 over the parts that are still visible, but sometimes it's just easier to clear everything and draw what you need again. To clear the canvas use `G.CLR` op. Please note that what you draw will not be saved with the scene - whenever a scene is loaded it starts with a blank canvas.

You can also draw individual LEDs:

```
G.LED x y level
```

There is no separate operator for lines - just use a rectangle with width or height of 1 (fill level is not used in this case).

Just by using these 3 operators you can now visualize your scene with the grid. For instance, you could make an LED blink whenever a script is executed, or draw rectangles with the size reflecting the value of a variable (`SCALE` op will be useful here to make sure it's scaled to the appropriate range).

Or you could do something more interesting - how about drawing rectangles with random coordinates and random size where each script uses a different brightness level?

Buttons

Visualizing things is great, but we want to use grid to control things too. This is easy to do by using operators for grid controls. Grid controls are typical user interface elements such as buttons and faders.

The simplest way to understand how a control works is to try it. But first let's clear the Metro script and clear everything we painted in the previous study by executing `G.CLR`. Open the live edit page and execute the following:

```
G.BTN 1 0 0 2 2 1 5 0
```

That's a lot of parameters! Let's plug in the grid and see what this op does. You should see a dimly lit 2x2 square in the top left corner. Press on any buttons within that square - it will light up and stay lit until you press it again. You now have a button!

But the most interesting parameter is **script** - this is the script that will get called whenever the button is pressed. This is how we can make scripts react to grid presses.

There is also **id** parameter - this is the button identifier. You have 256 buttons in total, numbered from 0 to 255. You will need this number if you want to change the button's parameters or if you need to get the button's current state. Let's take a look at the script again:

```
TR 1 G.BTN.V 1
```

We are setting the trigger output 1 to the current value of button 1 (we could use id 0 but in this case it's easier to remember if the id of the button matches the id of the trigger output). For buttons value is 1 when they're pressed and 0 when they are not pressed.

What if we want to add more buttons so that we can control all 4 trigger outputs? We could create 3 more buttons and assign them to scripts 2-4, which is not very efficient. Instead, let's assign them all to the same script. First, let's add the buttons:

```
G.BTN 2 2 0 2 2 1 5 1
G.BTN 3 4 0 2 2 1 5 1
G.BTN 4 6 0 2 2 1 5 1
```

Now change script 1 to this:

```
TR G.BTNI G.BTNV
```

If you try it with the grid now you should have 4 buttons controlling 4 outputs with one script. This is possible due to special shortcut ops: **G.BTNI** gives you the id of the last pressed button, and **G.BTNV** gives you its value (you could also use **G.BTN.V G.BTNI** for the latter but **G.BTNV** is shorter and easier to remember).

This is great, but we still had to use 4 commands to create 4 buttons. Since this is something we'd want to put in the Init script we want this to be as short as possible as well. This is where **G.BTX** op will be handy - it creates multiple buttons with one op. So we can replace this:

```
G.BTN 1 0 0 2 2 1 5 1
G.BTN 2 2 0 2 2 1 5 1
G.BTN 3 4 0 2 2 1 5 1
G.BTN 4 6 0 2 2 1 5 1
```

with this:

```
G.BTX 1 0 0 2 2 1 5 1 4 1
```

The first 8 parameters are the same as used by **G.BTN**: **id x y w h latch level script**. There are 2 extra parameters which specify how many columns

and rows we need. Here we're telling teletype to create a 4x1 block of buttons. Buttons created this way are placed next to each other and share the same width, height, background level, latch option and script assignment (the ids will be incremented sequentially). We will take advantage of this op when we build a [trigger sequencer].

As a final exercise for this study, try changing button type to momentary.

Starting simple

There are many different ways to use a grid with a teletype. Let's try some simple examples to show the range of things possible.

But first let's review the conventions behind grid ops that should make it easier to memorize them. As you've noticed all grid ops start with `G.`. This is followed by 3 letters signifying a control: `G.BTN` for buttons, `G.FDR` for faders etc.

To define a control you use the main ops `G.BTN`, `G.FDR`
To define multiple controls replace the last letter with X: `G.BTX`, `G.FDX`

When defining controls the first 4 parameters are always the same: `id, coordinates, width, height`. For buttons this is followed by `latching/level/script`, for faders - `direction/level/script`. When creating multiple controls the last 2 parameters are the number of columns and the number of rows.

Controls are created in the current group (set with `G.GRP`). To specify a different group use group versions of the 4 above ops - `G.GBT`, `G.GFD`, `G.GBX`, `G.GFX` and add the desired group as the first parameter.

All controls share some common properties, referenced by adding a `.` and:
`EN`: `G.BTN.EN`, `G.FDR.EN` - **en**ables or disables a control
`V`: `G.BTN.V`, `G.FDR.V` - **v**alue, simply 1/0 for buttons, range value for faders
`L`: `G.BTN.L`, `G.FDR.L` - background brightness **l**evel
`X`: `G.BTN.X`, `G.FDR.X` - **x** coordinate
`Y`: `G.BTN.Y`, `G.FDR.Y` - **y** coordinate

Group ops `G.GBTN.#` and `G.GFDR.#` allow you to get/set properties for a group of controls. To get/set properties for individual controls you normally specify the control id as the first parameter: `G.FDR.V 5` will return the value of fader 5. Quite often the actual id is not important, you just need to get or set a property on the last control pressed. As these are likely the ops to be used most often they are offered as shortcuts without a `.`: `G.BTNV` returns the state of the last button pressed, `G.FDRL` will set the background level of the last fader pressed etc etc.

All ops can be roughly grouped as follows: general use, groups, drawing, buttons, faders and area ops. There are some specialized ops as well that will be handy

for some specific use cases (and make sure to take advantage of being able to rotate and dim grid!). And now let's see them in action…

Trigger visualizer

It can be useful to have some visual feedback about what's going on with your scene. The most obvious candidate is tracking incoming triggers. We could use G.REC but it's actually easier to use buttons for this since we won't have to calculate the coordinates. We'll use 4x4 buttons, 2 rows with 4 buttons each:

```
I:
G.BTX 1 0 0 4 4 0 0 0 4 2
```

As a reminder, the first 4 parameters are id, coordinates, width, height. These are buttons, so next is latching/level/script: 0 for momentary, 0 for level, 0 for script (no script assigned), 4 buttons in 2 rows.

Now add 2 lines to each of the trigger scripts 1-8 as follows:

```
1:
G.BTN.L 1 15
DEL 50: G.BTN.L 1 0

2:
G.BTN.L 2 15
DEL 50: G.BTN.L 2 0
```

Each trigger script will set the corresponding button's brightness level to 15. We display it for 50 ms and then set it to 0. Now whenever a trigger is received the corresponding button will blink.

Since we already have buttons defined let's make them trigger scripts as well. Since all our scripts are used for triggers we'll use the Metro script to process button presses. Change I to:

```
I:
G.BTX 1 0 0 4 4 0 0 9 4 2
M.ACT 0
```

This will stop M from autotriggering and assign it to button presses (9 is for Metro script). Make sure to execute the Init script by pressing F10. Add this to the Metro:

```
M:
SCRIPT G.BTNI
```

Now when a button is pressed the corresponding script will execute. This scene can be further expanded by having each button's brightness level correspond to a value of some variable.

download the scene:

https://raw.githubusercontent.com/scanner-darkly/teletype_lib/main/grid/trigger_visua

Simple sequencer

A really simple 16 step sequencer:

```
I: G.FDX 0 0 0 1 8 1 0 0 16 1
```

We create 1 row of 16 vertical faders, each 8 LEDs high.

```
1: A WRAP + A 1 0 15
CV 1 N G.FDR.N A; TR.P 1
G.CLR; G.REC A 0 1 8 -3 -2
```

Script 1 increments the step (we store it in variable `A`), updates CV 1 with a note based on the value of the fader located on that step and sends a trigger to TR 1. The last line highlights the current step. Instead of using the fader value as a note number you could store notes in a pattern bank and then do something like `CV 1 PN 0 G.FDR.N A`. Or change `CV 1 N` to `CV 1 V` and use it to modulate something.

Trigger 1 serves as the clock input. If you want teletype to be the main clock update the Metro script to:

```
M SCALE 0 16383 500 50 PARAM
SCRIPT 1
```

This makes the knob control the clock rate and then it calls script 1 to advance the sequencer. We can use another trigger input for reset:

```
2: A 15
```

download the scene:

https://raw.githubusercontent.com/scanner-darkly/teletype_lib/main/grid/simple_sequ

Lofi oscilloscope

This is a very low res oscilloscope but can be fun as a way to visualize your patch.

```
M: A WRAP + A 1 0 15
B SCALE 0 V 5 0 7 IN
G.REC A 0 1 8 0 0
G.LED A B 8
M! SCALE 0 16383 200 10 PRM
```

`A` is the current step again, `B` is the input voltage scaled to 0..7. It will scale the incoming voltage from a 0..5V range to 8 available vertical pixels. Change `V 5` to desired range. Just remember the input is unipolar, so to see full cycle of an LFO you will need to offset it.

We clear the whole column first and then draw a dot at `A,B` coordinates. The last line is your time resolution - it'll adjust the Metro rate based on the knob.

download the scene:

https://raw.githubusercontent.com/scanner-darkly/teletype_lib/main/grid/lofi_oscillosc

video demo: https://www.instagram.com/p/BZ9dcUKAAw4

Next we'll try something a bit more complex…

Trigger sequencer

It might seem it would be difficult to create anything more complex than just a few buttons or faders controlling a few things. Let's try something significantly more complex to both prove that it's possible and show that it can be done with just a few lines. Let's build a 6 track 16 step trigger sequencer!

We'll start with the foundation. We'll use the top 6 rows as 6 tracks with 16 steps each. Each step is a separate button. We already know the op to create multiple buttons at once - `G.BTX`. Open the Init script and add the following:

I:
```
G.BTX 0 0 0 1 1 1 4 0 16 6
```

As a reminder, the parameters are `id x y w h latch level script columns rows`. We are creating a 16x6 block of 1x1 latching buttons starting with the top left corner (coordinates 0,0), the background level of 4 and no script assigned to them. Execute the Init script (by pressing `F10`) and give it a try with the grid - you should see the top 6 rows dimly lit, and you can press anywhere to turn that step on. The buttons don't do anything yet - we didn't even assign a script to them. That's because for sequencer functionality we don't actually need to trigger any actions when steps are changed, we just need to know their state (if they're "on" or "off") for the current sequencer step. Let's store the current step in variable `Z` and use the Metro script to update the step and update the outputs accordingly:

M:
```
Z WRAP + Z 1 0 15
G.GBTN.L 0 4 4
L 0 5: Y I; SCRIPT 8
```

We increment `Z` and use `WRAP` so that it goes back to step 0 after step 15. We will cover `G.GBTN.L` later when we talk about groups, for now let's just say it will

change the background level on all the buttons to 4. The last line doesn't do anything yet but it will when we make script 8 update the output for a row indicated by variable `Y`, and then our loop will do it for each of the 6 rows:

8:
```
D + Z * 16 Y
G.BTN.L D 8
TR + Y 1 G.BTN.V D
```

Here we update a trigger output for row `Y` (row 0 corresponds to trigger 1 etc, so we add 1) based on the corresponding button state. To do so we need to find the button id for a given row (`Y`) and a given step (`Z`). As you recall we used `G.BTX` to create 6 rows of 16 buttons. We used id 0 for the first button. `G.BTX` increments ids sequentially, row by row. So in the first row we have buttons 0..15, in the 2nd row we have buttons 16..31 and so on. To find our button we need to add the current step (`Z`) and 16 for each row after the first one. That's exactly what the formula in the first line does.

The next line contains a new op - `G.BTN.L`, which sets the background level to 8 for the specified button, so that we can see which step we are on at the moment. We don't have to change the level of the previous step back to 4 - we already did it in the Metro script (by setting it to 4 for all the buttons).

The last line updates the corresponding output based on the button state. This will work as a gate sequencer as the output will stay on for the whole step duration, if you prefer triggers replace the last line with `IF G.BTN.V D: TR.P + Y 1`.

Try it now - at this point it's already a fully working 6 track sequencer done in only 7 lines! We could easily make it into an 8 track sequencer by changing the loops above to `L 0 7`, but we're saving the last 2 rows for extra functionality. Let's use row 6 to jump to any step.

But first we need to find ids that haven't been used yet. Since we used 0-95 for the sequencer itself the next available id is 96. Open the Init script and add the following:

```
G.BTX 96 0 6 1 1 0 4 1 16 1
```

This creates a row of momentary buttons assigned to script 1, which we edit to contain the following:

1:
```
Z - G.BTN! 97
```

This sets the current step (`Z`) to the position determined by the id of the last pressed button. Since we used buttons 96-111 for this we need to subtract 96 to get the actual step number, but we subtract 97 because we want to be on

the previous step before the next clock. If you want the jump to be immediate change it to 96 and add the following (the last 2 lines will update the outputs - they are the same as in the Metro script):

```
M.RESET
G.GBTN.L 0 4 4
L 0 5: Y I; SCRIPT 7
```

You can try it now but to save on replugging let's just add the rest, starting with track mutes. We'll use 6 left buttons in row 8 for that. We already used buttons 1-111 so the next available id is 112. Add one more line to the Init script:

```
G.BTX 112 0 7 1 1 1 4 0 6 1
```

and change the last line in script 8 to:

```
A G.BTN.P + 112 Y
TR + Y 1 * A G.BTN.P D
```

`A` will have the state of the mute button for track `Y`, so if the button is pressed the output will get updated, otherwise it'll be muted (if you're doing the trigger version replace the last line with `IF * A G.BTN.P D: TR.P + Y 1`). Remember, by default newly defined buttons will be off, don't forget to press these buttons, otherwise your tracks will be muted! By the way, button state is saved with the scene (both to flash and USB), so it will remember which tracks were muted. If you prefer to always start with unmuted tracks just add `L 112 117: G.BTN.P I 1` to the Init script.

Finally, let's add a start/stop button. We'll use the rightmost button in row 8 for that. The next available id is 118 but let's use 127 instead. You are free to choose whichever ids work best but it helps having a consistent convention - in our case we've been using sequential ids where ids increment horizontally left to right, row by row:

```
0..15
16..31
...
112..127
```

We're not using ids 118-255 yet but if we want to use them later having this consistent identification will help us remember where each button is located and what each button does.

Initialize the start/stop button in the Init script by adding the following (we will enable it by default):

```
G.BTN 127 15 7 1 1 1 4 2
G.BTN.P 127 1
```

It's assigned to script 2 which simply updates `M.ACT` based on the button state:

```
2:
M.ACT G.BTM!
```

Don't forget to execute the Init script before you reconnect the grid. One last thing - this is driven by Metro clock, we want to be able to change speed without reconnecting the grid. Let's use the knob for that - add this to the Metro:

```
M SCALE 0 16383 500 50 PARAM
```

That's it - a 6 track 16 step trigger/gate sequencer with a start/stop button, jump to step and individual track mutes, initialized in 4 lines and functionality taking another 9 lines. That leaves enough room to add more features - as an exercise, try using a trigger input as a reset, add a button to switch direction or clock it externally.

But what about saving the pattern - if you have a good pattern going you probably don't want to lose it when you turn off your modular! Conveniently, you don't need to add any commands to do this - button and fader states are saved with a scene when you save it to flash or a USB stick, and are loaded automatically when you load a scene. Just remember to save your scene.

You do need to add scripts to store your sequences if you have a more complex scene where the same group of buttons is used for multiple pages. Bit operations are very useful for this as they will allow you to store a state of 16 buttons in just one pattern value by using individual bits (all values, variables and pattern values in Teletype have 16 bits). To see a detailed example on how to do it see the next section, 'Saving grid state'.

Another improvement that could be made is having a different background level for different sections. This is what groups allow us to do, see the subsequent section 'Groups'.

see the trigger sequencer in action: https://www.instagram.com/p/BXCbE1sgS-D_

grid sequencer scene:

https://raw.githubusercontent.com/scanner-darkly/teletype_lib/main/grid/grid_sequenc

trigger sequencer scene:

https://raw.githubusercontent.com/scanner-darkly/teletype_lib/main/grid/trigger_sequ

Save grid state

Store the states of multiple rows of 16 buttons in pages via binary.

With a row of 16 buttons, it is possible to store each row in one pattern slot as a 16-bit binary number. 6 rows of 16 steps * 8 pages (768 values) may be stored in just 48 Teletype pattern slots. Page 1 stores to pattern slots 0-5 store page 1, page 2 to 6-11, etc.

Create buttons

```
#1
G.BTX 0 0 0 1 1 1 3 8 16 6
G.FDR 0 0 7 8 1 2 3 8
```

1 - The 6 rows of 16 toggle buttons. Create 96 buttons, starting at the id **0**, top left (x **0**, y **0**), size of **1** by **1**, latching, **3** brightness when off, triggering script **8**, **16** columns, **6** rows.

2 - The page switcher. Create a fader, id **0**, in the bottom left of the grid (x **0**, y **7**), **8** buttons wide, **1** button tall, **2** is the fader type (coarse, horizontal dot), **3** brightness when off, triggering script **8**

Manage presses

```
#8
] + G.BTNY A; K G.BTNX
IF G.BTNV: P ] BSET P ] K
ELSE: P ] BCLR P ] K
A * G.FDRM 6
L 0 95: $ 7
```

1 - Set the var **]** to the row of the button pressed + the pattern number stored in **A**. Set **K** to the x position of the button pressed.

2 - If the button pressed is being turned on, set the bit at the x position to **1** for the pattern slot **]** (visible row + the page slot)

3 - If the button pressed is being turned off, clear the bit to 0.

4 - Both the buttons and the fader trigger this same script. Set A to the pressed fader value * 6. This is used to offset the patter slot for rows - page 1 to patterns 0-5, page 2 to patterns 6-11.

5 - Loop 96 times (for each button) and run script 7.

Update buttons

```
#7
K + A / I 16
G.BTN.V I BGET P K WRP I 0 15
```

1 - Each time this is run it is being iterated for each of the 96 buttons. Set **K** to the page-pattern offset + current button's row number. **/ I 16** gets the row number because Teletype rounds decimal places values down so **I** with a value less than 16 becomes 0, 16-31 becomes 1, 32-47 becomes 2.

2 - Set the current button's (**G.BTN.P I**) value to the bit (**WRP I 0 15**) from the number from the pattern for the button's row, offset by the current page (**P K**).

Groups

One of the most common tasks when programming a grid interface is updating a group of controls at once. In some cases this could be done with a loop, but loops are not always efficient or even possible. This is where groups can be very helpful. Every grid control belongs to one of the 8 groups. Whenever you initiate a control it will be assigned to the current group. We haven't used groups so far, but that simply means that all our controls where assigned to group 0 - this is the default group for each new scene. To change the current group we use **G.GRP** op:

```
G.BTN 0 1 1 1 1 1 0 0
G.GRP 1
G.BTN 1 2 1 1 1 1 0 0
```

In this example button 0 will be assigned to group 0, and button 1 will be assigned to group 1.

To demonstrate how groups can be useful let's create a group of radio buttons. Whenever one button is pressed whichever button was "on" before should be switch to "off" so that only one button is "on" at any given time. We could program it like this:

```
I:
G.BTX 0 0 0 1 1 1 4 1 8 1

1:
L 0 7: G.BTN.P I 0
G.BTN.P G.BTNI 1
```

We defined 8 buttons and assigned them to script 1. The script will set the value for each button to 0 (so that it's "off") and then set it to 1 for the last pressed button. Not too bad, but instead we could simply do:

```
1:
G.BTN.SW G.BTNI
```

G.BTN.SW (**SW** is short for 'switch') is a convenient op that sets the value to 0 for all the buttons in a group except the specified button (in our case, the last

pressed button) which gets its value set to 1. This works well for our radio buttons but we don't want it affecting any other buttons. To avoid that we can simply put the radio buttons into their own group.

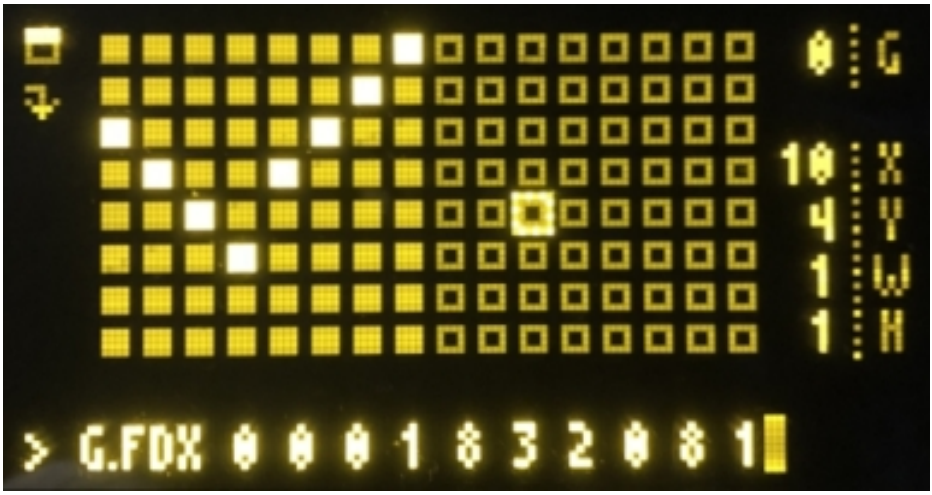
Let's look at another example. In the [trigger sequencer] study we highlighted the current step by changing the background level for the corresponding buttons to a higher one. We also needed to change the previous step back to a lower level. The easiest way to do this is by resetting the background level for all the buttons first, then setting it to a higher level for the current step. `G.GBTN.L` op lets us set the background level for all the buttons in a group. But what if we want to use a different level for the mute/start/stop buttons? Then we can simply place them in a different group.

You can also assign scripts to groups, and they will get executed whenever any control within that group is pressed. Another useful feature is `G.GRP.RST` which resets all the controls within a group - this is a good way to experiment with some new elements without affecting anything else. Just remember to move them into a proper group if needed. There is no separate op to reassign controls to a different group - simply select the group you need and execute the control ops again.

But perhaps the most useful purpose for groups is being able to create paged UIs.

Grid visualizer

As you may have discovered by now, when developing a grid scene you have to often switch between the grid and the keyboard, especially if something is not working as expected. This is where Grid Visualizer will be helpful. It allows you to not only see a visual grid representation but also emulate grid presses, so you might not even have to plug the grid until you complete your scene. As a matter of fact, Grid Visualizer allows you to use grid scenes without the actual grid!



Grid Visualizer is located on Live screen. To enable it, navigate to Live view and press **Alt-G**. You should now see a visual grid representation along with some additional information. Pressing **Alt-G** again will switch to the full grid mode. Press it again to get back to the default Live screen. You can still enter and execute commands while Grid Visualizer is on - this is a great way to test grid ops, as you can immediately see what it will look like. A good way to build up a grid interface is trying your ops in Live screen and then using history (**arrow up**) to get back to the command and copy it to an appropriate script. Don't forget you can execute **G.RST** to reset grid back to the initial clean state. If you're setting up your init script you can use **G.RST** and then execute the init script with **F10** to make sure it contains everything you need.

The highlighted square shows the current grid "cursor" position. If you press **Alt-Space** it will emulate pressing the corresponding grid button. You can move the cursor with **Alt-arrows**. If you additionally hold **Shift** you can expand the cursor to a bigger area. The 4 numbers on the right show you the current X/Y position and width/height of the currently selected area. **G** shows the currently selected group.

If you select an area bigger than 1x1 and use **Alt-Space** it will emulate a press

and hold - it will “press” the starting area point, hold it, then press the ending point, then release both. This is useful for fader slides and for scenes that use “press and hold” for defining loops and such. Being able to define a bigger area has one more purpose - if you press **Alt-PrtSc** it will insert the current x/y/width/height into the command line - very useful for defining faders and buttons.

You might notice you are able to move cursor outside of the visible area (Y will be 8 or higher). This is so that Grid Visualizer can be used with grids 256. As you can only see one half of grid 256 you will need to switch between the upper and the lower halves. This is done with **Alt-/** shortcut.

By default Grid Visualizer shows LEDs exactly as they would be on a varibright grid. It can still be difficult to remember where your grid controls are exactly (especially while you’re still making changes to your grid UI). Grid control preview helps with that by showing the outlines of all defined grid controls. Toggle it with **Alt-**. Grid control preview can be really helpful even after you finish developing a scene - it helps you identify at a glance what controls you have and where.

When you are in full Grid Visualizer mode you don’t need to use **Alt** for any of the shortcuts (although it will still work if you do) - this is a great way to use grid scenes without a grid.

Keyboard shortcuts

Alt-G switch between visualizer off/on/full view
Alt-Arrows move the grid cursor
Alt-Shift-Arrows select an area
Alt-Space emulate a button press
Alt-/ switch between upper and lower half (for grid 256)
Alt- toggle the control view (shows controls outlines)
Alt-PrtSc insert the current x/y/width/height (useful for creating controls)

Alt is not needed when in full grid visualizer mode.

Grid control mode

Grid ops give you the ability to build your own grid UI. But sometimes you just want to use grid to perform basic tasks, such as triggering scripts, editing pattern values etc. You could create a scene for that but instead you can just use the built in Grid Control Mode. Essentially, it turns grid into a Teletype controller.

It's not an alternative to grid scenes - as it happens, it can be especially useful when used together with a grid scene. By delegating some tasks to Grid Control Mode you can pack more functionality into your scene, and you can switch between different Teletype views and save and load scenes without having to reconnect the keyboard.

To turn it on simply press the front panel button (next to the USB port) while grid is connected (remember you shouldn't connect grid directly to teletype - make sure to power it externally!). The right side of grid will change to display the Grid Control UI, assuming you use grid 128. Grid control will also work on grids 64 but it will take over the whole grid (and on grids 256 it will use the right bottom quadrant). To exit Grid Control Mode press the front panel button again.

Grid Control Mode has pages that correspond to Teletype views - Live with variables, Live with Grid Visualizer, Presets, Tracker and 10 scripts. When you turn

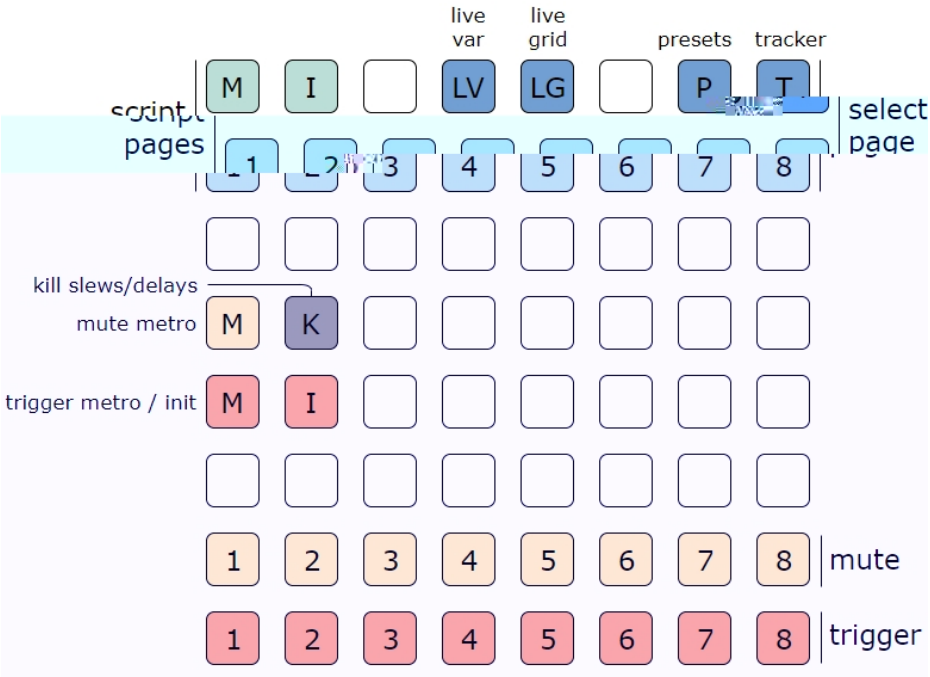


Figure 4: live/script screens shared layput

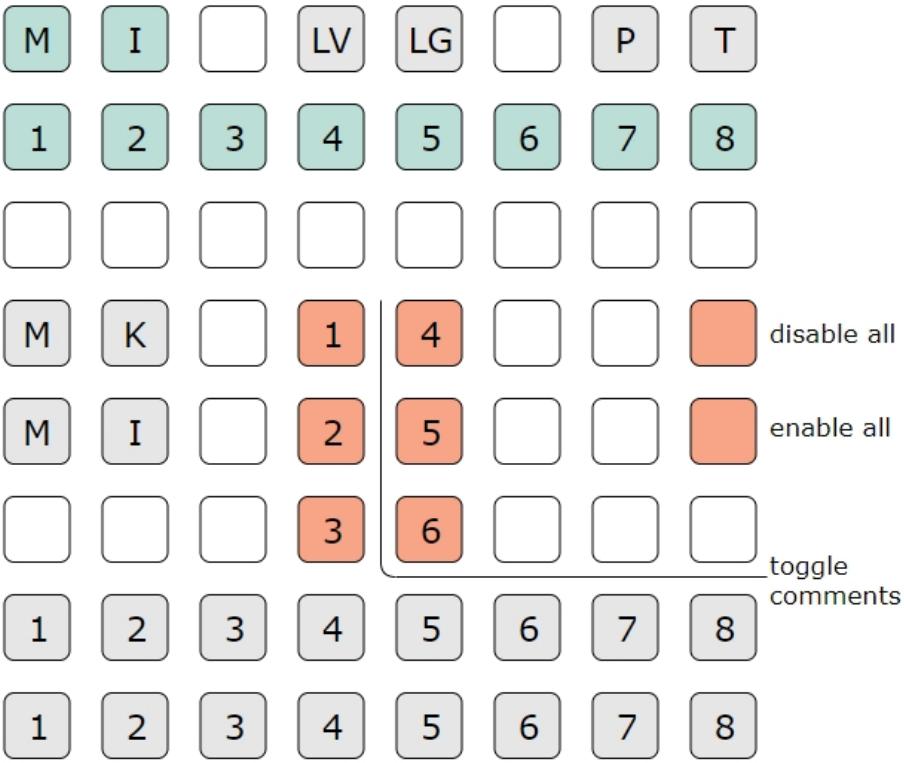


Figure 5: script pages

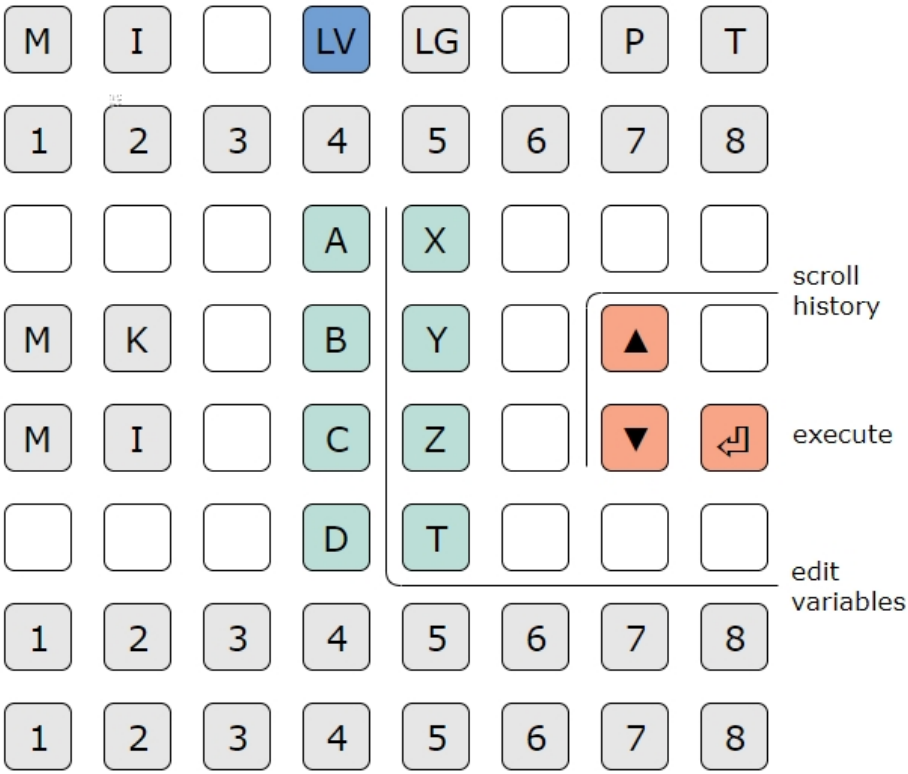


Figure 6: live page with variables

you could use a variable for sequencer position - editing this variable will give you the ability to jump to different positions). To edit a variable press on the corresponding button and hold it. While holding, if you press the button directly to the left it will decrement the value by 1. Pressing the button on the right will increment by 1. You can also increment/decrement by 10 by pressing buttons that are two to the left / two to the right.

The value is also shown on the bottom row (if it's within 0..16 range, or 0..8 on grid 64). Pressing in this row will set the value to that position. If you press and release a variable button without changing its value it will toggle the value between 0 and whatever it was before.

This Live page also allows you to scroll through the live command history and execute commands. This is a good way to have some extra commands available, just remember to enter them while you still have the keyboard plugged in.

Live Grid page

Live page with Grid Visualizer has grid specific controls. You can select the upper/lower grid page, change grid rotation and toggle the grid control view.

Presets page

Presets page keeps the top 2 rows but uses the rest of space to display the 32 available presets. Press on one of the preset buttons to select a preset - this will not load it but simply select it for loading/saving. You can scroll through a preset description with the buttons on the right. The remaining two buttons will load or save the currently selected preset. When you save it will display WRITE on the Teletype screen - press it again to confirm save (or press elsewhere if you want to exit without saving). Once you load or save a preset it will go back to whatever page you were on before switching to Presets.

Tracker page

Finally, the Tracker page utilizes the full 8x8 block. To exit, press the Tracker page button again, and it will go back to the previous page.

The Tracker page controls 8 rows of 4 pattern banks, mirroring what you see on Teletype. You can select a pattern page with the left column, or you can scroll with the 2 buttons in the right lower corner.

Editing tracker values is similar to editing variables - pressing and releasing will toggle the value (useful for trigger sequencers), pressing, holding and then pressing buttons to the left / to the right will increment/decrement by 1 or 10. If you press and hold and then press in a different row it will select pattern start and

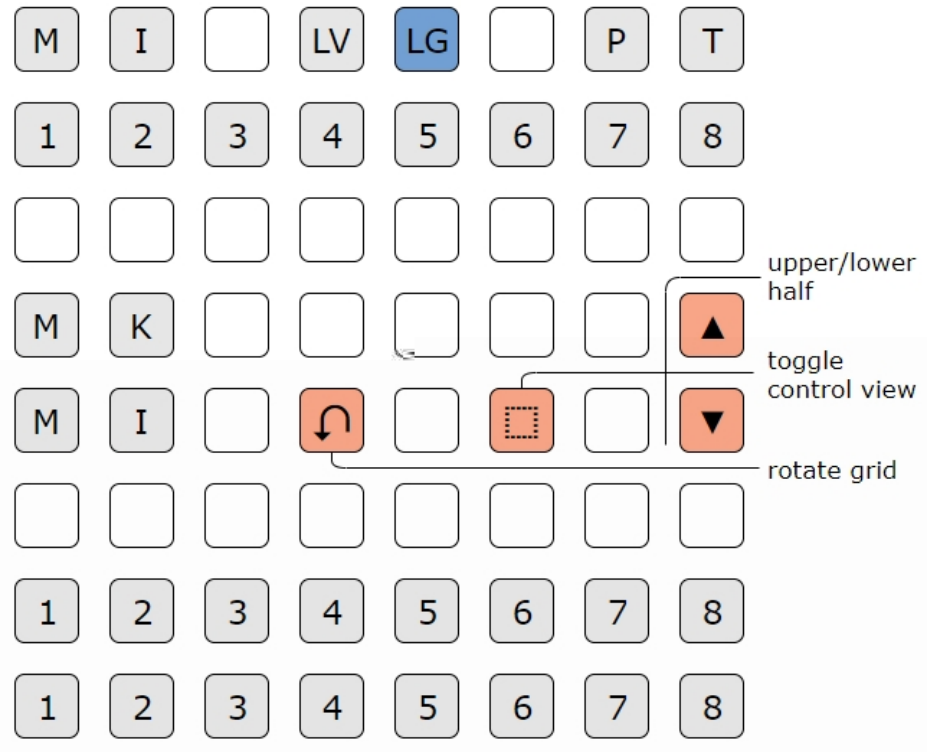


Figure 7: live page with grid visualizer

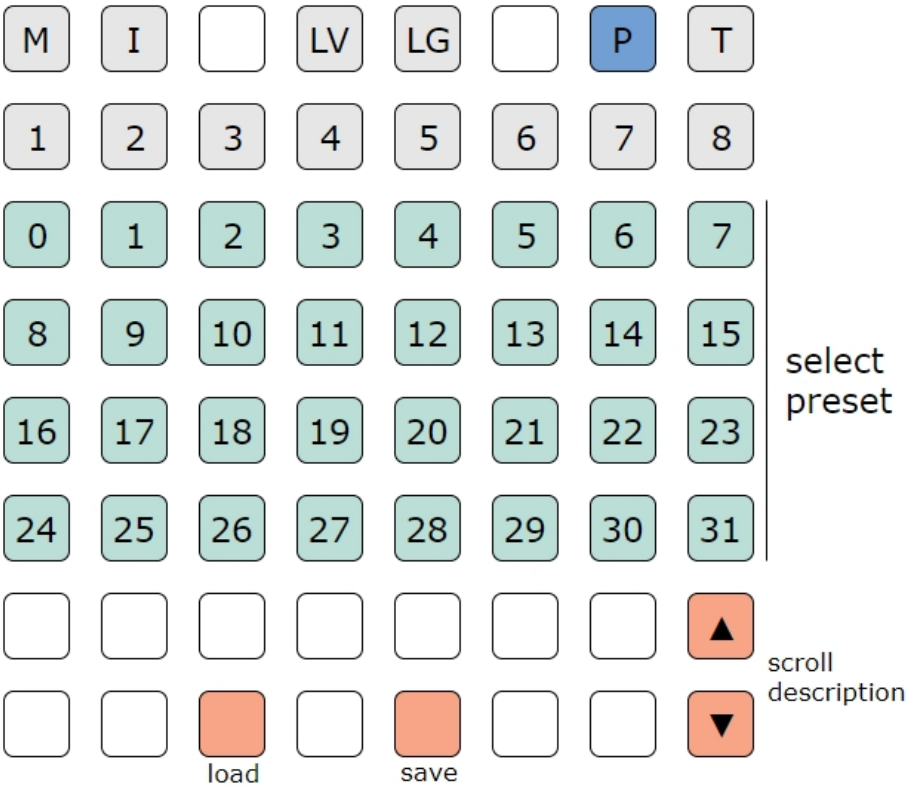


Figure 8: preset page

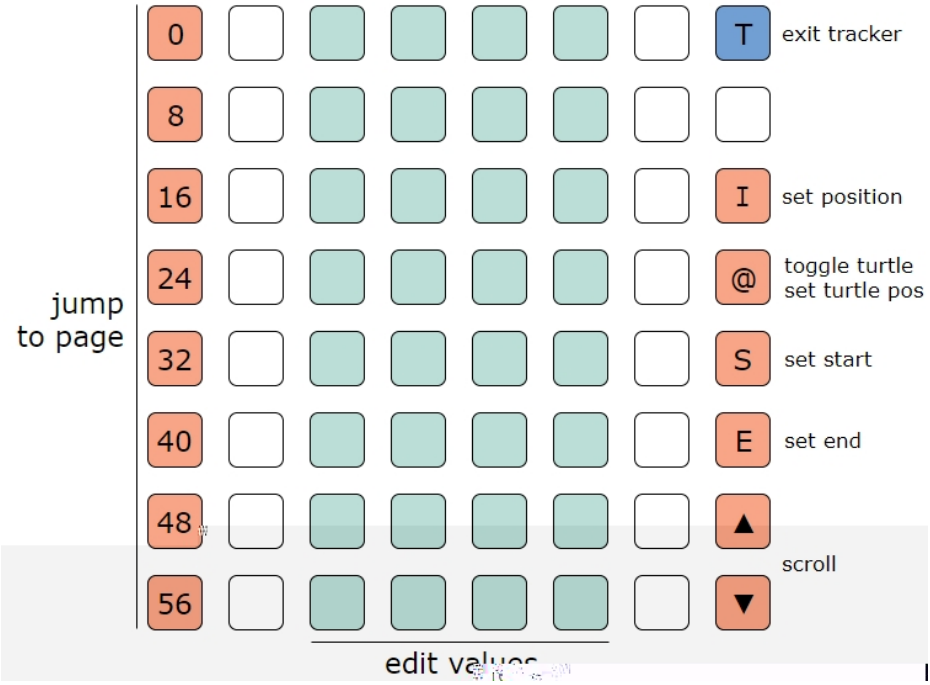


Figure 9: tracker page

end for one or more pattern banks - useful for defining loops. If you need to select a loop that is longer than 8 steps you can set start/end points separately by using “set start” / “set end” buttons on the right - press and hold and then press where you want the position to be.

The remaining 2 buttons allow you to select the current pattern position and the turtle position. Pressing and releasing the turtle button will toggle the turtle on and off.

Alphabetical list of OPs and MODs

OP *(set)*	*(aliases)*	Description
`$F s`		execute script `s` as a function
`$F1 s p`		execute script `s` as a function with 1 parameter `p`
`$F2 s p1 p2`		execute script `s` as a function with 2 parameters
`$L s l`		execute script `s` line `l`
`$L1 s l p`		execute script `s` line `l` as a function with 1 parameter `p`
`$L2 s l p1 p2`		execute script `s` line `l` as a function with 2 parameters
`$S l`		execute line `l` within the same script as a function
`$S1 l p`		execute line `l` within the same script as a function with 1 parameter `p`
`$S2 l p1 p2`		execute line `l` within the same script as a function with 2 parameters
`& x y`		bitwise and `x` & `y`
`? x y z`		if condition `x` is true return `y`, otherwise return `z`
`@ (x)`		get or set the current pattern value under the turtle
`@BOUNCE (1)`		get whether the turtle fence mode is `BOUNCE`, or set it to `BOUNCE` with `1`
`@BUMP (1)`		get whether the turtle fence mode is `BUMP`, or set it to `BUMP` with `1`
`@DIR (x)`		get the direction of the turtle's @STEP in degrees or set it to `x`
`@F x1 y1 x2 y2`		set the turtle's fence to corners `x1`, `y1` and `x2`, `y2`
`@FX1 (x)`		get the left fence line or set it to `x`
`@FX2 (x)`		get the right fence line or set it to `x`
`@FY1 (x)`		get the top fence line or set it to `x`

OP *(set)*	*(aliases)*	Description
`@FY2 (x)`		get the bottom fence line or set it to `x`
`@MOVE x y`		move the turtle `x` cells in the X axis and `y` cells in the Y axis
`@SCRIPT (x)`		get which script runs when the turtle changes cells, or set it to `x`
`@SHOW (1)`		get whether the turtle is displayed on the TRACKER screen, or turn it on or off
`@SPEED (x)`		get the speed of the turtle's @STEP in cells per step or set it to `x`
`@STEP`		move `@SPEED`/100 cells forward in `@DIR`, triggering `@SCRIPT` on cell change
`@WRAP (1)`		get whether the turtle fence mode is `WRAP`, or set it to `WRAP` with `1`
`@X (x)`		get the turtle X coordinate, or set it to `x`
`@Y (x)`		get the turtle Y coordinate, or set it to `x`
`A (x)`		get / set the variable `A`, default 1
`ABS x`		absolute value of `x`
`ADD x y`	`+`	add `x` and `y` together
`AND x y`	`&&`	logical AND of `x` and `y`
`AND3 x y z`	`&&&`	logical AND of `x`, `y` and `z`
`AND4 x y z a`	`&&&&`	logical AND of `x`, `y`, `z` and `a`
`ARC.A (d)`		send arc encoder event for ring `n`, delta `d`
`ARC.A.LED n x`		read arc LED buffer for ring `n`, LED `x` clockwise from north
`ARC.APP (x)`		get/set active app
`ARC.G x y (z)`		get/set grid key on/off state (`z`) at position `x`, `y`
`ARC.G.LED x y`		get grid LED buffer at position `x`, `y`
`ARC.G.P x y`		simulate grid key press at position (`x`, `y`)
`ARP.DIV v d`		set voice clock divisor (euclidean length), range [1-32]
`ARP.ER v f d r`		set all euclidean rhythm
`ARP.FIL v f`		set voice euclidean fill, use 1 for straight clock division, range [1-32]
`ARP.GT v g`		set voice gate length [0-127], scaled/synced to course divisions of voice clock

OP *(set)*	*(aliases)*	Description
ARP.HLD h		0 disables key hold mode, other values enable
ARP.RES v		reset voice clock/pattern on next base clock tick
ARP.ROT v r		set voice euclidean rotation, range [-32, 32]
ARP.RPT v n s		set voice pattern repeat, n times [0-8], shifted by s semitones [-24, 24]
ARP.SHIFT v o		shift voice cv by standard tt pitch value (e.g. N 6, V -1, etc)
ARP.SLEW v t		set voice slew time in ms
ARP.STY y		set base arp style [0-7]
AVG x y		the average of x and y
B (x)		get / set the variable B, default 2
BCLR x y		clear bit y in value x
BGET x y		get bit y in value x
BPM x		milliseconds per beat in BPM x
BREAK	BRK	halts execution of the current script
BREV x		reverse bit order in value x
BSET x y		set bit y in value x
BTOG x y		toggle bit y in value x
C (x)		get / set the variable C, default 3
CHAOS x		get next value from chaos generator, or set the current value
CHAOS.ALG x		get or set the algorithm for the CHAOS generator. 0 = LOGISTIC, 1 = CUBIC, 2 = HENON, 3 = CELLULAR
CHAOS.R x		get or set the R parameter for the CHAOS generator
CROW.AR x y z t		Creates an envelope on output x, rising in y ms, falling in z ms, and reaching height t.
CROW.C1 x		Calls the function 'ii.self.call1(x)' on crow.
CROW.C2 x y		Calls the function 'ii.self.call2(x, y)' on crow.
CROW.C3 x y z		Calls the function 'ii.self.call3(x, y, z)' on crow.
CROW.C4 x y z t		Calls the function 'ii.self.call4(x, y, z, t)' on crow.
CROW.IN x		Gets voltage at input x.

OP *(set)*	*(aliases)*	Description
`CROW.LFO x y z t`		Starts an envelope on output `x` at rate `y` where 0 = 1Hz with 1v/octave scaling. `z` sets amplitude and `t` sets skew for assymetrical triangle waves.
`CROW.OUT x`		Gets voltage of output `x`.
`CROW.PULSE x y z t`		Creates a trigger pulse on output `x` with duration `y` (ms) to voltage `z` with polarity `t`.
`CROW.Q0`		Returns the result of calling the function 'crow.self.query0()'.
`CROW.Q1 x`		Returns the result of calling the function 'crow.self.query1(x)'.
`CROW.Q2 x y`		Returns the result of calling the function 'crow.self.query2(x, y)'.
`CROW.Q3 x y z`		Returns the result of calling the function 'crow.self.query3(x, y, z)'.
`CROW.RST`		Calls the function `crow.reset()` returning crow to default state.
`CROW.SEL x`		Sets target crow unit (1 (default), to 4).
`CROW.SLEW x y`		Sets output `x` slew rate to `y` milliseconds.
`CROW.V x y`		Sets output `x` to value `y`. Use `V y` for volts.
`CROW1: ...`		Send following CROW OPs to unit 1 ignoring the currently selected unit.
`CROW2: ...`		Send following CROW OPs to unit 2 ignoring the currently selected unit.
`CROW3: ...`		Send following CROW OPs to unit 3 ignoring the currently selected unit.
`CROW4: ...`		Send following CROW OPs to unit 4 ignoring the currently selected unit.
`CROWM: ...`		Send following CROW OPs to all units starting with selected unit.
`CV x (y)`		CV target value
`CV.CAL n mV1V mV3V`		Calibrate CV output `n`
`CV.CAL.RESET n`		Reset calibration data for CV output `n`
`CV.GET x`		Get current CV value
`CV.OFF x (y)`		CV offset added to output
`CV.SET x y`		Set CV value, ignoring slew
`CV.SLEW x (y)`		Get/set the CV slew time in ms
`CV.CV x`		get the current CV value for channel `x`

OP *(set)*	*(aliases)*	Description
CY.POS x (y)		get / set position of channel x (x = 0 to set all), position between 0-255
CY.PRE (x)		return current preset / load preset x
CY.RES x		reset channel x (0 = all)
CY.REV x		reverse channel x (0 = all)
D (x)		get / set the variable D, default 4
DEL x: ...		Delay command by x ms
DEL.B t b: ...		Trigger the command up to 16 times at intervals of t ms, with active intervals set in 16-bit bitmask b, LSB = immediate.
DEL.CLR		Clear the delay buffer
DEL.G x t n d: ...		Trigger the command once immediately and x - 1 times at ms intervals of t * (n/d)^n where n ranges from 0 to x - 1.
DEL.R x t: ...		Trigger the command following the colon once immediately, and delay x - 1 commands at t ms intervals
DEL.X x t: ...		Delay x commands at t ms intervals
DEVICE.FLIP		Flip the screen/inputs/outputs
DIV x y	/	divide x by y
DR.P b p s		Drum pattern helper, b is the drum bank (0-4), p is the pattern (0-215) and step is the step number (0-15), returns 0 or 1
DR.T b p q l s		Tresillo helper, b is the drum bank (0-4), p is first pattern (0-215), q is the second pattern (0-215), l is length (1-64), and step is the step number (0-length-1), returns 0 or 1
DR.V p s		Velocity helper. p is the pattern (0-19). s is the step number (0-15)
DRUNK (x)		changes by -1, 0, or 1 upon each read saving its state, setting will give it a new value for the next read
DRUNK.MAX (x)		set the upper bound for DRUNK, default 255
DRUNK.MIN (x)		set the lower bound for DRUNK, default 0
DRUNK.SEED (x)	DRUNK.SD	get / set the random number generator seed for the DRUNK op
DRUNK.WRAP (x)		should DRUNK wrap around when it reaches it's bounds, default 0

OP *(set)*	*(aliases)*	Description
ELIF x: ...		if all previous IF / ELIF fail, and x is not zero, execute command
ELSE: ...		if all previous IF / ELIF fail, excute command
EQ x y	==	does x equal y
ER f l i		Euclidean rhythm, f is fill (1-32), l is length (1-32) and i is step (any value), returns 0 or 1
EVERY x: ...	EV	run the command every x times the command is called
EXP x		exponentiation table lookup. 0-16383 range (Y 0-10)
EZ x	!	x is 0, equivalent to logical NOT
FADER x	FB	Reads the value of the FADER slider x; default return range is from 0 to 16383. Up to four Faderbanks can be addressed; x value between 1 and 16 correspond to Faderbank 1, x between 17 and 32 to Faderbank 2, etc…
FADER.CAL.MAX x	FB.C.MAX	Reads FADER x maximum position and assigns the maximum point
FADER.CAL.MIN x	FB.C.MIN	Reads FADER x minimum position and assigns a zero value
FADER.CAL.RESET x	FB.C.R	Resets the calibration for FADER x
FADER.SCALE x y z	FB.S	Set static scaling of the FADER x to between min and max.
FLIP (x)		returns the opposite of its previous state (0 or 1) on each read (also settable)
FR (x)		get/set the return value when a script is called as a function
G.BTN id x y w h type level script		initialize button
G.BTN.EN id (x)		enable/disable button or check if enabled
G.BTN.L id (level)		get/set button level
G.BTN.PR id action		emulate button press/release
G.BTN.SW id		switch button
G.BTN.V id (value)		get/set button value
G.BTN.X id (x)		get/set button x coordinate
G.BTN.Y id (y)		get/set button y coordinate

OP *(set)*	*(aliases)*	Description
G.BTNI		id of last pressed button
G.BTNL (level)		get/set level of last pressed button
G.BTNP (value)		get/set value of last pressed button
G.BTNX (x)		get/set x of last pressed button
G.BTNY (y)		get/set y of last pressed button
G.BTX id x y w h type level script columns rows		initialize multiple buttons
G.CLR		clear all LEDs
G.DIM level		set dim level
G.FDR id x y w h type level script		initialize fader
G.FDR.EN id (x)		enable/disable fader or check if enabled
G.FDR.L id (level)		get/set fader level
G.FDR.M id (value)		get/set fader value
G.FDR.PR id value		emulate fader press
G.FDR.P id (value)		get/set scaled fader value
G.FDR.X id (x)		get/set fader x coordinate
G.FDR.Y id (y)		get/set fader y coordinate
G.FDRI		id of last pressed fader
G.FDRL (level)		get/set level of last pressed fader
G.FDRM (value)		get/set value of last pressed fader
G.FDRP (value)		get/set scaled value of last pressed fader
G.FDRX (x)		get/set x of last pressed fader
G.FDRY (y)		get/set y of last pressed fader
G.FDX id x y w h type level script columns rows		initialize multiple faders
G.GBT group id x y w h type level script		initialize button in group
G.GBTN.C group		get count of currently pressed
G.GBTN.H group		get button block height
G.GBTN.I group index		get id of pressed button
G.GBTN.L group odd_level even_level		set level for group buttons
G.GBTN.P group value		set value for group buttons

OP *(set)*	*(aliases)*	Description
G.GBTN.W group		get button block width
G.GBTN.X1 group		get leftmost pressed x
G.GBTN.X2 group		get rightmost pressed x
G.GBTN.Y1 group		get highest pressed y
G.GBTN.Y2 group		get lowest pressed y
G.GBX group id x y w h type level script columns rows		initialize multiple buttons in group
G.GFD grp id x y w h type level script		initialize fader in group
G.GFDR.L group odd_level even_level		set level for group faders
G.GFDR.N group value		set value for group faders
G.GFDR.RN group min max		set range for group faders
G.GFDR.V group value		set scaled value for group faders
G.GFX group id x y w h type level script columns rows		initialize multiple faders in group
G.GRP (id)		get/set current group
G.GRP.EN id (x)		enable/disable group or check if enabled
G.GRP.RST id		reset all group controls
G.GRP.SC id (script)		get/set group script
G.GRP.SW id		switch groups
G.GRPI		get last group
G.KEY x y action		emulate grid press
G.LED x y (level)		get/set LED
G.LED.C x y		clear LED
G.RCT x1 y1 x2 y2 fill border		draw rectangle
G.REC x y w h fill border		draw rectangle
G.ROTATE x		set grid rotation
G.RST		full grid reset
GT x y	>	x is greater than y
GTE x y	>=	x is greater than or equal to y

OP *(set)*	*(aliases)*	Description
`HZ x`		converts 1V/OCT value `x` to Hz/Volt value, useful for controlling non-euro synths like Korg MS-20
`I (x)`		get / set the per-script variable `I`. *See also* `L:` *in control flow*
`I1`		get the first parameter when executing a script as a function
`I2`		get the second parameter when executing a script as a function
`I2M.AT x`		Send MIDI After Touch message with value `x` (0..127)
`I2M.B.CLR`		Clear the buffer, erasing all recorded notes in the buffer
`I2M.B.DIR x`		Set the play direction `x` (0..2) of the buffer
`I2M.B.END x`		Add a negative offset of `x` ms (0..32767) to the end of the buffer
`I2M.B.FB x`		Set the feedback length `x` (0..255) of the buffer
`I2M.B.L x`		Set the length of the buffer to `x` ms (0..32767)
`I2M.B.MODE x`		Set the buffer mode to `x` (0..1). 1) Digital 2) Tape
`I2M.B.NOFF x`		Set the note offset of recorded notes to `x` semitones (-127..127)
`I2M.B.NSHIFT x`		Set the note shift of recorded notes to `x` semitones (-127..127)
`I2M.B.R x`		Turn recording of notes into the buffer on or off
`I2M.B.SPE x`		Set the playing speed `x` (1..32767) of the buffer. `x = 100` is equivalent to 'normal speed', `x = 50` means double the speed, `x = 200` means half the speed, etc.
`I2M.B.START x`		Add an offset of `x` ms (0..32767) to the start of the buffer
`I2M.B.TOFF x`		Set the note duration offset ('time offset') of recorded notes to `x` ms (-16384..16383)
`I2M.B.TSHIFT x`		Set the note duration shift ('time shift') of recorded notes to `x` ms (-16384..16383)
`I2M.B.VOFF x`		Set the velocity offset of recorded notes to `x` (-127..127)

OP *(set)*	*(aliases)*	Description
`I2M.B.VSHIFT x`		Set the velocity shift of recorded notes to `x` (-127..127)
`I2M.C# ch x y z`		Play chord `x` (1..8) with root note `y` (-127..127) and velocity `z` (1..127) on channel `ch` (1..32)
`I2M.C.ADD x y`	`I2M.C+`	Add relative note `y` (-127..127) to chord `x` (0..8), use `x = 0` to add to all chords
`I2M.C.B x y`		Clear and define chord `x` (0..8) using reverse binary notation (`B...`)
`I2M.C.CLR x`		Clear chord `x` (0..8), use `x = 0` to clear all chords
`I2M.C.DEL x y`		Delete note at index `y` (0..7) from chord `x` (0..8), use `x = 0` to delete from all chords
`I2M.C.DIR x y`		Set play direction for chord `x` (0..8) to direction `y` (0..8)
`I2M.C.DIS x y z`		Set distortion of chord `x` (0..8) to `y` (-127..127) with anchor point `z` (0..16), use `x = 0` to set for all chords
`I2M.C.INS x y z`		Add note `z` (-127..127) to chord `x` (0..8) at index `y` (0..7), with `z` relative to the root note; use `x = 0` to insert into all chords
`I2M.C.INV x y`		Set inversion of chord `x` (0..8) to `y` (-32..32), use `x = 0` to set for all chords
`I2M.C.L x (y)`		Get current length / Set length of chord `x` (0..8) to `y` (1..8), use `x = 0` to set length of all chords
`I2M.C.QN x y z`		Get the transformed note number of a chord note for chord `x` (1..8) with root note `y` (-127..127) at index `z` (0..7)
`I2M.C.QV x y z`		Get the transformed note velocity of a chord note for chord `x` (1..8) with root velocity `y` (1..127) at index `z` (0..7)
`I2M.C.REF x y z`		Set reflection of chord `x` (0..8) to `y` iterations (-127..127) with anchor point `z` (0..16), use `x = 0` to set for all chords
`I2M.C.REV x y`		Set reversal of notes in chord `x` (0..8) to `y`. `y = 0` or an even number means not reversed, y = 1 or an uneven number means reversed. Use `x = 0` to set for all chords.

OP *(set)*	*(aliases)*	Description
`I2M.C.RM x y`	`I2M.C-`	Remove note `y` (-127..127) from chord `x` (0..8), use `x = 0` to remove from all chords
`I2M.C.ROT x y`		Set rotation of notes in chord `x` (0..8) to `y` steps (-127..127), use `x = 0` to set for all chords
`I2M.C.SC x y`		Set scale for chord `x` (0..8) based on chord `y` (0..8), use `x = 0` to set for all chords, use `y = 0` to remove scale
`I2M.C.SET x y z`		Set note at index `y` (0..7) in chord `x` (0..8) to note `z` (-127..127), use `x = 0` to set in all chords
`I2M.C.STR x y`		Set strumming of chord `x` (0..8) to `x` ms (0..32767), use `x = 0` to set for all chords
`I2M.C.TCUR w x y z`	`I2M.C.T~`	Set time curve to strumming for chord `w` (0..8) with curve type `x` (0..5), start value `y`% (0..32767) and end value `z`% (0..32767), use `w = 0` to set for all chords, use `x = 0` to turn off
`I2M.C.TRP x y`		Set transposition of chord `x` (0..8) to `y` (-127..127), use `x = 0` to set for all chords
`I2M.C.VCUR w x y z`	`I2M.C.V~`	Set velocity curve for chord `w` (0..8) with curve type `x` (0..5), start value `y`% (0..32767) and end value `z`% (0..32767), use `w = 0` to set for all chords, use `x = 0` to turn off
`I2M.CC x y`		Send MIDI CC message for controller `x` (0..127) with value `y` (0..127)
`I2M.CC# ch x y`		Send MIDI CC message for controller `x` (0..127) with value `y` (0..127) on channel `ch` (1..32)
`I2M.CC.OFF x (y)`		Get current offset / Set offset of values of controller `x` (0..127) to `y` (-127..127)
`I2M.CC.OFF# ch x (y)`		Get current offset / Set offset of values of controller `x` (0..127) to `y` (-127..127) for channel `ch` (1..32)
`I2M.CC.SET x y`		Send MIDI CC message for controller `x` (0..127) with value `y` (0..127), bypassing any slew settings
`I2M.CC.SET# ch x y`		Send MIDI CC message for controller `x` (0..127) with value `y` (0..127) on channel `ch` (1..32), bypassing any slew settings

OP *(set)*	*(aliases)*	Description
`I2M.CC.SLEW x (y)`		Get current slew time for controller `x` / Set slew time for controller `x` (0..127) to `y` ms (0..32767)
`I2M.CC.SLEW# ch x (y)`		Get current slew time for controller `x` / Set slew time for controller `x` (0..127) to `y` ms (0..32767) for channel `ch` (1..32)
`I2M.CCV x y`		Send MIDI CC message for controller `x` (0..127) with volt value `y` (0..16383, 0..+10V)
`I2M.CCV# ch x y`		Send MIDI CC message for controller `x` (0..127) with volt value `y` (0..16383, 0..+10V) on channel `ch` (1..32)
`I2M.CH (x)`	`I2M.#`	Get currently set MIDI channel / Set MIDI channel `x` (1..16 for TRS, 17..32 for USB) for MIDI out
`I2M.CHORD x y z`	`I2M.C`	Play chord `x` (1..8) with root note `y` (-127..127) and velocity `z` (1..127)
`I2M.CLK`		Send MIDI Clock message, this still needs improvement …
`I2M.CONT`		Send MIDI Clock Continue message
`I2M.MAX x y`		Set maximum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for current channel
`I2M.MAX# ch x y`		Set maximum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for channel `ch` (0..32)
`I2M.MIN x y`		Set minimum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for current channel
`I2M.MIN# ch x y`		Set minimum note number for MIDI notes to `x` (0..127), using mode `y` (0..3), for channel `ch` (0..32)
`I2M.MUTE (x)`		Get mute state / Set mute state of current MIDI channel to `x` (0..1)
`I2M.MUTE# (x)`		Get mute state / Set mute state of MIDI channel `ch` to `x` (0..1)
`I2M.N# ch x y`		Send MIDI Note On message for note number `x` (0..127) with velocity `y` (1..127) on channel `ch` (1..32)

OP *(set)*	*(aliases)*	Description
`I2M.NO# ch x`		Send a manual MIDI Note Off message for note number `x` (0..127) on channel `ch` (1..32)
`I2M.NOTE x y`	`I2M.N`	Send MIDI Note On message for note number `x` (0..127) with velocity `y` (1..127) on current channel
`I2M.NOTE.O x`	`I2M.NO`	Send a manual MIDI Note Off message for note number `x` (0..127)
`I2M.NRPN x y z`		Send MIDI NRPN message (high-res CC) for parameter MSB `x` and LSB `y` with value `y` (0..16383)
`I2M.NRPN# ch x y z`		Send MIDI NRPN message (high-res CC) for parameter MSB `x` and LSB `y` with value `y` (0..16383) on channel `ch` (1..32)
`I2M.NRPN.OFF x y (z)`		Get current offset / Set offset of values of NRPN messages to `z` (-16384..16383)
`I2M.NRPN.OFF# ch x y (z)`		Get current offset / Set offset of values of NRPN messages to `z` (-16384..16383) for channel `ch` (1..32)
`I2M.NRPN.SET x y z`		Send MIDI NRPN message for parameter MSB `x` and LSB `y` with value `y` (0..16383), bypassing any slew settings
`I2M.NRPN.SET# ch x y z`		Send MIDI NRPN message for parameter MSB `x` and LSB `y` with value `y` (0..16383) on channel `ch` (1..32), bypassing any slew settings
`I2M.NRPN.SLEW x y (z)`		Get current slew time / Set slew time for NRPN messages to `z` ms (0..32767)
`I2M.NRPN.SLEW# ch x y (z)`		Get current slew time / Set slew time for NRPN messages to `z` ms (0..32767) for channel `ch` (1..32)
`I2M.NT x y z`		Send MIDI Note On message for note number `x` (0..127) with velocity `y` (1..127) and note duration `z` ms (0..32767)
`I2M.NT# ch x y z`		Send MIDI Note On message for note number `x` (0..127) with velocity `y` (1..127) and note duration `z` ms (0..32767) on channel `ch` (1..32)
`I2M.PANIC`		Send MIDI Note Off messages for all notes on all channels, and reset note duration, shift, repetition, ratcheting, min/max

OP *(set)*	*(aliases)*	Description
I2M.PB x		Send MIDI Pitch Bend message with value x (-8192..8191)
I2M.PRG x		Send MIDI Program Change message for program x (0..127)
I2M.Q.CC x		Get current value (0..127) of controller x (0..127) received via MIDI in
I2M.Q.CH (x)	I2M.Q.#	Get currently set MIDI channel / Set MIDI channel x (1..16) for MIDI in
I2M.Q.LATCH x		Turn on or off 'latching' for MIDI notes received via MIDI in
I2M.Q.LC		Get the latest controller number (0..127) received via MIDI in
I2M.Q.LCC		Get the latest controller value (0..127) received via MIDI in
I2M.Q.LCH		Get the latest channel (1..16) received via MIDI in
I2M.Q.LN		Get the note number (0..127) of the latest Note On received via MIDI in
I2M.Q.LO		Get the note number (0..127) of the latest Note Off received via MIDI in
I2M.Q.LV		Get the velocity (1..127) of the latest Note On received via MIDI in
I2M.Q.NOTE x	I2M.Q.N	Get x (0..7) last note number (0..127) received via MIDI in
I2M.Q.VEL x	I2M.Q.V	Get x (0..7) last note velocity (1..127) received via MIDI in
I2M.RAT (x)		Get current ratcheting / Set ratcheting of MIDI notes to x ratchets (1..127) for current channel
I2M.RAT# ch x		Get current ratcheting / Set ratcheting of MIDI notes to x ratchets (1..127) for channel ch (0..32)
I2M.REP (x)		Get current repetition / Set repetition of MIDI notes to x repetitions (1..127) for current channel
I2M.REP# ch x		Get current repetition / Set repetition of MIDI notes to x repetitions (1..127) for channel ch (0..32)

OP *(set)*	*(aliases)*	Description
I2M.S# ch (x)		Get current transposition / Set transposition of MIDI notes to x semitones (-127..127) for channel ch (0..32)
I2M.SHIFT (x)	I2M.S	Get current transposition / Set transposition of MIDI notes to x semitones (-127..127) for current channel
I2M.SOLO (x)		Get solo state / Set solo state of current MIDI channel to x (0..1)
I2M.SOLO# (x)		Get solo state / Set solo state of MIDI channel ch to x (0..1)
I2M.START		Send MIDI Clock Start message
I2M.STOP		Send MIDI Clock Stop message
I2M.T# ch (x)		Get current note duration / Set note duration of MIDI notes to x ms (0..32767) for channel ch (0..32).
I2M.TIME (x)	I2M.T	Get current note duration / Set note duration of MIDI notes to x ms (0..32767) for current channel
IF x: ...		if x is not zero execute command
IIA (address)		Set I2C address or get the currently selected address
IIB cmd		Execute the specified query and get a byte value back
IIB1 cmd value		Execute the specified query with 1 parameter and get a byte value back
IIB2 cmd value1 value2		Execute the specified query with 2 parameters and get a byte value back
IIB3 cmd value1 value2 value3		Execute the specified query with 3 parameters and get a byte value back
IIBB1 cmd value		Execute the specified query with 1 byte parameter and get a byte value back
IIBB2 cmd value1 value2		Execute the specified query with 2 byte parameters and get a byte value back
IIBB3 cmd value1 value2 value3		Execute the specified query with 3 byte parameters and get a byte value back
IIQ cmd		Execute the specified query and get a value back
IIQ1 cmd value		Execute the specified query with 1 parameter and get a value back

OP *(set)*	*(aliases)*	Description
IIQ2 cmd value1 value2		Execute the specified query with 2 parameters and get a value back
IIQ3 cmd value1 value2 value3		Execute the specified query with 3 parameters and get a value back
IIQB1 cmd value		Execute the specified query with 1 byte parameter and get a value back
IIQB2 cmd value1 value2		Execute the specified query with 2 byte parameters and get a value back
IIQB3 cmd value1 value2 value3		Execute the specified query with 3 byte parameters and get a value back
IIS cmd		Execute the specified command
IIS1 cmd value		Execute the specified command with 1 parameter
IIS2 cmd value1 value2		Execute the specified command with 2 parameters
IIS3 cmd value1 value2 value3		Execute the specified command with 3 parameters
IISB1 cmd value		Execute the specified command with 1 byte parameter
IISB2 cmd value1 value2		Execute the specified command with 2 byte parameters
IISB3 cmd value1 value2 value3		Execute the specified command with 3 byte parameters
IN		Get the value of IN jack (0-16383)
IN.CAL.MAX		Reads the input CV and assigns the voltage to the max point
IN.CAL.MIN		Reads the input CV and assigns the voltage to the zero point
IN.CAL.RESET		Resets the input CV calibration
IN.SCALE MIN MAX		Set static scaling of the IN CV to between MIN and MAX.
INIT		clears all state data
INIT.CV x		clears all parameters on CV associated with output x
INIT.CV.ALL		clears all parameters on all CV's
INIT.DATA		clears all data held in all variables
INIT.P x		clears pattern number x
INIT.P.ALL		clears all patterns

OP *(set)*	*(aliases)*	Description
INIT.SCENE		loads a blank scene
INIT.SCRIPT x		clear script number x
INIT.SCRIPT.ALL		clear all scripts
INIT.TIME x		clear time on trigger x
INIT.TR x		clear all parameters on trigger x
INIT.TR.ALL		clear all triggers
INR l x h	><	x is greater than l and less than h (within range)
INRI l x h	>=<	x is greater than or equal to l and less than or equal to h (within range, inclusive)
] (x)		get / set the per-script variable]
]F.ADDR x		Sets JF II address (1 = primary, 2 = secondary). Use with only one JF on the bus! Saves to JF internal memory, so only one-time config is needed.
]F.CURVE		Gets value of CURVE knob.
]F.FM		Gets value of FM knob.
]F.GOD x		Redefines C3 to align with the 'God' note. x = 0 sets A to 440, x = 1 sets A to 432.
]F.INTONE		Gets value of INTONE knob and CV offset.
]F.MODE x		Set the current choice of standard functionality, or Just Type alternate modes (Speed switch to Sound for Synth, Shape for Geode). You'll likely want to put JF.MODE x in your Teletype INIT scripts. x = nonzero activates alternative modes. 0 restores normal.
]F.NOTE x y		Synth: polyphonically allocated note sequencing. Works as JF.VOX with chan selected automatically. Free voices will be taken first. If all voices are busy, will steal from the voice which has been active the longest. x = pitch relative to C3, y = velocity. Geode: works as JF.VOX with dynamic allocation of channel. Assigns the rhythmic stream to the oldest unused channel, or if all are busy, the longest running channel. x = division, y = number of repeats.

OP *(set)*	*(aliases)*	Description
`]F.PITCH x y`		Change pitch without retriggering. `x` = channel, `y` = pitch relative to C3.
`]F.POLY x y`		As `]F.NOTE` but across dual JF. Switches between primary and secondary units every 6 notes or until reset using `]F.POLY.RESET`.
`]F.POLY.RESET`		Resets `]F.POLY` note count.
`]F.QT x`		When non-zero, all events are queued & delayed until the next quantize event occurs. Using values that don't align with the division of rhythmic streams will cause irregular patterns to unfold. Set to 0 to deactivate quantization. `x` = division, 0 deactivates quantization, 1 to 32 sets the subdivision & activates quantization.
`]F.RAMP`		Gets value of RAMP knob.
`]F.RMODE x`		Set the RUN state of Just Friends when no physical jack is present. (`0` = run off, non-zero = run on)
`]F.RUN x`		Send a 'voltage' to the RUN input. Requires `]F.RMODE 1` to have been executed, or a physical cable in JF's input. Thus Just Friend's RUN modes are accessible without needing a physical cable & control voltage to set the RUN parameter. use `]F.RUN V x` to set to `x` volts. The expected range is V -5 to V 5
`]F.SEL x`		Sets target JF unit (`1` = primary, `2` = secondary).
`]F.SHIFT x`		Shifts the transposition of Just Friends, regardless of speed setting. Shifting by V 1 doubles the frequency in sound, or doubles the rate in shape. `x` = pitch, use `M x` for semitones, or `V y` for octaves.
`]F.SPEED`		Gets value of SPEED switch (`1` = sound, `0` = shape).

OP *(set)*	*(aliases)*	Description
`]F.TICK x`		Sets the underlying timebase of the Geode. `x` = clock. 0 resets the timebase to the start of measure. 1 to 48 shall be sent repetitively. The value representing ticks per measure. 49 to 255 sets beats-per-minute and resets the timebase to start of measure.
`]F.TIME`		Gets value of TIME knob and CV offset.
`]F.TR x y`		Simulate a TRIGGER input. `x` is channel (`0` = all primary JF channels, `1..6` = primary JF, `7..12` = secondary JF, `-1` = all channels both JF) and y is state (`0` or `1`)
`]F.TSC`		Gets value of MODE switch (`0` = transient, `1` = sustain, `2` = cycle).
`]F.TUNE x y z`		Adjust the tuning ratios used by the INTONE control. `x` = channel, `y` = numerator (set the multiplier for the tuning ratio), `z` = denominator (set the divisor for the tuning ratio). `]F.TUNE 0 0 0` resets to default ratios.
`]F.VOX x y z`		Synth mode: create a note at the specified channel, of the defined pitch & velocity. All channels can be set simultaneously with a chan value of 0. `x` = channel, `y` = pitch relative to C3, `z` = velocity (like `]F.VTR`). Geode mode: Create a stream of rhythmic envelopes on the named channel. `x` = channel, `y` = division, `z` = number of repeats.
`]F.VTR x y`		Like `]F.TR` with added volume control. Velocity is scaled with volts, so try `V 5` for an output trigger of 5 volts. Channels remember their latest velocity setting and apply it regardless of TRIGGER origin (digital or physical). `x` = channel, `0` sets all channels. `y` = velocity, amplitude of output in volts. eg `]F.VTR 1 V 4`.
`]F0: ...`		Send following JF OPs to both units starting with selected unit.
`]F1: ...`		Send following JF OPs to unit 1 ignoring the currently selected unit.

OP *(set)*	*(aliases)*	Description
]F2: ...		Send following JF OPs to unit 2 ignoring the currently selected unit.
]I x y		just intonation helper, precision ratio divider normalised to 1V
K (x)		get / set the per-script variable K
KILL		clears stack, clears delays, cancels pulses, cancels slews, disables metronome
KR.CLK x		advance the clock for channel x (channel must have teletype clocking enabled)
KR.CUE (x)		get/set the cued pattern
KR.CV x		get the current CV value for channel x
KR.DIR (x)		get/set the step direction
KR.DUR x		get the current duration value for channel x
KR.L.LEN x y (z)		get length of track x, parameter y / set to z
KR.L.ST x y (z)		get loop start for track x, parameter y / set to z
KR.MUTE x (y)		get/set mute state for channel x (1 = muted, 0 = unmuted)
KR.PAT (x)		get/set current pattern
KR.PERIOD (x)		get/set internal clock period
KR.PG (x)		get/set the active page
KR.POS x y (z)		get/set position z for track z, parameter y
KR.PRE (x)		return current preset / load preset x
KR.RES x y		reset position to loop start for track x, parameter y
KR.SCALE (x)		get/set current scale
KR.TMUTE x		toggle mute state for channel x
L x y: ...		run the command sequentially with I values from x to y
LAST x		get value in milliseconds since last script run time
LIM x y z		limit the value x to the range y to z inclusive
LIVE.DASH x	LIVE.D	Show the dashboard with index x
LIVE.GRID	LIVE.G	Show grid visualizer in live mode
LIVE.OFF	LIVE.O	Show the default live mode screen
LIVE.VARS	LIVE.V	Show variables in live mode

OP *(set)*	*(aliases)*	Description
LROT x y	<<<	circular left shift x by y bits, wrapping around when bits fall off the end
LSH x y	<<	left shift x by y bits, in effect multiply x by 2 to the power of y
LT x y	<	x is less than y
LTE x y	<=	x is less than or equal to y
LV.CV x		get the current CV value for channel x
LV.L.DIR (x)		get/set loop direction
LV.L.LEN (x)		get/set loop length
LV.L.ST (x)		get/set loop start
LV.POS (x)		get/set current position
LV.PRE (x)		return current preset / load preset x
LV.RES x		reset, 0 for soft reset (on next ext. clock), 1 for hard reset
M (x)		get/set metronome interval to x (in ms), default 1000, minimum value 25
M! (x)		get/set metronome to experimental interval x (in ms), minimum value 2
M.ACT (x)		get/set metronome activation to x (0/1), default 1 (enabled)
M.RESET		hard reset metronome count without triggering
MAX x y		return the maximum of x and y
ME.CV x		get the current CV value for channel x
ME.PERIOD (x)		get/set internal clock period
ME.PRE (x)		return current preset / load preset x
ME.RES x		reset channel x (0 = all), also used as “start”
ME.SCALE (x)		get/set current scale
ME.STOP x		stop channel x (0 = all)
MI.$ x (y)		assign MIDI event type x to script y
MI.C		get the controller number (0..127) at index specified by variable I
MI.CC		get the controller value (0..127) at index specified by variable I
MI.CCH		get the controller event channel (1..16) at index specified by variable I

OP *(set)*	*(aliases)*	Description
MI.CCV		get the controller value scaled to 0..+10V range at index specified by variable I
MI.CL		get the number of controller events
MI.CLKD (x)		set clock divider to x (1-24) or get the current divider
MI.CLKR		reset clock counter
MI.LC		get the latest controller number (0..127)
MI.LCC		get the latest controller value (0..127)
MI.LCCV		get the latest controller value scaled to 0..16383 range (0..+10V)
MI.LCH		get the latest channel (1..16)
MI.LE		get the latest event type
MI.LN		get the latest Note On (0..127)
MI.LNV		get the latest Note On scaled to teletype range (shortcut for M MI.LN)
MI.LO		get the latest Note Off (0..127)
MI.LV		get the latest velocity (0..127)
MI.LVV		get the latest velocity scaled to 0..16383 range (0..+10V)
MI.N		get the Note On (0..127) at index specified by variable I
MI.NCH		get the Note On event channel (1..16) at index specified by variable I
MI.NL		get the number of Note On events
MI.NV		get the Note On scaled to 0..+10V range at index specified by variable I
MI.O		get the Note Off (0..127) at index specified by variable I
MI.OCH		get the Note Off event channel (1..16) at index specified by variable I
MI.OL		get the number of Note Off events
MI.V		get the velocity (0..127) at index specified by variable I
MI.VV		get the velocity scaled to 0..10V range at index specified by variable I
MID.SHIFT o		shift pitch CV by standard Teletype pitch value (e.g. M 6,V -1, etc)

OP *(set)*	*(aliases)*	Description
MID.SLEW t		set pitch slew time in ms (applies to all allocation styles except FIXED)
MIN x y		return the minimum of x and y
MOD x y	%	find the remainder after division of x by y
MUL x y	*	multiply x and y together
MUTE x (y)		Disable trigger input x
N x		converts an equal temperament note number to a value usable by the CV outputs (x in the range -127 to 127)
N.B d (s)		get degree d of scale/set scale root to r, scale to s, s is either bit mask (s >= 1) or scale preset (s < 1)
N.BX i d (s)		multi-index version of N.B, scale at i (index) 0 is shared with N.B
N.C r c d		Note Chord operator, r is the root note (0-127), c is the chord (0-12) and d is the degree (0-3), returns a value from the N table.
N.CS r s d c		Note Chord Scale operator, r is the root note (0-127), s is the scale (0-8), d is the scale degree (1-7) and c is the chord component (0-3), returns a value from the N table.
N.S r s d		Note Scale operator, r is the root note (0-127), s is the scale (0-8) and d is the degree (1-7), returns a value from the N table.
NE x y	!= , XOR	x is not equal to y
NR p m f s		Numeric Repeater, p is prime pattern (0-31), m is & mask (0-3), f is variation factor (0-16) and s is step (0-15), returns 0 or 1
NZ x		x is not 0
O (x)		auto-increments *after* each access, can be set, starting value 0
O.INC (x)		how much to increment O by on each invocation, default 1
O.MAX (x)		the upper bound for O, default 63
O.MIN (x)		the lower bound for O, default 0

OP *(set)*	*(aliases)*	Description
0.WRAP (x)		should 0 wrap when it reaches its bounds, default 1
OR x y	||	logical OR of x and y
OR3 x y z	|||	logical OR of x, y and z
OR4 x y z a	||||	logical OR of x, y, z and a
OTHER: ...		runs the command when the previous EVERY/SKIP did not run its command.
OUTR l x h	<>	x is less than l or greater than h (out of range)
OUTRI l x h	<=>	x is less than or equal to l or greater than or equal to h (out of range, inclusive)
P x (y)		get/set the value of the working pattern at index x
P.+ x y		increase the value of the working pattern at index x by y
P.+W x y a b		increase the value of the working pattern at index x by y and wrap it to a..b range
P.- x y		decrease the value of the working pattern at index x by y
P.-W x y a b		decrease the value of the working pattern at index x by y and wrap it to a..b range
P.END (x)		get/set the end location of the working pattern, default 63
P.HERE (x)		get/set value at current index of working pattern
P.I (x)		get/set index position for the working pattern.
P.INS x y		insert value y at index x of working pattern, shift later values down, destructive to loop length
P.L (x)		get/set pattern length of the working pattern, non-destructive to data
P.MAP: ...		apply the 'function' to each value in the active pattern, I takes each pattern value
P.MAX		find the first maximum value in the pattern between the START and END for the working pattern and return its index

OP *(set)*	*(aliases)*	Description
P.MIN		find the first minimum value in the pattern between the START and END for the working pattern and return its index
P.N (x)		get/set the pattern number for the working pattern, default 0
P.NEXT (x)		increment index of working pattern then get/set value
P.POP		return and remove the value from the end of the working pattern (like a stack), destructive to loop length
P.PREV (x)		decrement index of working pattern then get/set value
P.PUSH x		insert value x to the end of the working pattern (like a stack), destructive to loop length
P.REV		reverse the values in the active pattern (between its START and END)
P.RM x		delete index x of working pattern, shift later values up, destructive to loop length
P.RND		return a value randomly selected between the start and the end position
P.ROT n		rotate the values in the active pattern n steps (between its START and END, negative rotates backward)
P.SEED (x)	P.SD	get / set the random number generator seed for the P.RND and PM.RND ops
P.SHUF		shuffle the values in active pattern (between its START and END)
P.START (x)		get/set the start location of the working pattern, default 0
P.WRAP (x)		when the working pattern reaches its bounds does it wrap (0/1), default 1 (enabled)
PARAM	PRM	Get the value of PARAM knob (0-16383)
PARAM.CAL.MAX		Reads the Parameter Knob maximum position and assigns the maximum point
PARAM.CAL.MIN		Reads the Parameter Knob minimum position and assigns a zero value
PARAM.CAL.RESET		Resets the Parameter Knob calibration

OP *(set)*	*(aliases)*	Description
PARAM.SCALE min max		Set static scaling of the PARAM knob to between MIN and MAX.
PM x y (z)		get/set the value of pattern x at index y
PM.+ x y z		increase the value of pattern x at index y by z
PM.+W x y z a b		increase the value of pattern x at index y by z and wrap it to a..b range
PM.- x y z		decrease the value of pattern x at index y by z
PM.-W x y z a b		decrease the value of pattern x at index y by z and wrap it to a..b range
PM.END x (y)		get/set the end location of the pattern x, default 63
PM.HERE x (y)		get/set value at current index of pattern x
PM.I x (y)		get/set index position for pattern x
PM.INS x y z		insert value z at index y of pattern x, shift later values down, destructive to loop length
PM.L x (y)		get/set pattern length of pattern x. non-destructive to data
PM.MAP x: ...		apply the 'function' to each value in pattern x, I takes each pattern value
PM.MAX x		find the first maximum value in the pattern between the START and END for pattern x and return its index
PM.MIN x		find the first minimum value in the pattern between the START and END for pattern x and return its index
PM.NEXT x (y)		increment index of pattern x then get/set value
PM.POP x		return and remove the value from the end of pattern x (like a stack), destructive to loop length
PM.PREV x (y)		decrement index of pattern x then get/set value
PM.PUSH x y		insert value y to the end of pattern x (like a stack), destructive to loop length
PM.REV x		reverse the values in pattern x

OP *(set)*	*(aliases)*	Description
PN.RM x y		delete index y of pattern x, shift later values up, destructive to loop length
PN.RND x		return a value randomly selected between the start and the end position of pattern x
PN.ROT x n		rotate the values in pattern x (between its START and END, negative rotates backward)
PN.SHUF x		shuffle the values in pattern x (between its START and END)
PN.START x (y)		get/set the start location of pattern x, default 0
PN.WRAP x (y)		when pattern x reaches its bounds does it wrap (0/1), default 1 (enabled)
PRINT x (y)	PRT	Print a value on a live mode dashboard or get the printed value
PROB x: ...		potentially execute command with probability x (0-100)
PROB.SEED (x)	PROB.SD	get / set the random number generator seed for the PROB mod
Q (x)		Modify the queue entries
Q.2P (i)		Copy queue to current pattern/copy queue to pattern at index i
Q.ADD x (i)		Perform addition on elements in queue
Q.AVG (x)		Return the average of the queue
Q.CLR (x)		Clear queue
Q.DIV x (i)		Perform division on elements in queue
Q.GRW (x)		Get/set grow state
Q.I i (x)		Get/set value of elements at index
Q.MAX (x)		Get/set maximum value
Q.MIN (x)		Get/set minimum value
Q.MOD x (i)		Perform module (%) on elements in queue
Q.MUL x (i)		Perform multiplication on elements in queue
Q.N (x)		The queue length
Q.P2 (i)		Copy current pattern to queue/copy pattern at index i to queue
Q.REV		Reverse queue
Q.RND (x)		Get random element/randomize elements

OP *(set)*	*(aliases)*	Description
Q.SH (x)		Shift elements in queue
Q.SRT (SRT)		Sort all or part of queue
Q.SUB x (i)		Perform subtraction on elements in queue
Q.SUM (x)		Get sum of elements
QT x y		round x to the closest multiple of y (quantise)
QT.B x		quantize 1V/OCT signal x to scale defined by M.B
QT.BX i x		quantize 1V/OCT signal x to scale defined by M.BX in scale index i
QT.CS x r s d c		quantize 1V/OCT signal x to chord c (1-7) from scale s (0-8, reference N.S scales) at degree d (1-7) with root 1V/OCT pitch r
QT.S x r s		quantize 1V/OCT signal x to scale s (0-8, reference N.S scales) with root 1V/OCT pitch r
R (x)		get a random number/set R.MIN and R.MAX to same value x (effectively allowing R to be used as a global variable)
R.MAX x		set the upper end of the range from -32768 – 32767, default: 16383
R.MIN x		set the lower end of the range from -32768 – 32767, default: 0
RAND x	RND	generate a random number between 0 and x inclusive
RAND.SEED (x)	RAND.SD , R.SD	get / set the random number generator seed for R, RRAND, and RAND ops
RRAND x y	RRND	generate a random number between x and y inclusive
RROT x y	>>>	circular right shift x by y bits, wrapping around when bits fall off the end
RSH x y	>>	right shift x by y bits, in effect divide x by 2 to the power of y
S: ...		Place a command onto the stack
S.ALL		Execute all entries in the stack
S.CLR		Clear all entries in the stack
S.L		Get the length of the stack
S.POP		Execute the most recent entry

OP *(set)*	*(aliases)*	Description
SCALE a b x y i	SCL	scale i from range a to b to range x to y, i.e. i * (y - x) / (b - a)
SCALE a b i	SCL0	scale i from range 0 to a to range 0 to b
SCENE (x)		get the current scene number, or load scene x (0-31)
SCENE.G x		load scene x (0-31) without loading grid control states
SCENE.P x		load scene x (0-31) without loading pattern state
SCRIPT (x)	$	get current script number, or execute script x (1-10), recursion allowed
SCRIPT.POL x (p)	$.POL	get script x trigger polarity, or set polarity p (1 rising edge, 2 falling, 3 both)
SEED (x)		get / set the random number generator seed for all SEED ops
SGN x		sign function: 1 for positive, -1 for negative, 0 for 0
SKIP x: ...		run the command every time except the xth time.
STATE x		Read the current state of input x
SUB x y	-	subtract y from x
SYNC x		synchronizes *all* EVERY and SKIP counters to offset x.
T (x)		get / set the variable T, typically used for time, default 0
TI.IN x		reads the value of IN jack x; default return range is from -16384 to 16383 - representing -10V to +10V; return range can be altered by the TI.IN.MAP command
TI.IN.CALIB x y		calibrates the scaling for IN jack x; y of -1 sets the -10V point; y of 0 sets the 0V point; y of 1 sets the +10V point
TI.IN.INIT x		initializes IN jack x back to the default boot settings and behaviors; neutralizes mapping (but not calibration)
TI.IN.MAP x y z		maps the IN values for input jack x across the range y - z (default range is -16384 to 16383 - representing -10V to +10V)
TI.IN.N x		return the quantized note number for IN jack x using the scale set by TI.IN.SCALE

OP *(set)*	*(aliases)*	Description
TI.IN.QT x		return the quantized value for IN jack x using the scale set by TI.IN.SCALE; default return range is from -16384 to 16383 - representing -10V to +10V
TI.IN.SCALE x		select scale # y for IN jack x; scales listed in full description
TI.INIT d		initializes all of the PARAM and IN inputs for device number d (1-8)
TI.PARAM x	TI.PRM	reads the value of PARAM knob x; default return range is from 0 to 16383; return range can be altered by the TI.PARAM.MAP command
TI.PARAM.CALIB x y	TI.PRM.CALIB	calibrates the scaling for PARAM knob x; y of 0 sets the bottom bound; y of 1 sets the top bound
TI.PARAM.INIT x	TI.PRM.INIT	initializes PARAM knob x back to the default boot settings and behaviors; neutralizes mapping (but not calibration)
TI.PARAM.MAP x y z	TI.PRM.MAP	maps the PARAM values for input x across the range y - z (defaults 0-16383)
TI.PARAM.N x	TI.PRM.N	return the quantized note number for PARAM knob x using the scale set by TI.PARAM.SCALE
TI.PARAM.QT x	TI.PRM.QT	return the quantized value for PARAM knob x using the scale set by TI.PARAM.SCALE; default return range is from 0 to 16383
TI.PARAM.SCALE x	TI.PRM.SCALE	select scale # y for PARAM knob x; scales listed in full description
TI.RESET d		resets the calibration data for TXi number d (1-8) to its factory defaults (no calibration)
TI.STORE d		stores the calibration data for TXi number d (1-8) to its internal flash memory
TIME (x)		timer value, counts up in ms., wraps after 32s, can be set
TIME.ACT (x)		enable or disable timer counting, default 1
TO.CV x		CV target output x; y values are bipolar (-16384 to +16383) and map to -10 to +10
TO.CV.CALIB x		Locks the current offset (CV.OFF) as a calibration offset and saves it to persist between power cycles for output x.

OP *(set)*	*(aliases)*	Description
`TO.CV.INIT x`		initializes `CV` output `x` back to the default boot settings and behaviors; neutralizes offsets, slews, envelopes, oscillation, etc.
`TO.CV.LOG x y`		translates the output for `CV` output `x` to logarithmic mode `y`; `y` defaults to 0 (off); mode 1 is for 0-16384 (0V-10V), mode 2 is for 0-8192 (0V-5V), mode 3 is for 0-4096 (0V-2.5V), etc.
`TO.CV.M x y`		target the CV to note `y` for output `x`; `y` is indexed in the output's current `CV.SCALE`
`TO.CV.M.SET x y`		set the CV to note `y` for output `x`; `y` is indexed in the output's current `CV.SCALE` (ignoring `SLEW`)
`TO.CV.OFF x y`		set the CV offset for output `x`; `y` values are added at the final stage
`TO.CV.QT x y`		CV target output `x`; `y` is quantized to output's current `CV.SCALE`
`TO.CV.QT.SET x y`		set the CV for output `x` (ignoring `SLEW`); `y` is quantized to output's current `CV.SCALE`
`TO.CV.RESET x`		Clears the calibration offset for output `x`
`TO.CV.SCALE x y`		select scale # `y` for CV output `x`; scales listed in full description
`TO.CV.SET x y`		set the CV for output `x` (ignoring `SLEW`); `y` values are bipolar (-16384 to +16383) and map to -10 to +10
`TO.CV.SLEW x y`		set the slew amount for output `x`; `y` in milliseconds
`TO.CV.SLEW.M x y`		set the slew amount for output `x`; `y` in minutes
`TO.CV.SLEW.S x y`		set the slew amount for output `x`; `y` in seconds
`TO.ENV x y`		trigger the attack stage of output `x` when `y` changes to 1, or decay stage when it changes to 0
`TO.ENV.ACT x y`		activates/deactivates the AD envelope generator for the CV output `x`; `y` turns the envelope generator off (0 - default) or on (1); `CV` amplitude is used as the peak for the envelope and needs to be > 0 for the envelope to be perceivable

OP *(set)*	*(aliases)*	Description
TO.ENV.ATT x y		set the envelope attack time to y for CV output x; y in milliseconds (default 12 ms)
TO.ENV.ATT.M x y		set the envelope attack time to y for CV output x; y in minutes
TO.ENV.ATT.S x y		set the envelope attack time to y for CV output x; y in seconds
TO.ENV.DEC x y		set the envelope decay time to y for CV output x; y in milliseconds (default 250 ms)
TO.ENV.DEC.M x y		set the envelope decay time to y for CV output x; y in minutes
TO.ENV.DEC.S x y		set the envelope decay time to y for CV output x; y in seconds
TO.ENV.EOC x n		at the end of cycle of CV output x, fires a PULSE to the trigget output n
TO.ENV.EOR x n		at the end of rise of CV output x, fires a PULSE to the trigget output n
TO.ENV.LOOP x y		causes the envelope on CV output x to loop for y times
TO.ENV.TRIG x		triggers the envelope at CV output x to cycle; CV amplitude is used as the peak for the envelope and needs to be >0 for the envelope to be perceivable
TO.INIT d		initializes all of the TR and CV outputs for device number d (1-8)
TO.KILL d		cancels all TR pulses and CV slews for device number d (1-8)
TO.M d y		sets the 4 independent metronome intervals for device d (1-8) to y in milliseconds; default 1000
TO.M.ACT d y		sets the active status for the 4 independent metronomes on device d (1-8) to y (0/1); default 0 (disabled)
TO.M.BPM d y		sets the 4 independent metronome intervals for device d to y in Beats Per Minute
TO.M.COUNT d y		sets the number of repeats before deactivating for the 4 metronomes on device d to y; default 0 (infinity)

OP *(set)*	*(aliases)*	Description
`TO.M.M d y`		sets the 4 independent metronome intervals for device `d` to `y` in minutes
`TO.M.S d y`		sets the 4 independent metronome intervals for device `d` to `y` in seconds; default `1`
`TO.M.SYNC d`		synchronizes the 4 metronomes for device number `d` (1-8)
`TO.OSC x y`		Targets oscillation for CV output `x` to `y`
`TO.OSC.CTR x y`		centers the oscillation on CV output `x` to `y`; `y` values are bipolar (-16384 to +16383) and map to -10 to +10
`TO.OSC.CYC x y`		targets the oscillator cycle length to `y` for CV output `x` with the portamento rate determined by the `TO.OSC.SLEW` value; `y` is in milliseconds
`TO.OSC.CYC.M x y`		targets the oscillator cycle length to `y` for CV output `x` with the portamento rate determined by the `TO.OSC.SLEW` value; `y` is in minutes
`TO.OSC.CYC.M.SET x y`		sets the oscillator cycle length to `y` for CV output `x` (ignores `CV.OSC.SLEW`); `y` is in minutes
`TO.OSC.CYC.S x y`		targets the oscillator cycle length to `y` for CV output `x` with the portamento rate determined by the `TO.OSC.SLEW` value; `y` is in seconds
`TO.OSC.CYC.S.SET x y`		sets the oscillator cycle length to `y` for CV output `x` (ignores `CV.OSC.SLEW`); `y` is in seconds
`TO.OSC.CYC.SET x y`		sets the oscillator cycle length to `y` for CV output `x` (ignores `CV.OSC.SLEW`); `y` is in milliseconds
`TO.OSC.FQ x y`		targets oscillation for CV output `x` to frequency `y` in Hertz
`TO.OSC.FQ.SET x y`		targets oscillation for CV output `x` to frequency `y` in Hertz (ignores slew)
`TO.OSC.LFO x y`		Targets oscillation for CV output `x` to LFO frequency `y` in millihertz
`TO.OSC.LFO.SET x y`		Targets oscillation for CV output `x` to LFO frequency `y` in millihertz (ignores slew)
`TO.OSC.M x y`		targets oscillation for CV output `x` to note `y`

OP *(set)*	*(aliases)*	Description
`TO.OSC.M.SET x y`		sets oscillation for CV output `x` to note `y` (ignores slew)
`TO.OSC.PHASE x y`		sets the phase offset of the oscillator on CV output `x` to `y` (0 to 16383); `y` is the range of one cycle
`TO.OSC.QT x y`		targets oscillation for CV output `x` to `y`
`TO.OSC.QT.SET x y`		set oscillation for CV output `x` to `y`, quantized to the current scale (ignores slew)
`TO.OSC.RECT x y`		rectifies the polarity of the oscillator for output `x` to `y`; 0 is no rectification, +/-1 is partial rectification, +/-2 is full rectification
`TO.OSC.SCALE x y`		select scale # `y` for CV output `x`; scales listed in full description
`TO.OSC.SET x y`		set oscillation for CV output `x` to `y` (ignores slew)
`TO.OSC.SLEW x y`		sets the frequency slew time (portamento) for the oscillator on CV output `x` to `y`; `y` in milliseconds
`TO.OSC.SLEW.M x y`		sets the frequency slew time (portamento) for the oscillator on CV output `x` to `y`; `y` in minutes
`TO.OSC.SLEW.S x y`		sets the frequency slew time (portamento) for the oscillator on CV output `x` to `y`; `y` in seconds
`TO.OSC.SYNC x`		resets the phase of the oscillator on `CV` output `x` (relative to `TO.OSC.PHASE`)
`TO.OSC.WAVE x y`		set the waveform for output `x` to `y`; `y` range is 0-4500, blending between 45 waveforms
`TO.OSC.WIDTH x y`		sets the width of the pulse wave on output `x` to `y`; `y` is a percentage of total width (0 to 100); only affects waveform 3000
`TO.TR x y`		sets the `TR` value for output `x` to `y` (0/1)
`TO.TR.INIT x`		initializes `TR` output `x` back to the default boot settings and behaviors; neutralizes metronomes, dividers, pulse counters, etc.
`TO.TR.M x y`		sets the independent metronome interval for output `x` to `y` in milliseconds; default `1000`

OP *(set)*	*(aliases)*	Description
TO.TR.M.ACT x y		sets the active status for the independent metronome for output x to y (0/**1**); default 0 (disabled)
TO.TR.M.BPM x y		sets the independent metronome interval for output x to y in Beats Per Minute
TO.TR.M.COUNT x y		sets the number of repeats before deactivating for output x to y; default 0 (infinity)
TO.TR.M.M x y		sets the independent metronome interval for output x to y in minutes
TO.TR.M.MUL x y		multiplies the M rate on TR output x by y; y defaults to **1** - no multiplication
TO.TR.M.S x y		sets the independent metronome interval for output x to y in seconds; default **1**
TO.TR.M.SYNC x		synchronizes the PULSE for metronome on TR output number x
TO.TR.POL x y		sets the polarity for TR output n
TO.TR.PULSE x	TO.TR.P	pulses the TR value for output x for the duration set by TO.TR.TIME/S/M
TO.TR.PULSE.DIV x y	TO.TR.P.DIV	sets the clock division factor for TR output x to y
TO.TR.PULSE.MUTE x y	TO.TR.P.MUTE	mutes or un-mutes TR output x; y is 1 (mute) or 0 (un-mute)
TO.TR.TIME x y		sets the time for TR.PULSE on output n; y in milliseconds
TO.TR.TIME.M x y		sets the time for TR.PULSE on output n; y in minutes
TO.TR.TIME.S x y		sets the time for TR.PULSE on output n; y in seconds
TO.TR.TOG x		toggles the TR value for output x
TO.TR.WIDTH x y		sets the time for TR.PULSE on output n based on the width of its current metronomic value; y in percentage (0-100)
TOSS		randomly return 0 or 1
TOSS.SEED (x)	TOSS.SD	get / set the random number generator seed for the TOSS op
TR x (y)		Set trigger output x to y (0-1)
TR.POL x (y)		Set polarity of trigger output x to y (0-1)
TR.PULSE x	TR.P	Pulse trigger output x

OP *(set)*	*(aliases)*	Description
TR.TIME x (y)		Set the pulse time of trigger x to y ms
TR.TOG x		Flip the state of trigger output x
V x		converts a voltage to a value usable by the CV outputs (x between 0 and 10)
VM x		converts 1V/OCT value x to an equal temperament note number
VV x		converts a voltage to a value usable by the CV outputs (x between 0 and 1000, 100 represents 1V)
W x: ...		run the command while condition x is true
W/.SEL x		Sets target W/2.0 unit (1 = primary, 2 = secondary).
W/1: ...		Send following W/2.0 OPs to unit 1 ignoring the currently selected unit.
W/2: ...		Send following W/2.0 OPs to unit 2 ignoring the currently selected unit.
W/D.CLK		sends clock pulse for synchronization
W/D.CLK.RATIO mul div		set clock pulses per buffer time, with clock mul/div (s8)
W/D.CUT count divisions		jump to loop location as a fraction of loop length (u8)
W/D.FBK lvl		amount of feedback from read head to write head (s16V)
W/D.FILT cutoff		centre frequency of filter in feedback loop (s16V)
W/D.FREEZE is_active		deactivate record head to freeze the current buffer (s8)
W/D.FREQ volts		manipulate tape speed with musical values (s16V)
W/D.FREQ.RNG freq_range		TBD (s8)
W/D.LEN count divisions		set buffer loop length as a fraction of buffer time (u8)
W/D.MIX fade		fade from dry to delayed signal
W/D.MOD.AMT amount		set the amount (s16V) of delay line modulation to be applied
W/D.MOD.RATE rate		set the multiplier for the modulation rate (s16V)

OP *(set)*	*(aliases)*	Description
W/D.PLUCK volume		pluck the delay line with noise at volume (s16V)
W/D.POS count divisions		set loop start location as a fraction of buffer time (u8)
W/D.RATE mul		direct multiplier (s16V) of tape speed
W/D.TIME seconds		set delay buffer length in seconds (s16V), when rate == 1
W/S.AR.MODE is_ar		in attack-release mode, all notes are plucked and no release is required'
W/S.CURVE curve		cross-fade waveforms: -5=square, 0=triangle, 5=sine (s16V)
W/S.FM.ENV amount		amount of vactrol envelope applied to fm index, -5 to +5 (s16V)
W/S.FM.INDEX index		amount of FM modulation. -5=negative, 0=minimum, 5=maximum (s16V)
W/S.FM.RATIO n d		ratio of the FM modulator to carrier as a ratio, numerator / denominator. floating point values up to 20.0 supported (s16V)
W/S.LPG.SYM symmetry		vactrol attack-release ratio. -5=fastest attack, 5=long swells (s16V)
W/S.LPG.TIME time		vactrol time (s16V) constant. -5=drones, 0=vtl5c3, 5=blits
W/S.NOTE pitch level		dynamically assign a voice, set to pitch (s16V), strike with velocity (s16V)
W/S.PATCH jack param		patch a hardware jack (s8) to a param (s8) destination
W/S.PITCH voice pitch		set voice (s8) to pitch (s16V) in volts-per-octave
W/S.POLY pitch level		As W/S.NOTE but across dual W/. Switches between primary and secondary units every 4 notes or until reset using W/S.POLY.RESET.
W/S.POLY.RESET		Resets W/S.POLY note count.
W/S.RAMP ramp		waveform symmetry: -5=rampwave, 0=triangle, 5=sawtooth (NB: affects FM tone)
W/S.VEL voice velocity		strike the vactrol of voice (s8) at velocity (s16V) in volts
W/S.VOICES count		set number of polyphonic voices to allocate. use 0 for unison mode (s8)

OP *(set)*	*(aliases)*	Description
W/S.VOX voice pitch velocity		set **voice** (s8) to **pitch** (s16V) and strike the vactrol at **velocity** (s16V)
W/T.CLEARTAPE		WARNING! Erases all recorded audio on the tape!
W/T.ECHOMODE x		Set to 1 to playback before erase. 0 (default) erases first
W/T.ERASE.LVL level		Strength of erase head when recording. 0 is overdub, 1 is overwrite. Opposite of feedback (s16V)
W/T.FREQ f		Set speed as a 'frequency' (s16V) style value **f**. Maintains reverse state
W/T.LOOP.ACTIVE x		Set the state of looping (0/1)
W/T.LOOP.END		Set the current time as the loop end, and jump to start
W/T.LOOP.NEXT dir		Move loop brace forward/backward by length of loop. Pos = fwd, neg = bkwd. 0 jumps to loop start (eg. retriggers). (s8)
W/T.LOOP.SCALE x		Multiply (positive) or divide(negative) loop brace by **x**. Zero resets to original window (s8)
W/T.LOOP.START		Set the current time as the beginning of a loop
W/T.MONITOR.LVL gain		Level of input passed directly to output (s16V)
W/T.PLAY playback		Set the playback state. -1 will flip playback direction
W/T.REC x		Sets recording state (0/1)
W/T.REC.LVL gain		Level of input material recorded to tape (s16V)
W/T.REV		Reverse the direction of playback
W/T.SEEK sec sub		Move playhead relative to current position. **sec** is a whole number of seconds, **sub** is subseconds where 1V = 1 sec. Uses 2 args because TT only supports 16bit args. (s16V)
W/T.SPEED n d		Set speed as a rate, calculated by **n** / **d** (numerator/denominator). Negative values are reverse (s16V)

OP *(set)*	*(aliases)*	Description
W/T.TIME sec sub		Move playhead to an arbitrary location on tape. sec is a whole number of seconds, sub is subseconds where 1V = 1 sec. Uses 2 args because TT only supports 16bit args. (s16V)
WRAP x y z	WRP	limit the value x to the range y to z inclusive, but with wrapping
WS.CUE x		Go to a cuepoint relative to the playhead position. 0 retriggers the current location. 1 jumps to the next cue forward. -1 jumps to the previous cue in the reverse. These actions are relative to playback direction such that 0 always retriggers the most recently passed location
WS.LOOP x		Set the loop state on/off. 0 is off. Any other value turns loop on
WS.PLAY x		Set playback state and direction. 0 stops playback. 1 sets forward motion, while -1 plays in reverse
WS.REC x		Set recording mode. 0 is playback only. 1 sets overdub mode for additive recording. -1 sets overwrite mode to replace the tape with your input
X (x)		get / set the variable X, default 0
Y (x)		get / set the variable Y, default 0
Z (x)		get / set the variable Z, default 0
^ x y		bitwise xor x ^ y
| x y		bitwise or x
~ x		bitwise not, i.e.: inversion of x