Teletype Studies

Part 1: Navigating and Making Edges

Hello, world?

Teletype starts up in *LIVE* mode. This is a basic interactive terminal where you can type in commands. Be sure the keyboard is plugged in. Try typing in:

```
HELLO
```

And then press **ENTER**. You'll see *UNKNOWN WORD: HELLO*. Teletype doesn't have great manners.

Backspace the greeting and instead type simply `7` and hit enter. Teletype will acknowledge you've typed in the number `7`, and clear the prompt for a new command.

This is how *LIVE* mode works. You type commands and they are executed immediately. Live mode is indicated with the `>` character as the prompt.

Edges await your command

But really, now we will make something happen.

```
TR A 1
```

Executing this command will turn ON output A. The LED will light up, and voltage will be applied to this jack.

Here `TR` takes two parameters: the first (*A*) specifies the output number/letter (A-D) and the second (*1*) is the state: off (0) or on (1).

To turn off output A:

```
TR A 0
```

If we want to find out the current value of an output, simply leave off the last parameter:

```
TR A
```

In the terminal the value will be displayed.

There are a few other ways to use the TR outputs. If you want to switch the state of the output, use the following:

```
TR.TOG B
```

The TR output B will flip to on if previously off, or off if previously on.

THIS → Push the **UP ARROW**. Your last command will be recalled. Hit enter to execute it. Hit up several times to go through the command history. You can also push down (after going up) to navigate through previous commands.

Bip-bip-bip

```
TR.PULSE A
```

Make a pulse. Try it. If TR output A is on when the pulse is executed, you'll get an inverted pulse.

How long is that pulse? Type:

```
TR.TIME A
```

Teletype will tell you. By default it is 100, which is in milliseconds. To change this to 200ms:

```
TR.TIME A 200
```

Now try `TR.PULSE A` for longer pulses (remember to use **UP ARROW** for command recall).

OK when do we make music?

What makes these little commands interesting is the ability to string a bunch of them together into a *SCRIPT* assigned to a trigger input– as if some other part of your modular types furiously each time a pulse came in.

We switch between *LIVE* and *EDIT* mode by hitting **TAB**.

Upon switching to *EDIT* mode however, you'll likely notice a bunch of text suddenly on the screen– Teletype ships with the first *SCENE* loaded. Let's take a quick detour to clear the scene.

Scene loading

Let's get a blank scene as a starting point. Hit **ESC** to enter the *SCENE* page.

Scenes are listed along with their help texts. Use the brackets [and] to move through the available scenes. Hit **ENTER** to load the selected scene. In our

case, let's load scene 9, which should be blank. Upon load you'll be thrown back to *LIVE* mode.

Back to the script

Hit **TAB** to get into *EDIT* mode. You'll see the prompt change to `1` indicating that you're editing script 1.

- Each trigger input (1-8) has its own script.
- A script is executed on the positive (low-to-high) edge of a trigger or gate.
- Scripts can have up to 6 commands (lines).

The *EDIT* interface simply lets you edit the commands of the script. Type in:

```
TR.TOG A
```

Patch some sort of timer into trigger input 1. Now on each incoming trigger signal the trigger output A is being flipped.

After entering a command the edit line is moved down, so you can add another line. Go ahead and do that:

```
TR.TOG B
```

Now your script has two lines, and one input is making two outputs flip around.

Say you want to edit the first line of the script– simply use the **UP ARROW** to navigate up there. The command will be copied to the edit prompt. You can use the **DOWN ARROW** similarly to navigate back to the bottom.

So far we've been modifying script 1. Use the brackets [and] to switch scripts. Hit] to edit script 2:

```
TR.TOG B
TR.TOG C
```

Now trigger input 2 will flip outputs B and C.

If you continue hitting the brackets you'll stumble upon the `M` and `I` scripts– we'll be describing those in future chapters.

Scissors and glue

Navigate back to `TR.TOG C` and hit **ALT-C**. This will copy the whole command. Hit] to go to script 3, and then **ALT-V** to paste the command. You can also use **ALT-X** to cut.

You can copy/paste between *LIVE* and *EDIT* modes, so if you try something out in *LIVE* mode you can quickly **UP** then **ALT-C** to copy it prior to pasting it into a script by hitting **TAB**.

Sometimes you may want to *insert* a command at the current position rather than overwrite the existing command. Use **SHIFT-ENTER** to insert rather than overwrite the current command.

Save it for later

Teletype saves scenes to internal flash memory. Presently it can store up to 32 scenes. The currently loaded scene will be automatically recalled on power-up, but you have to store it to memory first.

Hit **ALT-ESC** to get into *SCENE (WRITE)* mode. You should still be on blank scene 9– you can use the brackets [and] to switch where you're writing to– careful not to overwrite existing scenes accidentally!

You can type a scene title into the top line, and "helper text" into the lines below. Typically it's good to describe what the scene does, and we've found it best to give "function descriptions" for the trigger input scripts. For example:

```
JUMPY EDGES

THINGS GO UP AND DOWN

1: TOGGLE A B
2: TOGGLE B C
```

When you're ready to save your scene, hit **ALT-ENTER**. It's now in flash and will run immediately on your next power-up.

EXAMPLE: JUMPY EDGES

This scene is featured in the banner video above.

The following scene uses all 8 trigger inputs to act on outputs A-D. Try patching outputs A-D as gates to various sound-altering inputs.

Inputs 1-4 flip two sequential outputs, and Input 5 will flip all outputs. Try having input triggers 1-4 all patched, running at slightly different rates. You'll notice patterns emerge.

Input 6 pulses all outputs. Inputs 7 and 8 change the pulse length. Given the pulse will invert the signal, try interjecting some pulses while scripts 1-5 are running for some variation.

```
M
```

```
I
```

```
TR.TOG A
TR.TOG B

1
```

```
TR.TOG B
TR.TOG C

2
```

```
TR.TOG C
TR.TOG D

3
```

```
TR.TOG D
TR.TOG A

4
```

```
TR.TOG A
TR.TOG B
TR.TOG C
TR.TOG D

5
```

```
TR.PULSE A
TR.PULSE B
TR.PULSE C
TR.PULSE D

6
```

```
TR.TIME A 100
TR.TIME B 140
TR.TIME C 180
TR.TIME D 220

7
```

```
TR.TIME A 500
TR.TIME B 440
TR.TIME C 380
TR.TIME D 320

8
```

Reference

Commands

```
TR x y          set trigger output x (A-D) to y (0-1)
TR.TOG x        flip trigger output x to the opposite state
TR.PULSE x      pulse trigger output x
TR.TIME x y     set pulse time of trigger x to y (ms)
```

Part 2: Curves and Repetition

Clean sweep

Welcome to the second level! For an optimal experience first load up a blank scene. Let's try something new and load a scene without using the keyboard:

- Touch the front panel key next to the USB port and boom, you're in *SCENE* mode
- Turn the **param** knob to scroll through the scenes
- Touch and hold the same front panel key to load the scene
- (Alternatively, to exit without loading, just hit the key quickly)

Scene loading this way is particularly helpful if you're using precomposed scenes during performance and would rather not have a keyboard in front of your synth.

What is a bit?

Before we make sounds, we have to have a talk. About numbers. I'll try to make it fast.

Eurorack modular uses control voltages in a range of 0 to 10 volts. This is a continuous, analog range. Computers (including Teletype) are generally not analog and represent values (numbers) as a series of digital bits. (Here's a better explanation). Teletype uses signed 16 bit numbers, so -16384 through 16384. Let us explain further:

The CV outputs 1-4 can create voltages between 0 and 10. Internally, Teletype sees this as 0 through 16384, which is 14 bit. But it's somewhat inconvenient to remember 16383 is equal to 10 volts– so instead we use a lookup table. In *LIVE* mode, type:

```
V 10
```

You'll see `16384` pop up. Try `V 5`. You'll get half that.

Making volts

Let's actually hear it. Plug output CV 1 into the frequency input of an oscillator, and patch that so it makes sound. We change the CV value with with the `CV` command:

```
CV 1 V 2
```

The first argument `1` is which CV output you're changing. This can be 1-4. We're then using the `V` command to look up the value of 2 volts. Effectively this is the same as typing `CV 1 3277`.

Oscillators should track one octave per volt, so right now the oscillator should be playing two octaves above wherever it is tuned. To return back to zero, type:

```
CV 1 0
```

We don't really need to type `V 0` because it's the same as `0`, but you can if you want.

But you may want some value besides just octaves. (Though, if you are making octaves-only music, I want to hear it– please e-mail tehn@monome.org). Teletype only understands whole numbers– you can't use decimal points. We designed it this way to keep it as simple as possible.

To get 2.5 volts, use the `VV` lookup:

```
VV 250
```

You'll get 4096 (honestly it's not really important what the value is). But why did we type `250`? `VV` takes a range of 0 to 1000, where 1000 can be thought of as "10.00". So 250 actually means 2.50. You always need to specify two decimal places, otherwise it'll be a bug, so remember!

```
2.5v  = VV 250
1.25v = VV 125
6.02v = VV 602
0.22v = VV 22
5v    = VV 500
```

Got it? Use `V` for single whole volts, and `VV` for fractional volts.

Get temperamental

You may have tried this:

```
CV 1 VV 250
```

That's a half octave above the second octave, which just happens to be a tritone– a real note. But try:

```
CV 1 VV 11
```

And you'll be in micro-tonal territory. Instead of dividing volts by 12 yourself, Teletype does it for you with the `N` lookup table.

```
CV 2 N 7
```

This command sets CV output 2 to 7 semitones up, in equal temperament tuning. (We'll discuss custom tuning tables in a later study!)

Try changing that last value to hear some pretty clear tonal relationships:

```
CV 2 N 0
CV 2 N 9
CV 2 N 12
CV 2 N 16
```

The `N` table works for values above 12, with a full range of 0 to 127.

Offset

In addition to setting a CV output directly with `CV`, we can change the final output value with an offset:

```
CV.OFF 1 N 12
```

Here we've just set the `CV.OFF` for the first CV output to `N 12` which is equivalent to `V 1` or one octave. This is an offset added to all CV changes. So now:

```
CV 1 V 1
```

With the offset, CV output 1 now has two volts showing.

This mechanism is helpful if you have a steady sequence of values going to the `CV` command, but want to modulate the entire channel. Think of this as a master "tuning" knob on an old synth (but with much greater range). Get into *EDIT* mode (hit **TAB**) and make a few scripts:

```
1:
CV 1 N 0

2:
CV 1 N 7

3:
CV.OFF 1 0

4:
CV.OFF 1 N 5
```

The first two scripts change the CV value, and the second two change the CV offset.

THIS → Trigger a script from the keyboard by holding **WIN** (aka meta) and pressing **1-8** (for example, **WIN-1** to trigger script 1.) This is great for trying things out while writing scripts without having to patch inputs into Teletype.

Try triggering scripts 1-4 in various ways. Yeah?

Bend and drift

Up until now all CV changes have been sharp– both the `CV` and `CV.OFF` commands change the output immediately upon execution.

Each CV channel has a slew parameter which gives a transition time for value changes. Get into *LIVE* mode and type:

```
CV.SLEW 1 1000
```

This sets the slew time of the first CV channel to 1000, which is 1000 ms, or 1 second. All future `CV` and `CV.OFF` commands will now smoothly transition to their target over this interval. Slew times can be up to 32 seconds long.

Try triggering scripts 1-4 now.

In *LIVE* mode you'll see the small diagonal line icon (in the top right) light up when a CV is actively slewing to a target.

Fly direct

You can interrupt a mid-slewing CV in two ways:

1. Set `CV.SLEW` to 0, then issue a new `CV` value; or,
2. Use `CV.SET`

`CV.SET` stops the current transition and immediately sets the CV channel to the given value. This is useful in cases where you don't want to change the `CV.SLEW` value.

```
CV.SET 1 N 10
```

This command will bypass slew and set CV output 1 to 10 semitones up. Try changing script 2 to `CV.SET`– it'll change the vibe.

Investigating and mirroring

`CV`, `CV.SLEW`, and `CV.OFFSET` can all be read as well as set. In *LIVE* mode you can do this:

```
CV 1
```

Which will return the value of CV 1. This is going to be a straight number, so if you previously assigned CV 1 to `N 9` you'll get 1229, for example.

But we can also use `CV 1` as a value to set other values! Say you want to set CV 2 to the value of CV 1:

```
CV 2 CV 1
```

CV 1 is "read" and then assigned to CV 2. It's important to note that `CV.OFF` is not taken into consideration here, so if one of the channels has an offset, the resulting voltage outputs will potentially be different. Of course, this may be the desired effect.

You can similarly read `CV.OFF` and `CV.SLEW` to use in other assignments.

Repeat this: this this this this

You've probably stumbled upon the `M` script while in *EDIT* mode. This is the Metronome script. It is executed at a fixed interval. Get into *EDIT* mode and add this to the *M* script.

```
TR.PULSE A
```

TR output A should be blinking at you, pretty fast. The metro time is determined by the variable `M`. Get back into *LIVE* mode and type:

```
M
```

This will report the value of `M` which is the current metronome time, in ms. It should be 500, unless some other scene changed it for you prior to loading the blank scene. Let's set it to 200ms.

```
M 200
```

That trigger output should be blinking faster. Notice that the M icon is lit up in *LIVE* mode. Let's disable the metronome script:

```
M.ACT 0
```

Stopped. Now turn it back on:

```
M.ACT 1
```

We can give `M` a long time, up to 32 seconds. We can also hard-reset the counting of the metronome:

```
M.RESET
```

This will reset the count-down before the next execution of the M script.

Like the script 1-8 hot keys, you can manually execute the M script with **WIN-M**, even if the script is disabled with `M.ACT`.

The metronome script is exactly the same as the other scripts– it's simply triggered internally on a clock.

Happy, predictable startups

Scenes only store script and pattern data– the current state of variables (such as `M`) and CV parameters are unchanged. On hardware startup these will be initialized to 0, but when changing scenes whatever values were in memory will remain.

The Initialization script will help make scene startups predictable. Get into *EDIT* mode and navigate to the I script. This is executed on scene recall. You can manually execute it from the keyboard with **WIN-I**.

Things that normally go in the `I` script are things like:

```
M 700
CV.SLEW 1 500
```

Metro timing, CV slews, and other variables. Perhaps you want to start with particular TR output values or CV levels. Put them here.

EXAMPLE: COPYCAT FLIPS

This scene is featured in the banner video above.

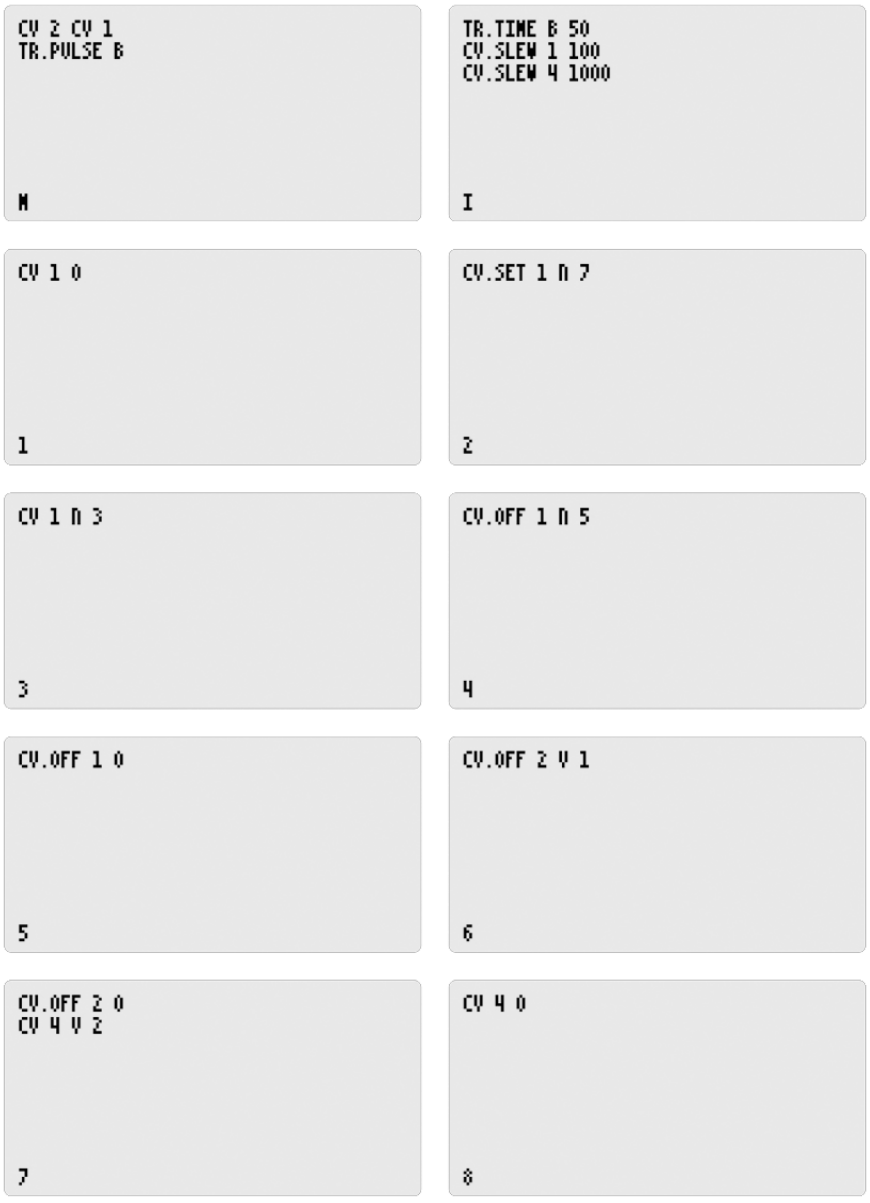
Use two oscillators– tune them to the same low note. Connect CV output 1 to the frequency input of one oscillator, and CV output 2 to the frequency input of the other. Use TR output B as a gate for the second oscillator. Hook CV 4 up to something in the sound path that changes timbre, like a wave folder or filter.

Inputs 1-3 change the root note of CV 1. Inputs 4-5 change the offset of CV 1. Inputs 6-8 change the offset of CV 2 (octaves) and modulate CV 4.

The metro script is running constantly, triggering the pulse and "sampling" CV 1, mirroring this to CV 2. It's similar to a sample and hold. By modulating the "drone" root note of CV 1, you'll have a following pulse stream with the other oscillator. As described above, modulating offsets doesn't change this CV mirroring, so we get different chordal combinations and progressions.

The init script makes sure the pulse width and slew times are all set up.

In the video above I'm driving some inputs with a step sequencer (White Whale) and others with A cyclic trigger generator (Meadowphysics). The first is more regular, while the second has some randomness in the pattern. The two overlapping paired with the configuration of the simple script create some nice musical shifts.



Suggested explorations

- Change slew times dynamically with script triggering for interjection

Reference

Commands

```
V x             lookup volt value x (0-10)
VV x            lookup precision volt value x (0-1000, for 0.00 to 10.00 volts)
N x             lookup note value x (0-127)
CV x y          set CV output x to y
CV.OFF x y      set CV offset x to y
CV.SLEW x y     set CV slew x to y
CV.SET x y      set CV x to y, bypass slew

M x             set metro script clock time to x
M.ACT x         enable/disable metro script clock (0/1)
M.RESET         reset counter for metro script
```

Part 3: Playing With Numbers

How to talk COMPUTER

We've made it to part three without *really* explaining how commands work– I appreciate your blind trust. But now it is time to talk about *syntax*. This sounds possibly boring but it's the key to understanding the great potential of the system, allowing us to make interesting and sophisticated interactions.

Before we begin, load up a new blank scene. We've described doing this in past studies (hint: hit **ESC**, navigate with] until you find a blank scene, hit **ENTER**).

Right to Left

In many ways Teletype is just a fancy calculator. Syntax follows prefix notation (aka Polish notation)– the *operator* is to the left of the *operands*. Get into *LIVE* mode and try this:

```
ADD 2 8
```

You'll see the result printed: 10. Here the *operator* is `ADD`, the *operands* (aka arguments) being `2` and `8`. `ADD` wants two arguments (values). And why did it print? Teletype is designed to display the returned value, if one is returned.

`ADD` returns a value. Let's use that fact to make a longer command:

```
ADD 2 ADD 4 8
```

What? It looks weird but it'll feel better soon. Read *right to left*– find the rightmost operator and give it arguments. Pulled apart, first we have

```
ADD 4 8
```

which equals 12. Let's mentally substitute that value into the remaining command:

```
ADD 2 (12)
```

which makes 14. Which should've been displayed as the result. Simple enough? Yeah! But things can get confusing:

```
ADD ADD 1 2 3
```

Yes, this works. Think of it like this:

```
ADD (ADD 1 2) 3
```

The rightmost operator uses the two closest arguments. Even this is legit:

```
ADD MUL 2 ADD 7 8 9
```

This decodes to `ADD (MUL 2 (ADD 7 8)) 9` which is 39. Perhaps not the friendliest command at first glance, but not a monster after a closer look.

Break it down, save it for later

You'd possibly rather make sounds than deciphering strange looking strings of text. We can make commands easier to read by using *variables*.

```
X ADD 1 2
```

Instead of printing the value 3, `X` is now assigned 3. Type `X` alone on the command line and it'll print its value.

KNOW THIS! → A variable (such as `X`) or parameter (such as `CV 1`) gets *set* if it is the leftmost *WORD* of a command. Otherwise the *value* is read and returned.

Consider that complicated command we had earlier: `ADD ADD 1 2 3`

Here's a more readable way:

```
X ADD 1 2
ADD X 3
```

`X` is set to the result of `ADD 1 2`. Then `X` gets read and returns 3 in the second line. The first line is effectively substituted into the second, breaking apart the command into multiple lines. Sometimes this is helpful– but you'll also quickly run out of lines in a script. So there's a balance to be found.

Real talk

- Operators accept arguments.
- Operators typically return values.
- An argument is a value, so you can feed the returned value of an operator into another operator.
- Variables can be assigned values, and read as arguments.
- Parameters (CV, etc) can be read and used as arguments.

The takeaway: numbers are interchangeable. Once the *flow* of numbers makes sense, you'll be able to put commands together in different ways to achieve a wide range of musical goals.

Elementary

Basic arithmetic operators are `ADD`, `SUB`, `MUL`, `DIV`, and `MOD`. These all take two arguments.

Addition and multiplication are commutative– the order of the arguments don't matter: `ADD 1 3` and `ADD 3 1` are the same. But this is not the case for the others:

- `SUB 4 2` reads *4 - 2*
- `DIV 8 4` reads *8 / 4*
- `MOD 7 2` reads *7 % 2*

Reversing the arguments will mess with your calculation. And what the heck is *mod*? It'll give you the remainder after a division. So `MOD 7 2` is 1.

Short term memory

Variables are good for much more than simplifying commands. They store values which can then be manipulated in various ways.

Available variables: `X`, `Y`, `Z`, and `T`. `T` is typically used for time-based operations but can be used freely.

Variables can be both set and read in the same command. Consider this:

```
X 10
X ADD X 1
```

First `X` gets set to 10. In the second line the `ADD` reads `X` and adds 1, returning 11. So `X` gets set to 11. Push the up arrow to re-execute that last line. `X` is counting up by 1 each time.

Variables are global– they keep their value across scripts. For example `X` can be changed by script 1 and then read and further manipulated by script 2.

```
1:
X 0

2:
X ADD X 1

3:
Y N MUL X 2
CV 1 Y
```

Here script 1 resets `X` to 0. Script 2 increments `X` by 1. Script 3 assigns CV output 1 to the note `X` multiplied by 2, using `Y` as a temp variable to break up the command (this could be easily combined, of course).

With various staggered triggering of these three scripts, you will likely find music!

More is more

You may feel that four variables just isn't enough– c'mon, a four-note melody? (Enough for me most days.) Fortunately there is a whole different system for saving tons of numbers: *patterns*. We'll cover this in a very-near-future part of this series.

In the meantime, if you're really desperate for more variables– `A`, `B`, `C`, and `D` can be overwritten. By default these are initialized to 1-4 on startup.

As it happens, `TR A` it is exactly the same as `TR 1`. So if you overwrite `A`, be sure to use `TR 1` instead.

Save save save

Be warned that variables are not stored with scenes. You can load a new scene and the variables will remain the same. If you want to have a scene recall with specific variable values you'll need to use the `INIT` script. For example:

```
I:
X 8
Y 22
```

Now we have some defined values for `X` and `Y` when the scene is loaded.

Lastly, variables aren't saved when powering down. On power-up memory is cleared, but the *INIT* script is called on powerup which allows you to define startup values.

Here at the post office

A long time ago at the post office in Castaic, CA: the postmaster asked the lady ahead of us "How fast do you want this to ship?" There was a lot of confusion and shrugging and finally "I don't care." To which the postmaster responded resoundingly:

"Here at the post office we only deal in absolutes."

Teletype is not the post office. (OK, weird transition, sorry.)

```
1:
CV 1 N RAND 12
```

We've arrived at the moment you've been waiting for: random semitones streaming out your modular synth.

`RAND 12` will return a random number between 0 and 12.

Operators can have different numbers of arguments. The arithmetic operators so far have had two. `RAND` takes just one. This is important to remember when analyzing (and building) commands. Of course, the command sheet will help!

Let's make this immediately more musical by creating a whole tone scale:

```
CV 1 N MUL 2 RAND 12
```

Now we're creating a random number between 0 and 12, and multiply it by two. Recall that `N` does a note lookup for sending to CV outputs.

And what about this:

```
CV 1 N MUL X RAND Y
```

Yeah? Now we can manipulate the range (Y) and interval (X) from some other script!

Somewhere in between

```
RRAND 4 8
```

A random value from 4 to 8 (inclusive) will be returned. `RRAND` (range random) takes two arguments.

Recall that a `TR` index can be set with 1-4 rather than A-D? So we can turn on `TR A 1` with the identical command `TR 1 1`. But how about this:

```
TR RRAND 1 4 1
TR RRAND 1 4 0
```

The first line turns ON a random trigger output. The second line turns OFF a random trigger output. We can do the same with CV.

Infinite coins

`TOSS`

This operator has no arguments! It returns a 0 or 1, randomly.

`TR RRAND 1 4 TOSS`

This command sets a random TR output to a random state, on or off.

`CV 2 N MUL 5 TOSS`

Here we're multiplying 5 by either zero or one, which gives us either zero or 5.

The musical qualities of stumbling

`DRUNK`

Appropriately, `DRUNK` isn't quite normal. It's a variable, but it changes by 1, 0, or -1 each time you read it. But you can also reset its position:

`DRUNK 5`

However, next time you read the value:

`DRUNK`

You may get 4, 5, or 6.

`DRUNK` does not have boundaries, so you may need to constrain it within a range to keep it useful:

`CV 3 V MOD DRUNK 5`

Here we're creating single-volt steps between 0 and 4. You might get something like this:

0 → 1 → 0 → 0 → 4 → 3 → 3 → 4 → 3 → 2 → 1 → 1 → 2

The `MOD` operator wraps the edges.

EXAMPLE: VIKING

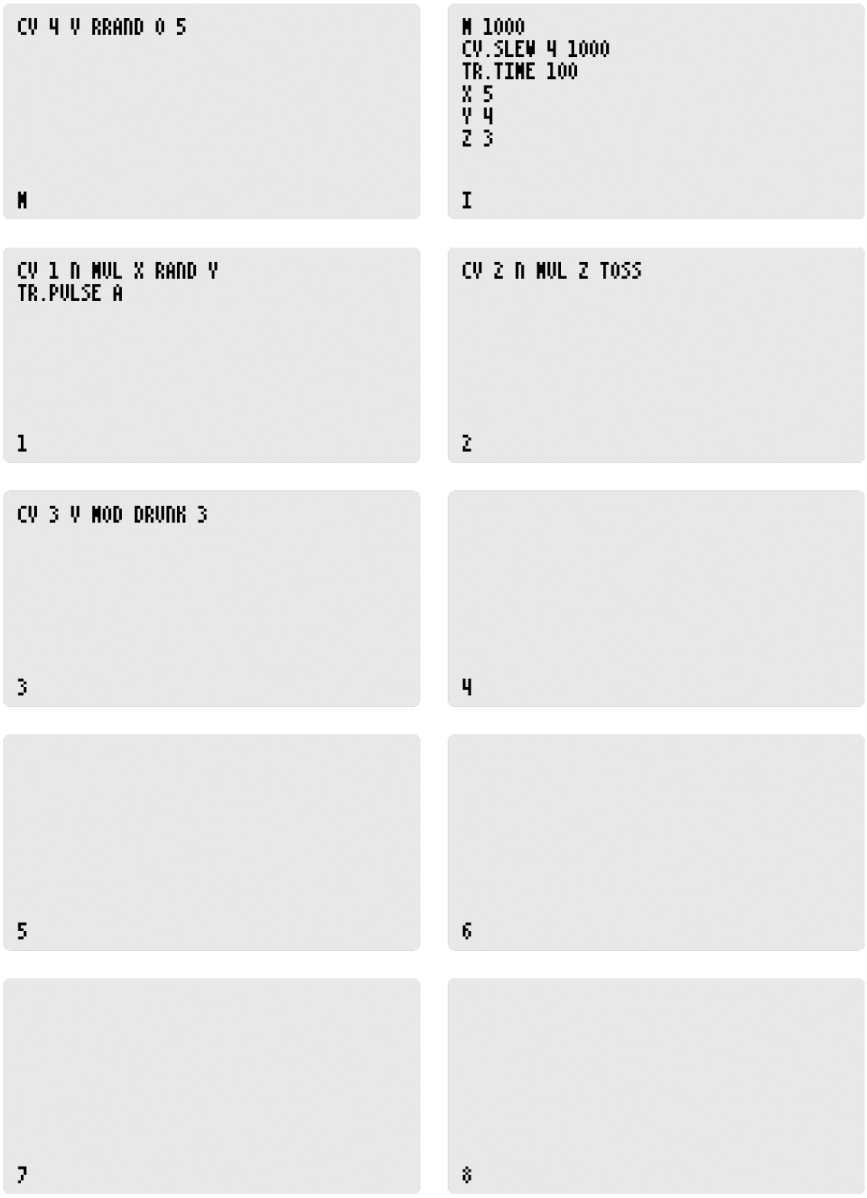
This scene is featured in the banner video above.

Like the previous study, we're using two oscillators with frequencies controlled by CV outputs 1 and 2. Remember to tune them to the same note prior to plugging in CV.

- Input 1 will randomly select a note for the first oscillator. TR out A will be pulsed.

- Input 2 will choose between two notes for the second oscillator. I used this as a low root note.
- Input 3 advances a drunk walk of single volts between 0 and 2.
- Metro running at 1 second interval, randomly slewing CV output 4 between random voltages 0 to 5.

Values are initialized on startup.



Suggested explorations

Get into *LIVE* mode and try changing some variables:

```
X 2
```

```
Y 12
Z 12
```

`X` changes the note spread and `Y` the range for voice 1. `Z` the interval of the low tone for voice 2.

Reference

Commands

```
ADD x y         add x + y
SUB x y         subtract x - y
MUL x y         multiply x * y
DIV x y         divide x / y
MOD x y         modulus x % y (return remainder)

RAND x          return random value from 0 and x
RRAND x y       return random value from x to y
TOSS            return a random value 0 or 1

X,Y,Z,T         variables
A,B,C,D         variables, initialized to 1-4 on startup
DRUNK           variable that changes by -1, 0, or 1 when read
DRUNK x         set DRUNK value to x
```

Part 4: Collect and Transform

One knob (feel all right)

By now you may suspect there are some rituals to be followed before the teleknowledge will rain down. Yes, I mean load a blank scene. (Hit **ESC**, navigate with] until you find a blank scene, hit **ENTER**). Then get into *LIVE* mode.

Read the knob value. Type:

```
PARAM
```

Turn left and right and repeat the command (remember **UP ARROW**). You'll quickly discover that the range is 0-16384, the same as how 0-10v is represented. This is convenient in some cases, but requires conversion in others. Let's start with the convenient. Get into *EDIT* mode:

```
M:
CV 1 PARAM
```

Yes, that's it. Now you have a sample and hold. The sampling interval is the interval of the metronome. For example, change the metronome time to 100ms:

```
M 100
```

Instead of reading the knob value, we can also read a CV input. The range is limited from 0-10v. It's protected, so don't worry about sending bipolar voltages. It can handle it.

```
IN
```

This reads the jack marked *in*, next to the *param* knob. The range is 0-16384 for 0-10v. Let's do a sample and hold that's triggered by an input:

```
1:
CV 1 IN
```

Each time input 1 is triggered, the voltage on *in* is read and CV output 1 is assigned this value.

TRY → Give CV 1 some slew. Yeah?

So smooth too smooth

Let's break up the continuum of numbers. Quantization is the procedure of constraining a number to a smaller set. Teletype has a quantization operator:

```
QT (input) (quantization)
```

Say we wanted to get only multiples of 1000 from `PARAM`:

```
QT PARAM 1000
```

Now when the the knob is read, we'll get values like this:

2000 → 3000 → 4000 → 4000 → 5000 → 6000 → 6000 → 7000

Here are some more useful variations for driving CV:

```
CV 1 QT PARAM N 1
CV 1 QT PARAM N 2
CV 1 QT PARAM DIV N 1 2
CV 1 QT PARAM V 1
```

The first line quantizes to semitones. Second line quantizes to a whole tone scale. Third line quantizes to quarter-tones (a semitone divided by two). Final line quantizes to whole volts (octaves).

`QT` will return the closest matching value to the interval. If the value is squarely in the middle, it will round up.

- `QT 40 10` → 40
- `QT 44 10` → 40

- `QT 47 10` → 50
- `QT 45 10` → 50

Shifting

Often we'll want to shrink the range of values we're using– this will come up commonly when using the *param* knob. Of course we could simply divide:

```
DIV PARAM 256
```

The result is a new range of 0-64.

Another way to scale down a value is with bit shifting. A shift right divides by powers of two:

- `RSH PARAM 1` → 0-8192
- `RSH PARAM 5` → 0-512
- `RSH PARAM 10` → 0-16

The first argument is the value to be shifted, the second how many places to shift. Shifting by 10 is equivalent of dividing by the 10th power of 2 is equivalent to dividing by 1024.

Shifting left multiplies rather than divides:

- `LSH 1 4` → 16

Consider this command:

```
CV 1 N LSH 1 X
```

We're outputting CV notes that are shifted according to `X`. If a script is modulating `X` from 0 to 5, we'll get exponential note values:

1 → 2 → 4 → 8 → 16 → 32

Setting boundaries

Sometimes you need to keep your numbers in line, to prevent them from wandering into unmusical territory. There are some authoritarian operators for this:

```
LIM DRUNK 4 10
```

The result will be from 4 to 10 despite where `DRUNK` might end up. `LIM` is saturating, so in this case if `DRUNK` is 1, we'll get 4.

```
LIM PARAM V 2 V 8
```

This will limit the *param* knob to 2-8 volts, creating "dead zones" at the top and bottom of the range.

If we're only interested in creating an upper or lower limit and not both, we can use `MIN` and `MAX`:

- `MAX 2 8` → 8
- `MIN 2 8` → 2
- `MIN 8 2` → 2

`MAX` returns the greater of two arguments. `MIN` returns the lesser. It follows that the argument order doesn't matter, 'cause greater is greater, mate.

We can also limit without saturating at the edges: wrap around a range:

```
WRAP PARAM 200 500
```

A value within the range 200 to 500 will always be returned. When `PARAM` goes above 500, it will wrap to 200 and keep climbing until it wraps again. (Aside: you could do this same thing with `ADD 200 MOD PARAM 300`.)

This will make a strange-feeling knob:

```
CV 1 WRAP PARAM 0 V 1
```

And this too:

```
X WRAP RSH PARAM 9 0 4
CV 1 N LSH 1 X
```

TICK TICK TICK

Teletype has a readable internal timer. Read the timer in *LIVE* mode:

```
TIME
```

The timer counts in milliseconds. As it lives within the Teletype datatype range, it rolls over at 32 seconds, and starts counting up from -32. This is a little weird. To set/reset the time:

```
TIME 0
```

You can set TIME to any value, not simply zero.

The timer can be started and stopped:

```
TIME.ACT 0
```

`TIME` will now halt and no longer be incremented.

```
TIME.ACT 1
```

The timer is now re-enabled. With `TIME.ACT` you can implement a start/stop mechanism similar to a stop-watch.

While variable `T` can be used for generally, it's helpful for code readability to use `T` when manipulating time-oriented numbers. For example:

```
1:
T TIME
TIME 0
M T
```

This script measures the time interval between triggers to input 1, and then assigns this interval to the metro script. Add trigger pulses to both script 1 and the metro script for an interesting synchronized but not phase-correct pulse machine.

With `TIME`, `PARAM`, and `IN` values you'll often want to smooth them out a bit. You can do a two-stage fundamental "smoother" by simply averaging. We do have an average operator:

```
AVG 3 6
```

This will return the average of 3 and 6, which is (3 + 6) / 2 = 4. So to create a simple smoother:

```
1:
M AVG T TIME
T TIME
TIME 0
```

Now the metro time is getting assigned a smoothed value. `T` is the previously read `TIME` value, so the order of these commands is important to have the intended effect.

EXAMPLE: SLOW READER

This scene is featured in the banner video above.

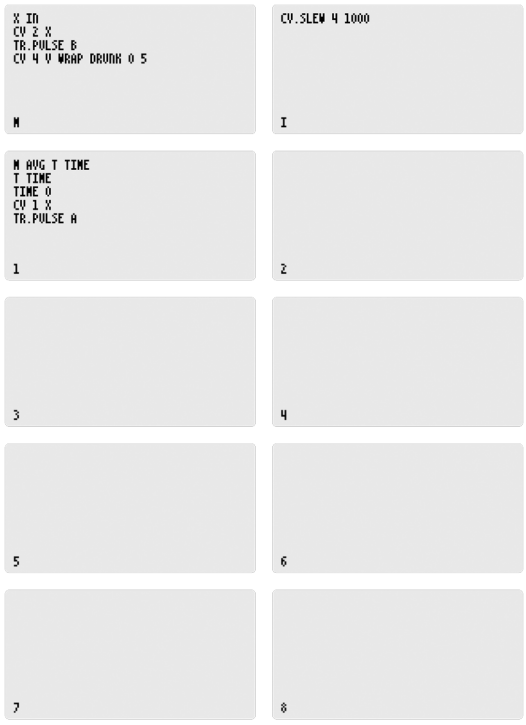
A clock input has its time interval measured. This interval is smoothed with the previous reading and then applied to the metronome script. Triggers are generated both for the metro and incoming clock. The effect is a sort of lagging synchronization where the phase likely does not align, creating interesting rhythms when the incoming clock is modulated.

On each metronome pulse the CV in jack is sampled and assigned to CV output 2. When the next clock to trigger input 1 arrives CV output 1 is also assigned this value, creating a following motion in most situations.

In the video above Earthsea is generating a melody, serving as the CV input. White Whale is being heavily underutilized simply as a clock out.

- 1: Clock input
- in: CV input

CV/TR output paired for voices 1 and 2. CV 4 outputs a smoothed drunk 0-5v, changed on the metro interval.



Reference

Commands

```
PARAM           read param knob
IN              read input CV jack

QT x y          quantize input x to nearest value y

RSH x y         shift x by y bits to the right
LSH x y         shift x by y bits to the left

LIM x y z       limit x to range y to z
MIN x y         return lesser of x and y
MAX x y         return greater of x and y

WRAP x y z      wrap input x to range y to z

TIME            read time interval in milliseconds
TIME x          set current time
```

```
TIME.ACT x      set timer active/inactive (1 or 0)

AVG x y         average x and y
```

Part 5: Pattern Tracking

Making lists

There comes a point when you'll want more numbers. Not just a few variables– you'll need bags of numbers, heaps of numbers, rivers of numbers.

Teletype has *patterns* which normal programming languages call arrays. You can think of a pattern as a fixed-length list of numbers.

- Each pattern has 64 indexes (positions).
- There are four patterns.
- Patterns store the same data type as the rest of Teletype: -32384 to 32384.
- Patterns are stored with the scene.

OK, why do we need heaps of numbers? There is no specific reason, but you can do quite a lot with such a list. The most obvious (and fun) is to make a sequence of notes. Teletype makes sequencers interesting because you don't need to follow a strict design– you can decide how the steps are recorded, played, looped, etc. But you can use patterns for various other purposes– times, custom scales, etc– they're just numbers. It's up to us as composer-performer-programmers to come up with a way to make numbers sound like music.

That last point of that earlier list is important. When you edit patterns and then save the scene, everything about the pattern will be stored alongside the scene scripts. So if you compose a melody using a pattern, it will be there when you next load the scene.

```
P 3
```

This reads index 3 of a pattern. Which pattern? We specify the "working" pattern with:

```
P.N 0
```

Here we set the working pattern to 0. There are four patterns, so this value can be 0-3. Let's store the number 10 to index 0:

```
P 0 10
```

Notice that we've just passed two arguments to the `P` operator to do a write. We "read" with just one argument:

```
CV 1 N P 0
```

Here we read index 0 of the working pattern, look up the note value, and assign that to CV output 1. You can check the working pattern by simply reading `P.N`.

If you want full random-access to pattern information and don't want to have to worry about the working pattern, use:

```
PN 2 4 100
PN 0 0
```

The first line writes 100 to pattern 2, index 4. The second line reads pattern 0, index 0.

Assistance for your list management

There are some operators that can do some smart tricks for you (well, "smart" for mid-last-century, but still cool). These operators do different things to the working pattern:

```
P.N 0
P.INS 0 5
```

Here we explicitly set our working pattern to 0, then insert the number 5 at index 0. This different than simply writing the number 5 to index 0, as first all pattern data is shifted back. For example, if the pattern 0 is:

3, 4, 8, 0...

And then we `P.INS 0 5` the pattern will become:

5, 3, 4, 8, 0...

We have a similar operator for removing a number, where the latter numbers are brought forward:

```
P.RM 1
```

This removes the number at index 1 (the number 3) which will result in:

5, 4, 8, 0...

Note that we do not need to constantly re-specify `P.N 0` for the working pattern– we set it once and then it will be remembered.

`P.INS` and `P.RM` do another helpful function– keep track of the *length* of the pattern. A pattern can be up to 64 elements long, but often we'll only want to use a fraction of this amount.

`P.L` reads and sets the length. Upon initialization each pattern has a length of zero. Changing numbers with `P` or `PN` does not affect the length. `P.INS` however increases the length by 1. `P.RM` subtracts 1 from the length.

Why do we care about the length? It's a helpful way of keeping track of what data in the pattern is being used, and what is either blank or garbage. `P.L` gives

you a way to keep track of what we're doing, and gives us some other ways to add and remove numbers:

```
P.PUSH 11
```

This will put the number 11 at the *end* of the pattern, and increase length by 1. So if we were continuing on with the pattern above (assuming `P.L` is 4), we'd have:

5, 4, 8, 0, 11...

The compliment to *push* is *pop*:

```
P.POP
```

This returns last element of the pattern and shortens the length by 1. So in this case, we'd get the number 11.

Read head antics

Using the working pattern and its length attribute we have a series of operators to facilitate creative reading:

```
P.I
```

This is the *read head* for the working pattern. Each pattern stores its own read position independently. You can read and write this position:

```
P.I 2
```

This command moves the read head to index 2.

```
P.HERE
```

The number at the read head is returned. (In this case, index 2, which was the number 8 in our pattern above). We can also write the the read head position:

```
P.HERE 0
```

This would set the number at the read head to zero.

THIS → `P.I` is the *index* (position) of the read head, while `P.HERE` is the *number* at the index.

```
P.NEXT
```

This adds 1 to the play head and returns the number at that position. So if we had:

2, 5, 8, 1, 7...

```
P.L 5
P.I 3
CV 1 N P.NEXT
```

The length is 5 and play head set to 3. The last line advances the play head to 4 and returns the number there, which would be 7 (we then convert it to a note value and send it to CV output 1).

What happens if we call `P.NEXT` again, but we're at the end of the list?

`P.WRAP` determines if we wrap to the start/end: 1 = yes, 0 = no. Each pattern has its own wrap attribute. So if wrap is on, `P.NEXT` will move the read head to zero and the first number of the pattern will be returned. If wrap is off, the play head will not advance, but you'll still get the final number returned. To turn on wrap:

```
P.WRAP 1
```

We can also play backwards in the pattern with `P.PREV`. Like `P.NEXT` it will respect the length and wrap of a pattern.

```
P.PREV
```

There are two other attributes for pattern playback– start and end. These are useful for sub-looping *inside* the length of a pattern. They govern the behavior of `P.NEXT` and `P.PREV`. Say for example:

2, 5, 8, 1, 7...

```
P.L 5
P.WRAP 1
P.START 1
P.END 3
```

If we call `P.PREV` repetetively, we'll get:

5 → 1 → 8 → 5 → 1 → 8 → 5 → 1 → 8 ...

This is how we make weird sequencers. Complete, algorithmic control over all of the playback and data manipulation of big lists of numbers.

Show me something

There's been a lot of imagining so far of what these lists look like. I have something to show you. Hit the **tilde** key, next to **1**, under **ESC**.

Why hello, you're now in *TRACKER* mode.

We could've called this spreadsheet mode, but back in the day we all agreed that musical spreadsheets would be called trackers. They are a fascinating way to create music– I grew up with Scream Tracker on an old DOS machine, though the very-modern Renoise is alive and well.

Teletype's tracker only slightly resembles these heavily-featured software packages, though. It excels (pun) at giving an interface for rapid viewing and editing of pattern data.

Navigation

The left column shows the index number. A scrollbar indicates your relative position in the editor. The highlighted number is your current edit position.

- **ARROWS** move the edit position.
- **ALT-LEFT** and **ALT-RIGHT** jump to the top and bottom.
- **ALT-UP** and **ALT-DOWN** jump up and down one page at a time.

Editing

- **DIGITS** typed in will modify the value.
- **BACKSPACE** remove rightmost digit.
- **-** flip sign.
- Brackets [and] nudge the value down or up by 1.
- **SPACE** toggle non-zero to zero, or zero to one.
- **ENTER** (if editing position one past length) will increment length and edit position.
- **SHIFT-ENTER** duplicate value at current position and shift pattern forward.
- **SHIFT-BACKSPACE** erases a value, sets it to 0.
- **ALT-X**, **ALT-C**, **ALT-V** all work as cut-copy-paste.
- **SHIFT-ALT-V** insert-pastes a number, shifting data and length forward.

Param knob live input:

- **CTRL** will overwrite the current position with the knob value scaled 0-31 (useful for scrolling in note data).
- **CTRL+SHIFT** will overwrite the current position with the knob value scaled 0-1023 (useful for scrolling in voltage data).

Attributes

- **SHIFT-L** set LENGTH of current pattern to position
- **SHIFT-S** set loop START of current pattern to position
- **SHIFT-E** set loop END of current pattern to position
- **ALT-L** jump to LENGTH of current pattern
- **ALT-S** jump to loop START of current pattern
- **ALT-E** jump to loop END of current pattern

EXAMPLE: DOLPHIN AND SEAGULL

This scene is featured in the banner video above.

A single pattern is used as musical content. For the video we had this:

0, 4, 9, 10, 12, 15 19

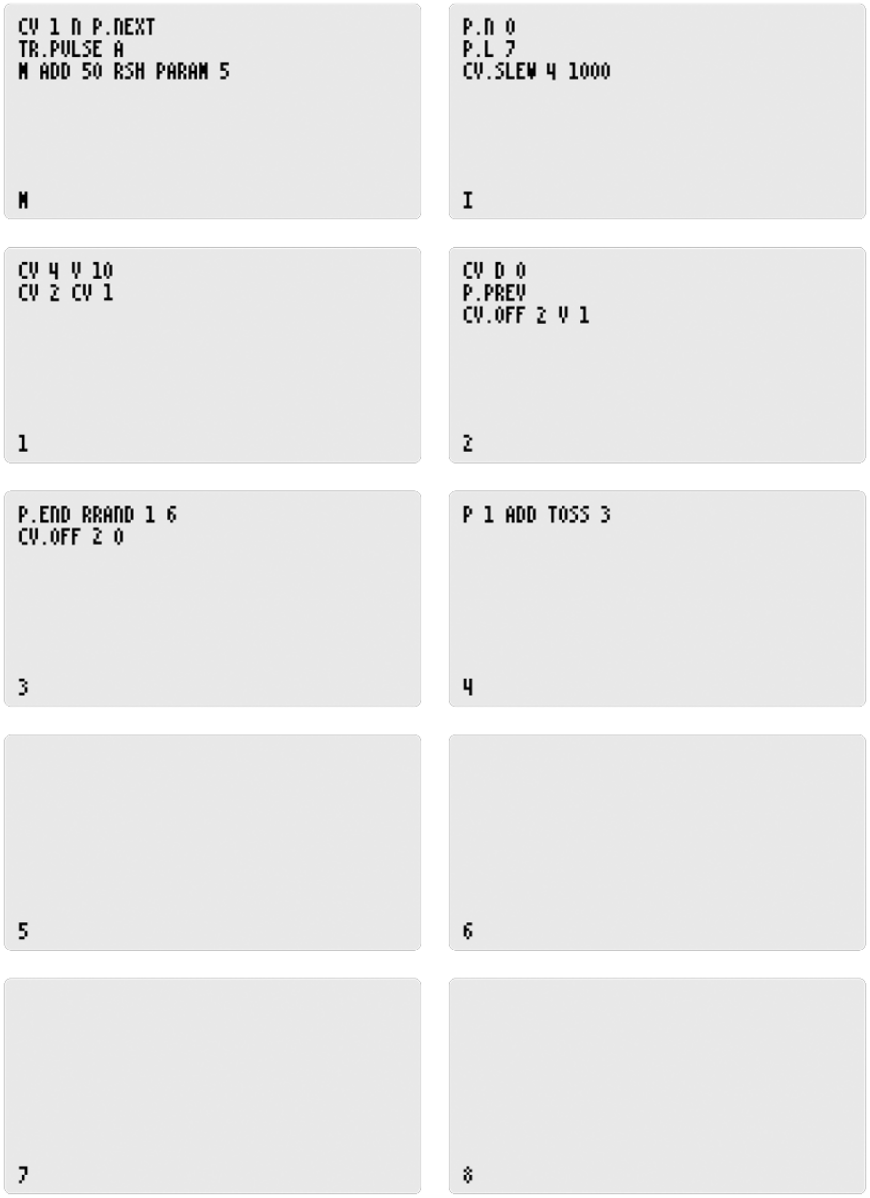
The metronome speed is controlled by the param knob, recalculated each time the metro script is executed. On each metro tick, we advance the pattern play head and set CV output 1. Trigger output 1 is pulsed.

- CV out 1: frequency of oscillator 1
- CV out 2: frequency of oscillator 2
- CV out 4: output level of oscillator 2
- Trigger out 1: output level of oscillator 1

CV 4 is initialized with a long slew.

If let run freely, the pattern will simply loop. Trigger inputs 1-4 for the following actions:

1. Ramp up CV 4 (level) and assign CV 2 to the current value of CV 1. This is basically a sample and hold.
2. Ramp down CV 4 and flip the octave of CV 2 up. Also interrupt the pattern playback by interjecting a P.PREV. It has a musical stalling/repeat effect.
3. Randomize the loop end of pattern playback. Set the octave of CV 2 back to normal.
4. Directly edit the pattern data, toggling between major and minor (3 or 4) on the second step.



In the video Teletype is being driven by an unsynchronized White Whale. This way the script triggers phase with the playback speed of the pattern– the result being the sample and hold between the two voices constantly shifts which tone is being copied to the second voice. The steps of sequence on the White Whale are in “choose” mode: one of several choices is made per step, so the action Teletype takes is variable per looped WW bar, yet regular enough to stay musical.

Reference

Commands

```
P x             read working pattern index x
P x y           write y to working pattern index x
P.N             read working pattern
P.N x           set working pattern to x (0-3)
PN x y          read pattern x index y
PN x y z        write z to pattern x index y

P.INS x y       insert y at index x of working pattern, increase length
P.RM x          remove element at index x of working pattern, reduce length
P.L             read working pattern length
P.L x           set working pattern length to x
P.PUSH x        add x to end of working pattern, increase length
P.POP           remove and return last element of working pattern, decrease length

P.I             read working pattern index (read head)
P.I x           set index for working pattern
P.HERE          read value at current index of working pattern
P.HERE x        write x to current index of working pattern
P.NEXT          advance index of working pattern and return value
P.PREV          rewind index of working pattern and return value

P.WRAP x        specify wrapping behavior for working pattern (1 = wrap, 0 = do not wrap)
P.START x       set loop start to x for working pattern
P.END x         set loop end to x for working pattern
```

Part 6: Maybe Later Remembering

Tell the truth

And what is truth anyway? Fortunately Teletype (and pretty much all digital systems) have a reductive view on the matter:

- 0 → FALSE
- EVERYTHING ELSE → TRUE

And why do we care about truth? Because sometimes we need to ask hard questions... like, is variable `X` greater than 2?

```
GT X 2
```

Teletype will tell you. If `X` is greater than 2, the value `1` (TRUE) will be returned. Otherwise you'll get `0` (FALSE).

Here's a list of relational operators that can let you test a condition:

```
EQ x y
NE x y
GT x y
LT x y
```

Represented are **EQ**uals *(x == y)*, **N**ot **E**quals *(x != y)*, **G**reater **T**han *(x > y)*, and **L**ess **T**han *(x < y)*. All return true or false, represented as 1 or 0.

And what shall we compare? How about we check if CV input `IN` is greater than 3V, and then output that as a gate on TR output A:

```
TR A GT IN V 3
```

There are some helper operators for comparing to zero:

```
EZ x
NZ x
```

These are **E**quals **Z**ero and **N**ot **Z**ero. Both take only one argument: the number to be compared to zero.

There ends up being a neat trick for flipping a variable between 0 and 1:

```
X EZ X
```

Here, when X is 0, it will become 1. And conversely, if 1, will become 0. Having this in a script will create a toggle. For example, you could use a trigger input to toggle the metronome:

```
M.ACT EZ M.ACT
```

The moment of decision

Now that we're full of truths and falsehoods, let's conditionally execute some commands. Introducing a new word: *PRE*. It's basically a short command that comes ahead of another command, separated by a **:** (colon). Here's a simple example:

```
IF X : TR.TOG A
```

The *PRE* `IF` takes one argument which is evaluated as true or false. If true, the remaining command after the separator (:) will be executed. So, according to our understanding of truth:

- if X is 0, nothing happens
- if X is anything but 0, TR output A gets toggled

We can expand the command within a *PRE* to become more complicated, however:

```
IF GT PARAM V 5 : CV 1 PARAM
```

Here knob input `PARAM` is read and compared to 5 volts. If greater, CV output 1 is assigned to `PARAM`, basically limiting the lower bound of the knob-to-cv-output assignment. (Note that this could more effectively be accomplished with `CV 1 LIM PARAM V 5 V 10`.)

We like randomness, right?

```
IF TOSS : P.NEXT
```

Here we maybe (50/50 chance) advance the pattern sequence.

```
IF LT RAND 100 75 : P.NEXT
```

Here we have a 75% chance of advancing the sequence. Given this is the sort of thing some of us like to do a lot, we created a simplified *PRE* just for probabilities:

```
PROB 75 : P.NEXT
```

This command is identical to the one above. The argument is a number from 0-100, representing chance to execute.

Followup

Sometimes the inquiry must continue.

```
IF X : TR 1 0
ELIF Y : TR 1 1
ELSE : TR 1 TOSS
```

When an `IF` gets a false argument (and the command is bypassed) we have the opportunity to match a new condition (with `ELIF`), or just have a fallback command (with `ELSE`). Let's trace this script given some assumed values of `X` and `Y`:

- X = 0, Y = 0 → TR 1 TOSS
- X = 0, Y = 1 → TR 1 1
- X = 1, Y = 1 → TR 1 0

Note that in the last condition (where X is 1) it simply doesn't matter what value is in `Y` as the `ELIF` and `ELSE` will be skipped. Similarly, if an `ELIF` is successfully executed, the following `ELSE` will not be executed.

You can stack up several `ELIF` statements in a row:

```
IF EQ X 0 : CV 1 V RAND 5
ELIF EQ X 1 : CV 1 0
ELIF EQ X 2 : CV 1 V RAND 10
ELIF EQ X 3 : CV 1 V 10
ELSE : TR.TOG 1
```

This script checks the variable `X` against the values 0-3 and has a "default" action if it isn't within range.

Note that you might run into command length issues while doing conditional statements. You may need to break up the condition command. This command is too long:

```
IF GT PARAM V 5 : CV 1 V RRAND 2 8
```

This is not:

```
X GT PARAM V 5
IF X : CV 1 V RRAND 2 8
```

Later, dudes

For your scheduling needs, there is a *PRE* for postponing a command:

```
DEL 250 : TR.TOG 1
```

`DEL` takes one argument: delay time in milliseconds. The command above will toggle TR output 1 after 250ms.

In *LIVE* mode the second icon (an upside-down U) will be lit up when there is a command in the `DEL` buffer. You can delay a command up to 16 seconds, and 8 commands can fit into the buffer.

You can clear the delay buffer (canceling the pending commands) with a single op:

```
DEL.CLR
```

A sort-of TODO list?

Teletype has a command stack which can lead to some musically interesting exploration. This might feel weird at first, but stick with us though it.

Say you want a command to execute later, not based on a time (where you'd use `DEL`) but rather by another command. We can achieve said desire thusly:

```
S : TR.TOG 1
```

Now the command `TR.TOG 1` is on the stack– it has not been executed. To execute the command:

```
S.POP
```

This executes the *most recently added* command on the stack. Think of the stack as a pile of donuts– the most recently added is the one on top. `S.POP` executes and removes the top command. If you're really hungry, though:

```
S.ALL
```

This executes the entire stack, which leaves it empty. Why is this interesting, though? Take the following two scripts:

```
1:  S : CV 1 N RAND 10

2:  S.ALL
    TR.PULSE A
```

Say script 2 was executing on a regular interval, and script 1 was somewhat random. Given we're using the stack, musical events get time quantized to only happen on the execution of script 2. This is a great method to achieve sync when desired.

The stack can hold 8 commands. If you try to add more, and the stack is full, the command will simply be thrown away. We can read the stack length (height):

```
S.L
```

This will return the number of commands waiting. We can also clear the stack:

```
S.CLR
```

Another interesting use of the stack is the "not sure what'll come out" method:

```
1:  S : TR.TOG A

2:  S : CV 1 V RAND 5

3:  S : TR.PULSE B

4:  S.POP
```

If we're triggering scripts 1-4 at irregular intervals, the stack will fill up with commands while script 4 is executing and removing a single command at a time.

EXAMPLE: PROBABLE SALVATION

This scene is featured in the banner video above.

A metronome triggers a pulse and executes the stack. Trigger inputs 1 and 3 add commands to the stack. All input scripts possibly modulate the tempo using a `PROB` *PRE*.

Script 2 conditionally changes CV output 4 based on the state of var `X`. `X` is toggled between 1 and 0 by script 3– creating some additional dynamic behavior between the two scripts.

In the banner video, all inputs are driven by a WW sequence that has a few trigger-choice steps. TT is driving the frequency of an oscillator from CV out 1.

CV out 2 controls an overtone timbre of this oscillator. CV out 4 controls the level of an additional sub oscillator. Trigger out 1 is connected to an envelope to spike the oscillator's level.

```
TR.PULSE A
S.ALL




M
```

```
TR.TIME A 30
CV.SLEW 4 100
CV.SLEW 2 200




I
```

```
S : CV 1 N MUL 5 RAND 4
PROB 10 : M 100
PROB 30 : TR.TOG 1




1
```

```
IF X : CV 4 V RAND 6
PROB 40 : M 150




2
```

```
X EZ X
S : CV 2 V MUL X 2
PROB 10 : M 200




3
```

```
PROB 50 : M 300
TR 1 0




4
```

```
5
```

```
6
```

```
7
```

```
8
```

Reference

Commands

```
EQ x y          x == y
NE x y          x != y
GT x y          x > y
LT x y          x < y
EZ x            x == 0
NZ x            x != 0

IF x : ..       execute command if x is true
ELIF x : ..     execute command (else) if x is true
ELSE : ..       execute command after failed if

PROB x : ..     probability to execute command, 0-100

DEL x : ..      delay command by x milliseconds
DEL.CLR         clear in-process delays

S : ..          add command to stack
S.POP           execute and remove most recently added command
S.ALL           execute and remove all commands on stack
S.CLR           clear stack
S.L             read only, size of stack
```

Part 7: Incantations

Misfits

Teletype has a few bits that are somewhere between *OPS* and *VARS*. They perform functions, and they have memory.

`O`

`O` (the letter, not zero) resembles a normal variable. You can set it to a value. And then you can read it. But when you read it *again*, it will auto-increment. And each time you read it after that, it'll increase by 1.

```
O 2
O
O
O
```

Here we set `O` to 2. Then the following three reads are: **2** → **3** → **4**.

Furthermore, there is `Q` which implements a queue, which is similar to a shift register.

```
Q 2
```

Puts the number `2` into the queue. At startup the queue is only 1 element long, so it's exactly like a normal variable. However, let's change the length of the queue:

```
Q.N 2
```

Now when we read `Q`, we'll be reading the value we assigned two stages ago.

```
Q.N 2
Q 3
Q 4
Q
```

The final command `Q` returns `3`. The length of the queue can be dynamically changed and the contents will be preserved. As a bonus feature, we can get the average of the elements in the queue:

```
Q.N 3
Q 0
Q 25
Q 5
Q.AVG
```

The result is 10. (0 + 25 + 5) / 3

This can be used as a smoother.

```
Q PARAM
CV 1 Q.AVG
```

Try putting this in a metro script. Each time the script is executed the "history" is pushed along and the average is an overall smoothing of the input. Given a short queue length `Q.N 3` smoothing would be minor, or you can do extreme smoothing with `Q.N 16`. The queue can be up to 16 stages in length.

We do this all the time

Iteration is the repetition of a process. It sounds like this.

The *PRE* `L` (which somewhat mundanely stands for "loop") allows you to execute a single command many times, with access to a counter variable per execution. Ahhhh, what does that mean?

```
L 1 4 : P.PUSH 55
```

This is what happened:

```
P.PUSH 55
P.PUSH 55
P.PUSH 55
P.PUSH 55
```

L takes two arguments before the separator: a starting number and an ending number. This dictates how many times the command will execute, so in the case above P.PUSH 55 happens four times: 1, 2, 3, 4. (Recall that P.PUSH adds a value to the current pattern– so we just dumped a bunch of 55's into a pattern.)

What makes L more useful is that we have access to the counter, using the special variable I. Check this out:

```
L 1 4 : CV I 0
```

Which effectively translates to:

```
CV 1 0
CV 2 0
CV 3 0
CV 4 0
```

With this one fancy command we zeroed all CV outputs. How this works is the variable I gets updated with the loop count on each iteration. This ends up being particularly helpful for INIT scripts setting up defaults:

```
I:  L A B : TR.TIME I 20
    L 1 4 : CV.SLEW I MUL I 100
```

Recall that A-D are simply 1-4, so they work fine with a loop. And did you see that last trick? I can be used for more than indexing– here it gets multiplied, effectively translating to:

```
CV.SLEW 1 100
CV.SLEW 2 200
CV.SLEW 3 300
CV.SLEW 4 400
```

Note that you can also count backwards, for example:

```
L 4 1 : P.PUSH I
```

Which would push **4** $\rightarrow$ **3** $\rightarrow$ **2** $\rightarrow$ **1** onto the pattern.

Automaticity

I just learned that is really a word. Loops are additionally great for generating pattern content.

```
L 1 64 : P.PUSH RAND 1000
```

Jump over to *TRACKER* mode and you'll see a bunch of random numbers. Use `CV 1 VV P.NEXT` repeatedly and you'll have a 0-10V wiggly-something. If you want to re-randomize the pattern, you could re-execute the loop after first setting `P.L 0` (this is because `P.PUSH` adds after the pattern length). But you could also just overwrite everything directly this way:

```
L 0 63 : P I RAND 1000
```

Here we're using the `I` as the pattern index.

Microtonalicity

That, my friend, is not a word. Microtonality can be achieved using patterns and is most easily achieved using loops.Some scales are easy to implement. Let's check them out quickly before going to patterns.

Quarter tone

```
CV 1 DIV N X 2
```

Given note `X` we're simply dividing a semitone by two.

Eighth tone

```
CV 1 DIV N X 4
```

Basically the same as above. Extrapolate this to get even smaller semitone scales.

100 tones per octave

```
CV 1 VV X
```

Just a different way of thinking about volts.

5 tones per octave

```
CV 1 MUL X DIV V 1 5
```

We're just doing some multiplying and dividing at this point.

The reason these all work is because they're evenly spaced– the distance between each pitch is the same.

RELATED ASIDE → One trick we can do with these sorts of scales is make a quantizer. First, let's set our desired quantization to the variable `X`, in the case below a whole tone scale:

```
X N 2
```

And then process the `IN` jack and output it to `CV 1`:

```
CV 1 QT IN X
```

Try executing this script on a metro, or manually trigger it with new note-ons. And then change `X` while playing notes.

Tuning patterns

Let's start with an equal temperament scale as a foundation for making our new scale.

```
L 1 64 : PN 0 I N I
```

Check the tracker view. We've basically copied the `N` note table to a pattern 0. Recall that `PN` takes an extra argument to specify which pattern to read or write.

We can now treat pattern 0 as our note map, rather than `N`. For example with note value `X`:

```
CV 1 PN 0 X
```

Now we can go into pattern 0 and retune the table. This works well in tracker mode– use the bracket keys to change by small values.

Let's now copy one entire pattern 0 to pattern 1:

```
L 1 64 : PN 1 I PN 0 I
```

This looks a little funny, but it works!

Now with two "scales" in our pattern bank, we can easily (with a script) toggle between the two. Just use `Y` as a scale variable:

```
CV 1 PN Y X
```

We still have pattern 2 and 3 for actual note-sequence data. Let's put something musical in there:

```
P.N 3
P.PUSH 5
P.PUSH 0
P.PUSH 4
P.PUSH 0
P.PUSH 2
```

And now we can use the "sequence" in pattern 3 to play the "scale" of pattern 0:

```
CV 1 PN 0 P.NEXT
```

`P.NEXT` will read the active pattern specified by `P.N`. These note numbers get pushed through our custom scale.

The takeaway point is that patterns can be used for a variety of tasks. Note sequences, scales, timings, etc. They're just a bunch of numbers!

Hidden workings

Teletype has psychic powers to control other grid-based modules, namely White Whale, Meadowphysics, and Earthsea. (Though actually this requires an extra ribbon cable connected behind the modules).

With the `II` command you can remotely control parameters of the the trilogy modules. For example:

```
II WW.POS 5
```

With a White Whale connected, this command will cut to position 5 of the currently playing sequence. All `II` commands are simply a key (such as `WW.POS`) and a secondary argument. Here's the full list:

White Whale

```
WW.PRESET       recall preset
WW.POS          cut to position
WW.SYNC         cut to position, hard sync clock (if clocked internally)
WW.START        set loop start
WW.END          set loop end
WW.PMODE        set play mode (0: normal, 1: reverse, 2: drunk, 3: rand)
WW.PATTERN      change pattern
WW.QPATTERN     change pattern (queued) after current pattern ends
WW.MUTE1        mute trigger 1 (0 = on, 1 = mute)
WW.MUTE2        mute trigger 2 (0 = on, 1 = mute)
WW.MUTE3        mute trigger 3 (0 = on, 1 = mute)
WW.MUTE4        mute trigger 4 (0 = on, 1 = mute)
WW.MUTEA        mute cv A (0 = on, 1 = mute)
WW.MUTEB        mute cv B (0 = on, 1 = mute)
```

Meadowphysics

```
MP.PRESET       recall preset
MP.RESET        reset positions
MP.SYNC         reset positions & hard sync (if clocked internally)
MP.MUTE         mutes the output of a channel (1 - 8)
MP.UNMUTE       unmutes (enables) the output (1 - 8)
MP.FREEZE       freezes the advancement of a channel (1 - 8)
MP.UNFREEZE     unfreezes (enables) advancement of the channel (1 - 8)
```

Earthsea

```
ES.PRESET       recall preset
ES.MODE         set pattern clock mode (0 = normal, 1 = II clock)
ES.CLOCK        (if II clocked) next pattern event
ES.RESET        reset pattern to start (and start playing)
ES.PATTERN      set playing pattern
ES.TRANS        set transposition
```

```
ES.STOP         stop pattern playback
ES.TRIPLE       recall triple shape (1-4)
ES.MAGIC        magic shape (1: halfspeed, 2: doublespeed, 3: linearize)
```

One highly requested feature was external clocking of the Earthsea. Here's how it works:

- Record a pattern using the grid, as normal.
- Send TT command `II ES.MODE 1`
- Now the Earthsea is being clocked via TT.
- Use `II ES.CLOCK 1` to send a clock pulse, via a script, live, metro, etc.

Note that the Earthsea needs a clock event for both note-on and note-off, so you will likely need to double your clock speed. Another possibility is to use a `DEL` to always send two `ES.CLOCK` messages:

```
1:  II ES.CLOCK 1
    DEL 100 : II ES.CLOCK 1
```

This method has a couple issues. First, note length is always `100`. If you trigger script 1 faster than 100ms, you'll get a weird phase problem and the note on/offs will get unsync'd. There are various other ways of approaching this issue that may require slightly more logic.

Sudden change of direction

Lastly (yes, we're at the end) you can load an entire scene from a command:

```
SCENE 4
```

This will load scene 4. You can check the current scene number simply by reading `SCENE`.

So, you can quite simply make a scene-advance script thusly:

```
SCENE ADD SCENE 1
```

This gets pretty close to the mythic patch-recallability feature much sought in modular. Teletype won't move the wires for you, however.

EXAMPLE: LABYRINTH

This scene is featured in the banner video above.

There are two completely separate processes running in this scene. The first is a 5 step sequence which is gets dynamically written by various triggers:

- 1: Step forward in sequence, note on CV output 1
- 2: Reset pattern to 5-semitone spaced scale
- 3: Reset pattern to 7-semitone spaced scale

- 4: Module each step of pattern randomly by 0-2
- 5: Transpose each step of the pattern up by 2 semitones

In the video, the White Whale (controlled by the grid) is driving this section. The clock out advances the sequence, and trigger steps on the grid trigger the pattern-modifying scripts. So this is a sort of meta-sequence which is often changing, where curious emergent repetitions often arise.

The second section is an input smoother, which in the video controls a filter sweep. A simple footswitch output (full scale) is connected to the IN jack of the Teletype. The metro script reads this value into a Q buffer (which is set very long by the init script), and then the Q.AVG is sent to CV output 2. We've basically scripted a rudimentary slope limiter in just a few lines of code.

```
0 RSH IN 4
CV 2 VV 0.AVG




M
```

```
M 50
CV.SLEW 2 100
0.N 16200




I
```

```
CV 1 N P.NEXT
IF MOD P.I 2 : TR.PULSE 1




1
```

```
L 0 4 : PN 0 I MUL I 5




2
```

```
L 0 4 : PN 0 I MUL I 7




3
```

```
L 0 4 : P I ADD RAND 2 P I




4
```

```
L 0 4 : P I ADD 2 P I




5
```

```




6
```

```




7
```

```




8
```

Reference

Commands

```
O               like a normal variable, but auto-increments on each read
Q               read or add value to shift register
Q.N             read of set length of shift register
Q.AVG           return average of contents of shift register

L a b : ...     loop command with I assigned a to b per iteration

SCENE           load stored scene
```