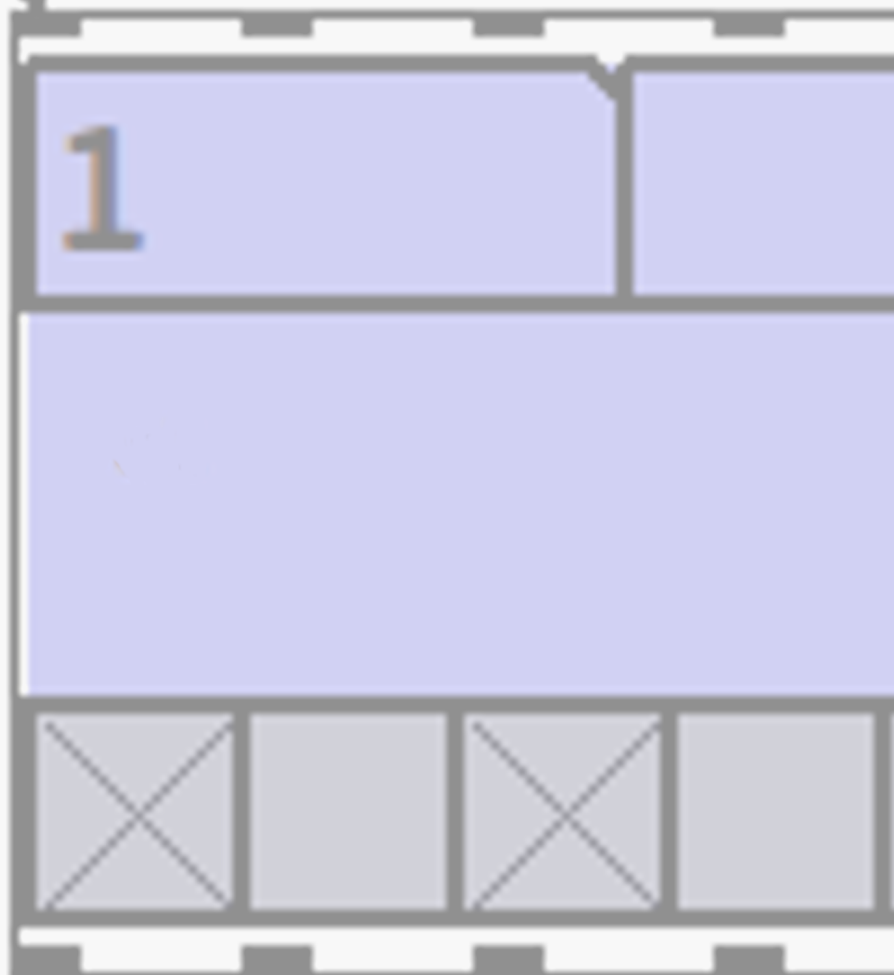# THE CONTEXT MANUAL

V 0.3.2

the user's guide to Context, an advanced sequencer for PD

Table of Contents

THE CONTEXT MANUAL

0.3.1

1.INTRODUCTION

WHAT IS CONTEXT?

Context is a powerful sequencer built in Pure Data (PD) which re-imagines music as a compositional network. Context incorporates traditional step sequencing and timeline playback into a small, simple GUI tool which can be replicated, modified and interconnected in the PD patch. Rather than forcing you to work within a pre-existing software suite, Context invites you to make your own sequencing environment, and to customize that environment according to your own taste and the demands of your composition.

Context is one tool a single tool ready for man functions. Among its features, Context is:

1. **A STEP SEQUENCER, A SAMPLE PLAYER, A DATABASE, A LOGIC GATE, AND AN EMBEDDABLE TIMELINE**. Don't be fooled by the minimalist interface—a context object can be any or all of these things at once.

2. **POWERFUL AND FEATURE RICH**. Context has over 100 parameters, making it infinitely adaptable and constantly surprising.

3. **EASY TO USE**. Context has a clutter-free layout which puts user experience first, making it accessible even for PD beginners. Most common tasks in context can be accomplished using the GUI; an advanced menu system takes care of the rest.

4. **A LANGUAGE**. Context has its own dedicated control language, giving the user text-based control over the network, and its behaviour. The language and the GUI are well integrated, giving you immediate control over any setting.

5. **MODULAR**. Context is designed to be replicated and interconnected to form compositional networks. As with modular synthesis, the user has a great deal of control over the nature and behaviour of the environment she builds.

6. **GENERATIVE**. Context can incorporate random processes at any level of composition, and context networks have the ability to modify themselves.

7. **ALGORITHMIC**. Compositions can proceed according to custom rules and formulas designed by the user.

8. **SAVABLE**. Context states saves straight into the .pd file, without ever requiring any special attention from the user.

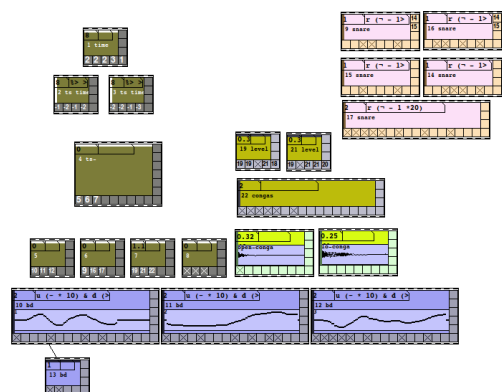*Illustration 1: a single context*

Illustration 2: a context network

WHAT IS THE PURPOSE OF CONTEXT?

Context is difficult to describe in a succinct way, but understanding the aims of the Context sequencer will perhaps help newcomers better understand what it is.

1. **Context is a single tool built for multiple purposes** and as such deviates from the normal sotware paradigm, which assigns certain tools to certain tasks. It might be more accurate to say that Context doesn't have *any* purpose; until the user finds one for it.

2. **Context provides an alternative to the traditional Digital Audio Workstation (DAW)**. Most DAWs are software environments which the user must conform to in order to compose music. This creates a learning barrier for beginners, and compatability problems for open plattforms such as PD. Context is not a software environment; rather, it is tool with which environments can be made. A Context network can hve much of the functionality of a DAW system, but is infinitely more adaptable to the user's needs.

3. **Context is both linear and non-linear**. Most sequencing software restricts you to a certain way of thinking about composition, either as notes on a score, traker arrangements, nodes on a network, stochastic formulas or and Markhov chains, etc. Context offers you all of these options but demands none of them; it can be used linearly, non-linearly, algorithmically, and in many other ways besides. Perhaps more importantly, a network can be partly linear and partly non-linear—in Context, these terms are not mutually exclusive.

IS CONTEXT RIGHT FOR YOU?

Context is designed to be appropriate both for beginners and advanced PD users and composers. The key to this is its modular design. A beginner can easily hook a single Context up to a synth that they are working on, letting Context assume control of sending messages while they focus on patch design and DSP. An advanced user or performer can make a network consisting of hundreds of Context objects, creating complex generative compositions on a whole array of custom PD istruments. Since Context sends PD messages, it can be made to interact seamlessly with any existing PD patch.

As a sequencing tool, Context deals with messages and events rather than sounds and signals. As such, it has no particular sound and can be used to create music in any genre or style. Context ships with a few basic synthesizers for demonstration purposes, but the role of designing or choosing instruments is reserved for the user.

If you want to use Context, it's best that you already know—or are willing to learn—some basics about PD and sound design. Context itself requires learning, but—unlike most other music software—the bare basics of Context are enough to make interesting and enjoyable music. Extensive documentation and help files are built into the program, so you can plan to learn as you go.

ABOUT THIS MANUAL

The Context Manual is part of the Context documentation project. It is meant as a beginner's guide, explaining the basic function of Context, how to use it, and providing a reference for terms. However, this manual is not:

1. A complete reference for all of context's functions. For that, refer to the file 'context_commands_all.pd';

2. A stylistic guide. Knowing how to use the software is one thing; knowing how to make music is another matter entirely. See the video tutorial series (forthcoming) for this;

Besides this manual, the other documentation resources currently available are:

1. **The `*-help.pd` files** for Context and other objects in the Context library. These generally give quick explanations and refer to relevant sections of the manual;

2. **Built in menu help prompts**. These are located on every page of the Context menu system (see Section 8: SENDING COMMANDS). They are meant as quick clarification and reference, and so might be less useful to first time users;

3. **Example files**. These are all stored in the 'helpfiles' folder and are refered to throughout this manual. It is recommended to study them as they come up.

4. **Video tutorials**. (forthcoming).

The sections of this manual are presented in order of importance. Most sections refer to material that has been covered in previous sections, but not subsequent ones. Cross references are made whenever possible to assist readers who are jumping to a particular section. Sections 3 to 10 cover the basics and are essential reading for anyone who wants to use Context seriously. Sections 11 to 20 cover specialized features, so it might make sense to skip them at first and return to them when required.

The terminology used throughout this documentation is consistent within itself, but might prove confusing to newcomers. A complete list of terms can be fount in Section 23: GLOSSARY OR TERMS, but one peculiarity should be highlighted here: Throughout this manual, "a Context" usually refers to a specific instance of the Context abstraction.

This manual uses the conventional text abreviations for depicting PD object.

• **[square brackets]** mean a PD object, ie. **[float]**.

2.INSTALLATION

Context is a PD abstraction and requires no installation as such. However, it is quite picky about PD versions and externals, so configuration can br tricky.

The requirements are:

- **OS compatibility**: Windows (tested) Linux (tested) or MAC (untested)

- **PD version**: Vanilla 0.47 or later

- **Externals**: Context requires the following externals:

    - zexy
    - cyclone
    - moocow
    - flatgui
    - list-abs
    - iemguts (v 0.2.1 or later)

PD installation is not covered in this manual; got to http://msp.ucsd.edu/software.html and http://puredata.info/ for information on this.

Once you have PD installed, go to *Help → Find Externals*. Search for and download each of the externals individually, making sure to get the right one one for your OS architecture.

Next, you need to declare the path to Context on your hard drive for PD to find it. Go to *Edit → Preferences → Path.* Select *New* and select the Context folder, so it looks like this:
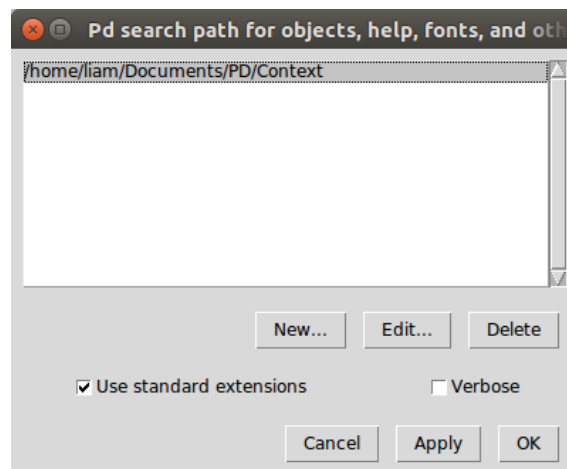
Illustration 3: declaring the path

Your system should now be ready to load Context. Open a new patch (Ctrl + N) and create a new object called `[context]`. If it loads without errors, you're good to go! If there are errors in the console window, it most likely means that one or more of the external libraries hasn't been found, so make sure that you are declaring the externals folder properly.[1]

➢ See Section 21: TROUBLESHOOTING if you run into problems trying to get Context to load.

---

[1] Advanced note: Context declares all its externals through the [library/object] method and doesn't contain any [declare] or [import] objects.

3. HOW DOES CONTEXT WORK?

SUMMARY

- Every Context object receives 'start' triggers and sends sequenced messages ('hits')
- The process of sending sequenced outputs is called the *Context cycle*.
- One Context can start another, creating feedback loops.

BASIC PROCESS

As a sequencer, Context's main job is to decide what happens when. To this end, Context stores musical events (ie. 'play C# ') and makes these events occur in a timed, orderly fashion. The 'musical events' that Context stores are actually PD messages, stored in a database.

The overall process of sending events out at the right time—that is, the process of sequencing—is referred to as the *Context cycle*, with an individual event often being called a *hit*.

The messages that context fires can be directed towards another PD patch, or they can be towards another context (typically to start it). This means that **context can control another PD patch at the same time as it controls itself**—a key principle behind the context sequencer.

➢ See example file '01_basic_process.pd' for a working demonstration of this basic process.
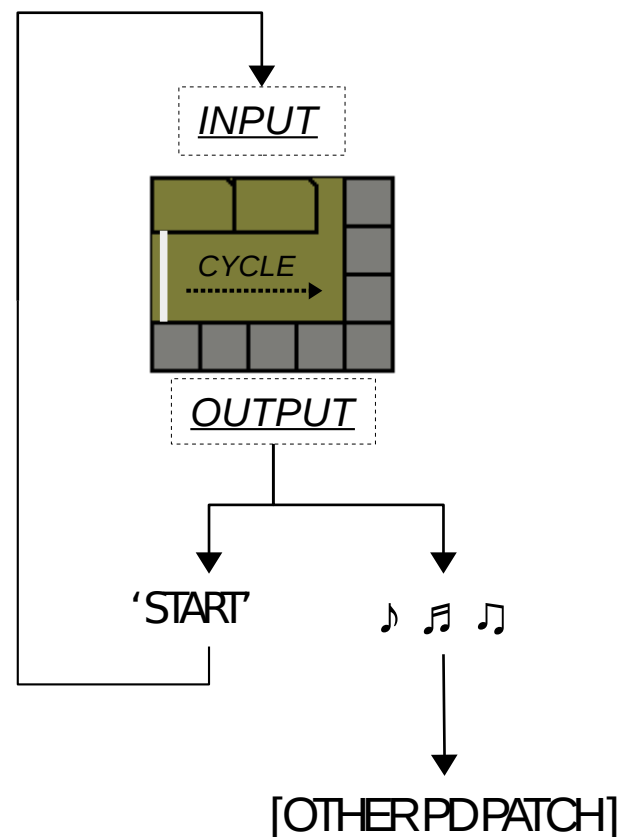
Illustration 4: the basic context process. See 22 COMPLETE SCHEMATIC OF CONTEXT PROCESS for a complete version of this diagram.

CYCLE

The 'cycle' mentioned in Illustration 4 is the process of starting various events as dicated by the toggles, spread out over a certain time, the *cycle time*. It is depicted by the cursor (the thin white line on the green background above) moving across the screen, usually from left to right. More information about the cycle can be found in Section 6: PATTERN AND BURST and Section 10: TIMING.

INPUT AND OUTPUT

There are two types of input for context: **inlets** and **receive channels**.

Likewise, there are two types of output: **outlets** and **send channels**.

The in/outlets are the familiar PD connection terminals found at the top and bottom of the object. The send & receive channels are "wireless" messages sent through [send] and [receive] objects inside the patch (the channels used, ie. the arguments for [send] and [receive], are discussed in Section 7: ID AND CHANNELS).

Both inlets and receive channels interpret incoming messages in the same way and are essentially identical to each other. Output is a little more complicated—see Section 5: CONNECTIONS.

FEEDBACK

As Illustration 4 suggests, feedback is at the heart of a Context network. Although it is not strictly necessary, most Context networks will employ feedback loops at some, perhaps many levels, and viewing networks as compositions implies thinking about feedback.

There are several types of feedback loops to consider:

1. **CLOSED LOOPS** occur when one Context feeds directly back into itself. These can be used to form a regular pattern that will repeat continuously until the system is interrupted. Closed loops are useful for creating regularity and for "powering" other Contexts, but they can be dangerous if the loop is too small or receives too many hits (see Section 10: SAFETY).

2. **OPEN LOOPS** are feedback paths that might or might not be followed, depending on random decisions (see Section 11: BURST SETTINGS). They are useful for creating random patterns in a network, and to provide a "sink" so that feedback doesn't built up too much.

3. **EXTENDED LOOPS** occur when more than one Context is used to form the loop. They can be extended indefinitely.

➢ See example file '02_feedback.pd' for a working demonstration of different feedback types[2].

Note that a context may feed back into itself, as this diagram suggests, or it may feed into another context.

4. USING THE GUI

SUMMARY
- To start any Context, double click on it
- To stop, click and hold
- To resize, hold Shift and click and drag outside of the Context boundary
- To open the menu, right click and select *properties*

Before we go any further, it will be useful to know a few things about using the GUI. The most common tasks in Context are straightforward mouse and the keyboard combinations, but it is essential to know what they are, or else Context will seem very confusing.

STARTING AND STOPPING CONTEXT

Starting Context in the GUI is easy—simply double click anywhere on the green canvas area. The cursor will scroll across the screen to indicate the cycle (note that this won't appear to happen if the cycle time is 0, it will cross the screen instantaneously). Try this out on '02_feedback.pd' to get a feel for it.

---

2 See also '07_burst_and_structure.pd', but this refers to material covered in subsequent sections.

Stopping Context is similarly easy. While a Context is in cycle, click and hold on the canvas area.

BUG: Sometimes a Context won't recognize a double click. If this happens, click on the blank canvas area outside the Context and then try again. This will be fixed one day…

Besides the GUI method, there are a host of Context commands for different options of starting and stopping, including jumping to a middle point of the Context cycle. See Section 8: CONTEXT COMMANDS (INPUT LANGUAGE)) to learn about commands, and the file 'context_commands_all.pd' to see the full range of starting and stopping options.[3]

RESIZING CONTEXT

Resizing the context GUI is easy: just hold the Shift key and then click + drag in the area just to the left or bottom of the context. The axis will be resized and toggles created or destroyed accordingly. The selectable area is exactly one toggle-box width outside of the context GUI, roughly the space occupied by the curly brackets in diagram 5.
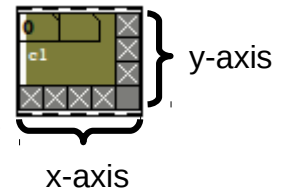
Illustration 5: context axes

- Single click and drag to increase / decrease the number of toggles in the array;

- Double click and dray to make Context bigger / smaller without changing the number of toggles;

- Use the Control key instead of the Shift key to change the cycle time along with the size.

➢ See 'context-help.pd' for more help with resizing.

ACCESSING THE MENU

Context has a menu system where you can change its settings quickly using only the mouse. To access the menu, simply right click on a Context and select *properties.* The menu will then pop-up, and you can scroll through the various options. Each menu page has a help prompt message at the bottom, where you can find out more about what the settings do.
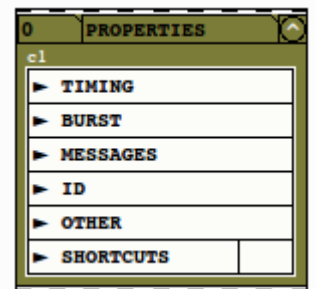
Illustration 6: the Context menu

- Click on the menu items to select or scroll

  ○ ► means a further menu;

  ○ ▲ means a command which expects input from the atoms above;

  ○ ■ means a command which does not expect input and will send as soon as you click.

- Click on the top-right hand button to return to the previous menu, or double click on it to return to the main Context view.

- If you select a setting which expects a numerical input, the float atom (top left) will be highlighted. Click and scroll or type into the atom to set the setting.

- If you select a setting which expects a text input, the symbol atom (top middle) will be highlighted. Click and type into the atom and press enter to set the setting.

See Section 8: CONTEXT COMMANDS (INPUT LANGUAGE) for more information about the menu and commands.

---

3 See also Section 17: MARKER.

OTHER GUI SHORTCUTS

There are various other GUI shortcuts which are covered at different places in this manual.

- **Toggle floats** see Section 5: ACCESSING FLOAT CONNECT;

- **The symbol atom** see Section 9: GUI ACCESS TO DATABASE;

- **The axis flip** see Section 6: FLIPPING THE AXES;

- **The memory bank** see Section 13: ACCESSING THE BANK;

- **The overlay** see Section 14: ACCESSING AND SAVING THE OVERLAY.

➢ See help file "GUI_manipulation.pd" for a complete list of GUI shortcuts.

5. CONNECTIONS

> ### SUMMARY
> - Contexts can be connected via cable connections or float connections
> - Float connections are accessed by clicking and scrolling on any toggle while the mouse is held
> - Float connections cancel sequenced messages

Making music with context typically means connecting many Context objects together to form a network. A Context receives commands (typically the 'start' command), and sends sequenced messages. Since messages can also be commands, Context networks become loops, where one context starts another, or starts itself, in a sequenced manner.

There are two main types of connection in Context: **CABLE CONNECTIONS** and **FLOAT CONNECTIONS**.

CABLE CONNECTIONS

Cable connections are regular PD connections, used to connect one object's outlet to another object's inlet.

In Context, all inlets are functionally the same, so it doesn't matter which inlet receives a connection.[4] Outlets, on the other hand, are distinguished from each other. Each outlet corresponds to one toggle from the x-axis toggle array (by default the Pattern array), and is time-synched to that toggle, so that it will send its output at a specific time in the Context cycle[5].

As a result of this design, we have a fundamental property:

forming a connection from outlet A on one context to any inlet on context B, a rule is established: *when outlet A fires, context B starts*. This is the main logic that drives context networks.

---

4 The reason why Context has multiple at all is largely aesthetic: a wide context looks tidier if its connections are spread across the top, rather than all forced into the left hand corner. However, it is possible to program special functions for different inlets using an overlay hack. This is discussed in Section 14: HACKING.

5 Unfortunately, the outlets don't quite line up with the toggles, so the correspondence isn't always easy to see. However, an outlet's corresponding toggle is always directly above it, and you will soon get used to matching them, even when they may not align perfectly.

➢ See file '03_connections.pd' to interact with this example (Illustration 7).
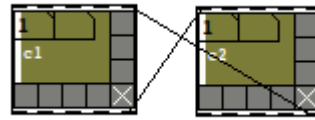
Illustration 7: two contexts forming a loop.

FLOAT CONNECTIONS

A large network can contain hundreds or thousands of connections, which can make a patch messy to the point of being unreadable. The solution to this problem is the **FLOAT CONNECTION**, akin to the [send] and [receive] objects which are used to make invisible connections.
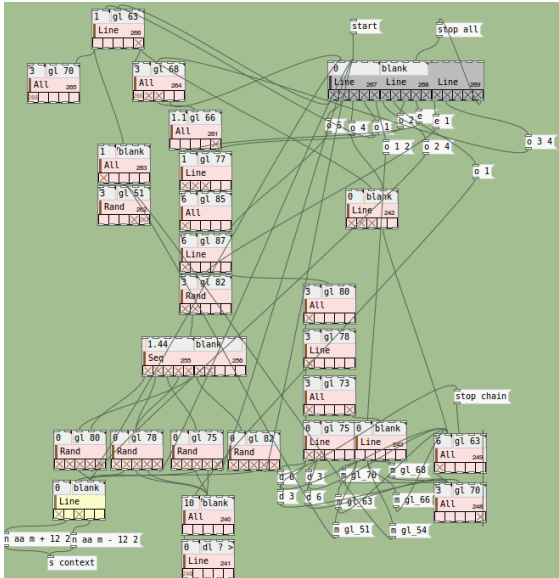
Illustration 8: an early prototype of context, without float connections. Not very tidy!

EXPLANATION OF FLOAT CONNECT

Besides its on-off state, every toggle in context holds one TOGGLE-FLOAT, an arbitrary number that is specified by the user. For instance, a toggle can hold the number 4 and be on, or it can hold the number 4 and be off. Positive toggle floats are used to form wireless connections, according to a context's ID number (see Section 7: ID AND CHANNELS). So if a toggle holding the number 4 fires, it will start another context with ID number 4.
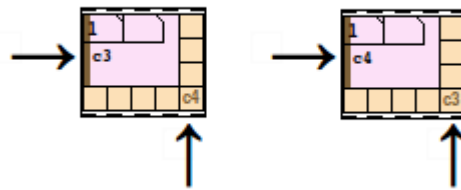
Illustration 9: two contexts float connected together to form a loop. Notice how the toggle-float numbers correspond to the ID numbers.

ACCESSING FLOAT CONNECT

Accessing toggle floats is easy: simply click and hold on any toggle and drag the mouse cursor up. You will see the numbers c1, c2, etc. scrolling up, corresponding to different ID numbers. Clicking once on the toggle turns its selection on of off, so the modified toggle still functions as a regular toggle. Dragging the toggle back down to 0 will turn the toggle float off and bring back the familiar x. In this way, you can form connections without cluttering the canvas.

Note that negative toggle-floats, accessed by scrolling the mouse down, do something quite different. Theeir function is outlined in Section Section 12: VARIABLES WITHOUT ARGUMENTS.

DIFFERENCE BETWEEN CABLE AND FLOAT CONNECTION

Besides the visual appearance, there is one other important difference between a cable and a float connection. When an ordinary toggle (ie. a toggle with no float, or float 0) fires, it sends a '`start`' command to the outlet *and* a sequenced message to the send channels. This means that its output is received by any context cable connected to it, and by any external [receive] object which is receiving sequenced messages and sending them onwards to other patches. When a float connect toggle fires, it sends a message only to the target context, but not to the send channel. **This means that float connect is only used to send commands within a network, not to outside objects.** The reason for this is very practical: it is a common requirement to start another context without sending a sequenced message. So if you want to trigger an

event within the network only, use float connects; but if you want to to trigger an event in and out of the network at the same time, use cable connections.

Another difference is that in PD, an object cannot be connected to itself. A float connection, on the other hand, can easily reference its own ID number, allowing a single Context to be set into a loop.

	CABLE CONNECT	FLOAT CONNECT
starts other context	YES	YES
sends sequenced message	YES	NO
allows self-connection	NO	YES

Table 1: comparison between cable- and float-connect

6. PATTERN AND BURST

SUMMARY

- The x-axis toggles are the Pattern, which fire stepwise during the Context cycle
- The y-axis toggles are the Burst, which fire simultaneously at the end of the cycle
- The Pattern and Burst can be flipped to assign Context outlets to the Burst
- The last toggle is a hybrid between the Pattern and the Burst

Every Context has two axes, the x-axis and the y-axis, which can be of any length. By default, the x-axis carries the PATTERN and the y-axis carries the BURST. The PATTERN is a familiar step sequencer which plays a given pattern in time with the context cycle. This is the most important part of context, because it is used to store and play back melodies and rhythms. Patterns can easily be changed by clicking on the toggles to select or deselect them.

The BURST series of toggles (on the x-axis) fire all at once at the end of the context cycle. These are used to take care of the logic of a Context network, deciding what happens after the context cycle is finished, for example which Context(s) should start when this one finished. The determination of exactly which bursts fire is covered in Section 11: BURST SETTINGS.

The double axis arrangement of the toggles presents a problem, as PD only allows for outlets on the bottom (x-axis), not the side (y-axis). How do you access the output from the burst toggles? There are two solutions to this problem. The first is to use float connections (see Section 5: CONNECTIONS), which are equally accessible on both axes. The second is to use the `:M` command which flips the axes. The y-axis now becomes the pattern and the x- the burst, thus assigning the bottom outlets to the burst. '02_feedback_types.pd' has examples of Context's with flipped axes.

FLIPPING THE AXES

To flip the x- and y-axes, either use the `:M` command, or hold the Shift key and double click on the float-atom. The cursor will move as confirmation.

➢ The distinction between the pattern and the burst is central to context, because it allows each context to behave temporally as well as logically. "Play this pattern, then do something else" — that is basic routine followed each context cycle. This allows a composer to simmultaneously develp a musical phrase and to decide that phrase's consequences. Deciding how much weight to put on the pattern or the burst is a matter of style. Different choices can lead to very different types of Context network.

THE LAST TOGGLE

Regardless of context's orientation (`:M`), the pattern and the burst meet at the bottom right toggle. This is referred to as the 'last toggle'. The last toggle behaves in a special way, partly like a burst, and partly like a pattern toggle.

The first thing to know about the last toggle is that it fires at the very end of the cycle. If we think of one context as representing one bar of music, this means that the last toggle is the first beat of the next bar, not the last beat of the first bar. So the default context width of 5 boxes (`:X 5`) represents 4 beats in a bar, plus the first beat of the next bar. This is readily seen when the last toggle is used to loop the context, as in '02_connections.pd'.
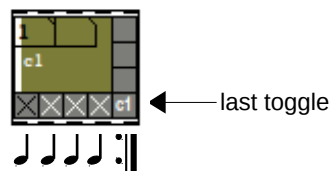
 ← last toggle

Illustration 10: musical notation equivalent of a context :X 5

In this way, the last toggle behaves as if it were a burst toggle. It also reads from the burst database, not the pattern, reaffirming this identity. However, there is one important factor which distinguishes the last toggle from the burst: **the last toggle is immune to all of the burst selection settings** (see Section 11: BURST SETTINGS). If the last toggle is selected, it will definitely fire, and in this regard it behaves exactly like the pattern. This behaviour might seem confusing, but it is very useful in practice.

The last toggle is also immune to the database limit (`:e and :f`). If the last toggle is open, it will always read the last term of the burst database, or a 'bang' if this is empty. (See Section 9: DATABASE LIMITS to learn more about database limits).

	PATTERN TOGGLES	BURST TOGGLES	LAST TOGGLE
playback	definite	controlled by burst settings	definite
primary database	x-axis (:a)	y-axis (:b)	y-axis (last term)
flipped database	y-axis (:b)	x-axis (:a)	y-axis (last term)
primary database limit	:e	:f	no
flipped database limit	:f	:e	no

Table 2: comparrison between pattern, burst and last toggles.

7.ID AND CHANNELS

SUMMARY

- Each Context has one unique ID number and any number of ID tags
- Commands can be sent to Context with [send my-tag]
- Sequenced messages can be received from Context with [receive my-tag-]

Each Context object has one unique ID number and any number of ID tags (unique or shared). The ID number is assigned automatically on creation and saved with the parent patch. It is generally advisable to let ID numbers take care of themselves, although you can choose them manually using the `:ID` command, or in the menu (*ID → ID number*). A context has no tags by default—these need to be assigned by the `:n` command or using the menu (*ID → tags*). To assign using the menu, click on the menu option, then on the symbol atom, and then type in the desired name(s) and press return. If a typed word matches one of the existing tags, it is removed from the tag list; otherwise, it is added.[6]

Each ID term, whether it's a number or a tag, creates one dedicated send and one dedicated receive channel for the given context.

To send a message to the receive channel, use the object **[send my-ID]** where "my-ID" stands for any ID term. (Note that ID numbers needs a 'c' in front of it—for example **[send c3]**.

To receive a message from the send channel, use the object **[receive my-ID-]** where "my-ID" stands for any ID term. Note the dash at the end of the term, used to distinguish the send from the receive channel.
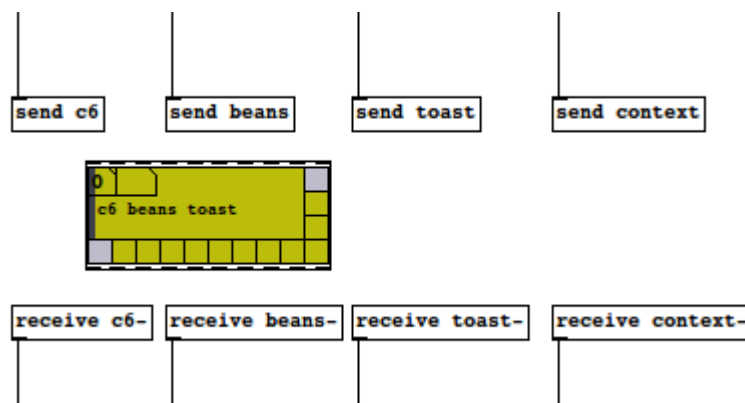
Illustration 11: a context pictured with all its elligible send and receive channels

One Context can have any number of tags, and the same tag can be shared between any number of contexts. This means that tags can be used to form groups of any kind.

Channels 'context' and 'context-' are shared by all contexts ('c' for 'Context'). You can communicate with all open contexts this way, ie. `[; context stop(`.

---

6 Note that a context command is ineligible to be a tag--tagging a context ' `:c` ' would be a very bad idea!

➢ See '04_ID_and_channels.pd' for working examples of tags.

8. CONTEXT COMMANDS (INPUT LANGUAGE)

SUMMARY
- Context commands control Context's behaviour
- Commands can be sent via inlets, receive channels or creation arguments
- The context GUI and menu also send commands

Each context has over 100 settings determining its behaviour. These include the length and width of a context, the toggle selection, the cycle time, and so on. Literally everything that can be done to a context has a corresponding command, and the commands themselves form a rudimentary language which controls context.

It is not important for a user to know everything about the command language, but it is important to have a basic understanding of how it works if you want to take full advantage of Context's capabilities. This section deals with how to use the command language. The file 'context_commands_all.pd' outlines the function of each command individually, with 'context_commands_brief.pd' providing an executive summary.

COMMAND SYNTAX

Every command follows the simple structure of a command term (often referred to simple as the 'command'), sometimes followed by an operator and further arguments.

:s ➢ This command starts Contexts

(command only)

:d 4 ➢This command sets the cycle time to 4

(command, argument)
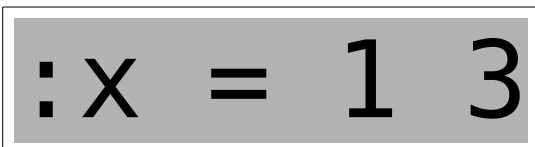
:x = 1 3 ➢This command opens the x-axis toggle 1 and 3, clearing all the rest

(command, operator, arguments)

The command term is distinguished by a colon followed by a letter, or pair of letters, which are caps sensitive[7]. Operators and arguments are taken as per the syntax of the particular command. This is detailed in the file 'context_commands_all.pd'.

Commands work in combination. For instance, ' `:x = 1 3 :d 4 :s` ' would open x-axis toggles 1 and 3, set the cycle time to 4, and start context, all at the same time. Furthermore, this command would be identical to ' `:d 4 :s :x 1 3` '; the oderer of the commands typically doesn't matter. It is important to note that these commands are not comma separated, as is the PD convention. In fact, separating a series of commands with commas will usually invalidate the list.

SENDING COMMANDS

There are 6 ways of sending commands to context.

1. **Sending via inlets**. Although not often the most efficient, this is the most intuitive way of sending messages to context. It is the easiest method for demonstration, and hence it is the method used throughout this documentation. *(See Section 5: CABLE CONNECTIONS).*
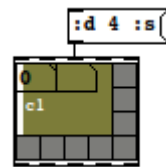
Illustration 12: command via inlet

2. **Sending via receive channels**. This method is useful if you want to send a message to a group of contexts via tags. Sending to ID numbers also works, although it has no clear advantage over 1. *(See Section 7: ID AND CHANNELS).*
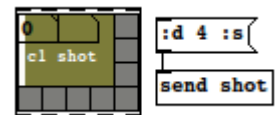
Illustration 13: command via send channel

3. **Sending via sequenced messages**. The messages the Context sends can be directed towards another Context. This powerful technique allows a context network to modify itself, called *dynamic networking*. Note that this actually works using receive channels, as above. *(See Section 12: DIRECTION CHARACTERS).*

Illustration 14: command via sequenced message

4. **As a creation argument.** Context creation arguments follow exactly the same syntax as Context commands. This is how Context saves its state in the parent patch.
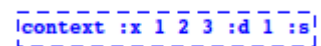
Illustration 15: command as creation argument

---

7 Some commands also have aliases—'`start`' would have done the same thing as '`:s`'

5. **Typing a period (.) followed by a command into the symbol atom.** This is usually the quickest way of changing a setting, if you know the right command. Pressing enter returns to the normal symbol view (*See Section 9: GUI ACCESS TO DATABASE*).

Illustration 16: command by symbol atom

6. **The context menu and the GUI** effectively let you change the context state without knowing the right command. This is often the easiest way, especially for beginners! To access the menu, simply right click on a Context and select *Properties*.

> ➢ Confused? Just remember that sending a command to the inlets, sending it to the receive channel, putting it as a creation argument, and taking a GUI or menu shortcut are essentially the same thing!

Illustration 17: command by menu

WHAT DO ALL THE CONTEXT COMMANDS DO?

This is tantamount to asking "what are all of Context's functions?" hence it is impossible to answer this question in a single section of the manual. However, it may be useful to note some broad, non-exhaustive categories of commands, the same as are partially seen in Illustration 17 of the menu:

1. **TIMING**. `:d :k :E`... define how the cycle time is set and let you control random or positional timings. See Section (10: TIMING).

2. **BURST**: `:h :l :m :o :p`... define how burst toggles are selected at the end of the cycle. See Section (11: BURST SETTINGS).

3. **ID:** `:ID :n :c`... define a Context's ID number, tags, colour etc. (See Section 7: ID AND CHANNELS).

4. **TOGGLES**: `:x :y :i :j`... define the toggle and toggle float arrays.

5. **DATABASES**: `:a :b :e :f`... define the content and behaviour of the message databases (see Section 9: MESSAGE DATABASES).

6. **OPERATION**: `:s stop :r`... define immediate operations like starting and stopping.

> ➢ See the file 'context_commands_brief.pd' for examples of these commands.

> ➢ See the file 'context_commands_all.pd for a list of all possible commands.

The file 'context_commands_all.pd' lists Context commands exhaustively, together with explanations and examples. 'context_commands_brief.pd gives an incomplete, less overwhelming overview of the most important commands. It is recommended that you refer to both of these resources regularly as a learner.

MODIFYING COMMANDS

Some commands set a setting absolutely, whereas others will alter it given the current value. Modifying commands are usually demarcated with an operator. A simple example of this is the command ` :d + 2 `, which adds 2 to the cycle time, whatever it may be. Any command that stores a value can be modified in the same way using arithmetic symbols (`:d  :k  :te`...).

Many commands store only a 1 or a 0, indicating wither a particular function is to be turned on or off, for example, `:t`. Such parameters can all be set absolutely, by sending a value, or they can be toggled by sending the command on its own, without an argument. For instance, ` :t ` will turn the `:t` setting on if it is off, and off if it is on.

`:x, :y, :i, :j , :a` and `:b` each have their own set of modifying operations. These are detailed in the 'context_commands_all.pd' file.

QUOTATION MARKS

Many commands are used to store symbols and lists. For instance, `:a` is used to store lists for the x-axis database, ie. " `:a my message` ". A problem occurs if you want to send data as arguments which is identical with a command, ie. if you want to store the text ' :a ' (or ' :b ' or ' `stop` ') in the `:a` database. The solution is quotation marks. For instance, if you send Context the message `:a ' :a '` , it know to treat the text inside the quotation marks as text rather than a command.[8]

Some cases call for quotation marks within quotation marks (see Section 13: MEMORY BANK). In these cases, you must alter between single and double quotation marks, ie.

` " ' my message ' " `.

9. MESSAGE DATABASES

<div style="border:1px solid; background:#5a2a4a; color:white; padding:1em;">

SUMMARY

- The `:a` and `:b` databases store messages, corresponding to each toggle
- `:e` and `:f` limit how far the database is read during playback
- The symbol atom provides a window to view and edit the databases

</div>

Context has two databases, `:a` and `:b`, corresponding to the x- and y-axis, which store messages for retrieval during the Context cycle. A message can be any text, principally messages that you want to be interpreted by an object outside of context (ie. 'play C#'). Messages can also include message variables and context commands—see Section 12: MESSAGE VARIABLES (OUTPUT LANGUAGE).

---

8 Note that quotation marks are only needed in command messages. If you type commands into the symbol atom or the database edit tool, the quotation marks will take care of themselves (see Section 9: GUI ACCESS TO DATABASE).
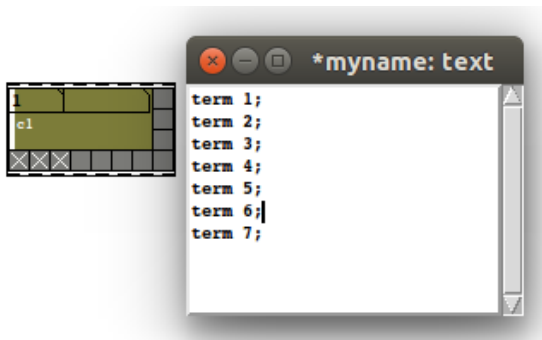
Illustration 18: the message databases appear as a text-window, which you can edit.

The most important thing to know is that **each term in the database corresponds to one particular toggle on the x- or y-axis**. When that toggle fires, it sends the message corresponding to its place on the toggle array, ie. the third toggle from the left on the x-axis sends the third term of the x database. This way, Context can send as many messages in one cycle as there are toggles.

DATABASE LIMITS

There is a limiting factor to this correspondence: the database limits `:e` and `:f`. The database limits set a ceiling on how far the database will be read. Ie. '`:e 3`' means that the database will be read only until the 3rd term, and any subsequent toggles will repeat the third term. If `:e` is equal to or greater than the length of the x-axis then the limit will have no effect, but otherwise it will always mitigate the outgoing messages.

Why are there database limits? Because most uses of Context require for only a single message to be stored and sent. The database limits effectively let choose whether you want Context to store a single list or a multi-term database. The simpler use is the most common, and so default setting for `:e` and `:f` is 1. If you want to make full use of the databases and send multiple messages, use `:e` and `:f` to extend the limits.

To help you remember them, the database limits are indicated on the Context GUI by the colour of the toggles. For instance, the Context depicted in Illustration 19 has settings `:e 3` and `:f 1`.

THE LAST TOGGLE

The last toggle is immune to database limits; it will always be read from the last place on the :b database (or the :a database if the axes are flipped (`:M 1`)). See Section 6: THE LAST TOGGLE.

➤ See file 05_database_limits.pd for a working demonstration.

GUI ACCESS TO DATABASE

The message databases are hidden inside the Context patch, but there is a window into them from the GUI. This is the symbol atom.

The symbol atom lets you view and edit one line of the database at a time. You will see this when a Context is playing—the symbol atom changes when a new message is sent. If you click on the atom and type something, you can enter a new line of text in the last term that was played.

The tricky thing is knowing which term is currently selected. The toggles help with this: **whichever database term is currently in view has its correspoding toggle cross highlighted in black**[9].

You can switch between different terms as well by hovering the mouse-cursor over the Context and using the arrow keys to scroll. Left and right will scroll through the x-axis database (`:a`) while up and down wil scroll through the y-axis (`:b`). It is a good idea to always press an arrow key once before editing, just to make sure you confiirm which term is selected. It might take a bit of getting used to, but soon you'll be able to view and edit the entire database without leaving the main GUI.
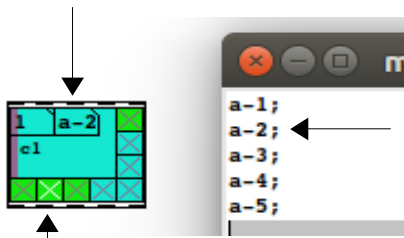
Illustration 19: the green toggles indicate the database limits, and the white cross indicated the position of the symbol atom in the database.

TIPS

• You can only scroll with the arrow key until the end of the database limit. However, holding down the shift key overrides this, allowing to scroll to the end whatever the limit.

• While scrolling, you can also press Tab to toggle the toggles on or off. That way you can edit the pattern and the burst entirely using the keyboard. (Note that the mouse-cursor has to be held over the Context for this to work.)

➢ See file 05_database_limits.pd (subpatch [using the symbol atom]) for a working demonstration.

RESIZING THE SYMBOL ATOM

The symbol atom can be frustratingly small, and longer messages often run over the edge. There are two solutions to this problem. One is resizing the symbol atom, which can be done in the same way as resizing the Context canvas. Hold down the Shift key and click + drag above the right-hand edge of the symbol box. Unfortauntely, a symbol can be no larger than the width of the Context, so this will only be useful in larger Contexts.

Anoter solution is to Shift + double click on the symbol atom. This will bring up a text entry box on top of the Context canvas, which you can enter text into as a regular symbol. The entry box also has the added benefit of allowing for scrolling through and selecting text. It still only lets you view one entry from the databases at a time though, so for serious editing it's still best to call the database using `:a -open`.

Illustration 20: Context with entry box

BUG: The entry crashes if it receives square brackets. Do not use it if the message contains these symbols.

MESSAGE PREPEND AND APPEND

Each term of the message database is independent and unique from every other, but there is a way of defining common elements of an outgoing message. This is the `:S` `:T` `:U` and `:V` settings, which prepend or append a set of constant terms onto a message before it is parsed. For instance, if you set

---

9 If the toggle crosses are already black, then it is highlighted in white.

the command `:T stick`, then every outgoing message from the `:a` database would have the word "stick" appended to it.

The difference between the four commands is as follows:

	prepend	append
:a	:S	:T
:b	:U	:V

Table 3: distinction between message constants

These commands can easily be accessed through the menu: go to *properties → messages → x/y-axis → prepend/append*.

Message constants can be used for any type of custom purpose, but really they are designed for directors (see Section 12: DIRECTION CHARACTERS). For instance, if you set `:S < drums`, then every outgoing message from the `:a` database will be send to the channel "drums". Likewise, if you set `:T . :c ? 30`, then every outgoing message from the `:a` database will trigger a new colour for Context. Why not?

10. TIMING

SUMMARY

- The float atom defines the Context cycle time
- Negative numbers put the cycle in reverse
- `:z` prevents stack overflows
- `:te` speeds up or slows down the cycle time as a percentage

The cycle time or cycle duration is the amount of time it takes Context to complete its cycle, ie. to play its full Pattern and then process the Burst. The cycle time is independent of the size of Context, so enlarging the x-axis will not change the speed, but will effectively change the resolution, or number of beats in the bar.

The cycle time is very easy to control: just scroll the float-atom in the top-left corner of Context to select the duration, in seconds. If you change the timing mid-cycle, it will update in real-time, and the cursor will slow down or speed up accordingly.

Illustration 21: the float atom controls the cycle time

REVERSE

If you select a negative cycle time, Context will work in reverse. The burst will play before the pattern, and the pattern will play backwards, just as expected.

Reverse mode is simple and intuitive, but creates one special problem pertaining to float connection. A Context which uses its last toggle to loop itself will have that loop broken when the cycle time is reversed, since the self-connection now comes at the beginning, not the end of the cycle.

Illustration 22: this Context uses float connection to loop itself at the last toggle, but what happens when the cycle is reversed?

The solution to this problem is the `:rd` setting, which reverses the logic of the last toggle, so that it plays at the end of the pattern, despite being alligned with the burst. This allows self-looped Contexts to remain intact even when the cycle has been reversed. Since it is counter

intuitive it is off by default, but it is very easy to turn this setting on if you want to play around with reversing loops.

➢ See 'context_commands_all.pd' for a working example of :rd and cycle reversal.

➢ Note that the :rev command will also reverse the cycle, without you having to reset the time.

SAFETY

One important question in modular sequencing is how to deal with sequences of zero duration (See '08_burst_and_structure.pd'). But with no duration, a Context can wreck havoc in a network, due to the fact that there is no delay beween one action and its feedback response (see Section 3: FEEDBACK). The first toggle presents the same problem, since it always fires at t=0. Consider in diagram 22, what would happen if the first toggle (on the right here, since the cycle is reversed) were to fire at the beginning of the cycle? It would call c3 to start immediately, causing a stack overflow.

The Context solution is the :z setting which adds two protection mechanisms into the Context process. With :z on,

1. **The minimum cycle time allowed is 0.01 seconds (10 miliseconds)**. This protects against stack overflows since at least some time will elapse before a Context has the chance to send a message back to itself.

2. **Any start message received through the inlet or receive channels will automatically block the inlets for a period of 10 miliseconds**. This protects against the first-toggle overflows described in the paragraph above.

These two protection mechanisms are effective at preventing almost any stack overflow in the network, so they are on by default. However, this comes at a cost. There are some scenarios where you want to have a Context of 0 seconds, especially if you are using the burst as a logic gate. In these cases, you can turn :z off, but beware stack overflows!

CALCULATION

SUMMARY
- Calculation alters the relationship between :d and the cycle time
- :K defines :k as a multiplier, divider, exponent or map to the :d value

Calculation changes the relationship between the the cycle time and the :d value (ie. the number in the float atom). Usually there is a simple one-to-one correspondence between the two, with the :d value denoting the cycle time in seconds, but this can be altered using the :k and :K settings (or the *timing → calculation* menu).

As a simple example, the context state :K 1 :k 60 would determine that the :d value measures minutes instead of seconds, making it easier to alter the time cycle of longer Contexts. How to interpret this command is explained below.

:K selects the calculation mode, of which there are five: 1 *multiplier,* 2 *divider,* 3 *exponent,* 4 *map,* and 0 *none* (default). :k selects a value which pertains to the calculation denoted by the :K mode (default

1), or, in the case of `:K 4`, a list. In each case, the `:K` setting indicates that :d value will be modified by `:k` to select a new cycle time in seconds.

1. **MULTIPLIER** turns the `:k` value into a coefficient, such that $\{cycle\, time\}=d*k$. This is useful for changing the scale of the timing.

2. **DIVIDER** turns the `:d` value into the denominator, such that $\{cycle\, time\}=\dfrac{k}{d}$. This is useful if we think in terms of musical bar lengths. For some length `:k` (say 1 second), `:d 2` means half that length, `:d 3` a third, etc.

3. **EXPONENT** turns the `:d` value into an exponent, such that $\{cycle\, time\}=k^d$. This is useful if we think in terms of standard musical divisions. `:d 1` represents a whole note, `:d 2` a half note, `:d 3` a quarter note, etc. If `:k 3` is set then `:d 2` would become a third note, `:d 3` a ninth, etc. Note that unlike the other settings, the default `:k` value here is 2.

4. **MAP** expects a list in the `:k` setting and maps `:d` values to the list, such that $\{d_1=k_1, d_2=k_2...\}$ etc. If the `:d` value is greater than the number of terms in the `:k` list, then the `:d` value itself is selected.

0. **OFF** turns calculation off. The cycle time is then equal to the :d value, regarless of `:k`.

WHY USE CALCULATION?

At first glance, calculation might seem redundant, since a Context cycle is going to be the same whether it is calculated as 6 or 2x3. But the point of calculation is to let you *change* the cycle time in scaled or non-linear way. Calculation becomes very useful if you plan to change the cycle time dynamically, and especially in combination with random timing (`:E 1`) (see Section 10 RANDOM MODE).

TEMPO

The `:te` command lets you increase or decrease the cycle time by percentage. For instance, `:te 150` would mean that the cyce time is 50% more than `:d` (or whatever calculation has been made on `:d`). This serves an identical function to the calculation multiplier (`:K 1`). It is there because it is useful to have two multipliers, so that one can still be changed if the other has been set. The `:te` setting is very useful for changing the tempo of a network globally (see 'tempo_change.pd').

POSITIONAL MODE

Besides calculation, there are two special functions that can be selected to help with Context timing, POSITIONAL MODE and RANDOM MODE. These can be selected using the `:E` command (or *timing → delay* mode in the Context menu). Positional mode has further settings `:B` and `:C` pertaining to it, while Random mode has `:D :q` and `:Q`. (These can be found in the menu under *timing → position options* and *timing → random options*).

> ### SUMMARY
> - Positional mode relates the cycle time to Context's canvas position
> - `:E 2` turns positional mode on
> - `:C` controls the scale
> - `:B` controls the resolution

Positional mode (`:E 2`) makes use of the spacial nature of PD's programming canvas. When positional mode is on, the cycle time is defined by a Context's position on the canvas, and updates in real time as the the Context is moved. This brings an element of physicality to the network, since the spacial relationships between the different Context units are used to calculate the cycle time.

The scalar value is calculated radially from a given origin point. The origin is the top-let corner of another the Context whose ID number is one less than the Context with positional mode on (or the 0-0 point of the canvas if there is no such Context created).

➢ Positional mode offers an easy way of altering the cycles of two Contexts at once; see '06_positional_mode.pd'

POSITIONAL SCALE AND RESOLUTION

The manner in which the `:d` value is affected by dragging it across the screen is determined by the `:C` and `:B` commands. `:C` (scale) sets how fast the `:d` value scrolls. The default is 20, and higher numbers will result in faster change. `:B` (resolution) sets the intervals at which the :d value will change. For instance, `:B 2` means that :d will jump from 2 to 4 to 6 as Context is moved across the canvas, without hitting anything in between. Fractions are allowed (ie. `:B 0.25`) and long recurring fractions work well for thirds (ie. `:B 0.3333333`).

With these two settings, the user has a great deal of control over how the position of a Context affects its timing with positional mode.

RANDOM MODE

> ### SUMMARY
>
> - Random mode chooses random times for the Context cycle
> - `:E 1` turns random mode on
> - `:d` sets the upper limiting
> - `:q` sets the lower limiting
> - `:D` sets the resolution
> - `:Q` turns auto-advance off
> - `bump d` advances on demand

Random mode (`:E 1`) is one of the most useful Context features. Its function is simple: it chooses a random value for the cycle duration at the end of each cycle. This is an easy way of adding uncertainty into a network, and can be used in many ways to make generative rhythms.

The random numbers fall within a range. The bottom limit is defined by the :q command (or *timing → random options → minimum* in the menu), which is default 1. (Note that `:q 0` allows cycle times of zero, but you should read Section 10: SAFETY before setting this).

The upper limit is defined by the last `:d` value seen before `:E 1` was switched on. You can also select a new upper limit by scrolling the float atom or sending a `:d` command. Note that random mode selects a new cycle time but doesn't actullay effect the `:d` value, so the upper limit remains constant until you change it, despite the fact that the float atom changes.

There are two more commands that relate to random mode: `:D` and `:Q`.

`:D` is a resolution setting which works similarly to random resolution (`:B`), determining the allowed intervals between different random timings. So `:D 2` allows 2 4 6… second cycles (until the upper limit), `:D 0.5` allows 0.5, 1, 1.5… etc.

:Q decides whether or not a new random selection is made at the end of every cycle. Leave it on to achieve the maximum variability of timing within a network; turn it off if you want stability. A special command `bump d` advances a new random timing when `:Q` is off, allowing you to call a new cycle time on demand.

RANDOM MODE AND CALCULATION

Even with the maximum, minimum and resolution settings, random mode can seem problematic if the user wants more control over the random series. Using random mode in conjunction with calculation (Section 10: CALCULATION) greatly expands the possibilities. For instance, using random mode in conjunction with `:K 3` establishes that a random time will be twice, four times or eight times… the length of the original cycle duration. In this way, you can define random series that contain musical significance.

---

11. BURST SETTINGS

<div style="background-color:#5a1a3a; color:white; padding:10px;">

SUMMARY

- The Burst toggles fire at the end of the cycle, but exactly which ones fire can be varied
- The `:L` command offers three presets, ALL, SEQUENCE and RANDOM
- `:h` `:p` `:l` and `:w` determine how many toggles are selected
- `:o` `:P` `:R` and `:W` determine which toggles are selected

</div>

The Burst is a series of toggles, by default the y-axis toggles, which fire simultaneously at the end of the cycle. The beauty of the Burst is that you have a great deal of control over which toggles fire, including the ability to set linear or random sequences, and to determine how many toggles will fire. Because of its adaptability, the Burst is the main "decision maker" in Context, and can be used to define generative sequences and patterns. The parameters that control the Burst are complicated, but it is worth getting to know them, as they can be some of the most important factors which differentiate one Context from another in a network.

➢ Only selected toggles are counted by the Burst, so turning a toggle off guarantees that it will not fire. If you hover the mouse over the Burst toggles, Context will highlight the toggles which are cued up to fire at the end of the next cycle. The cued-up toggles are called the "Burst selection".

➢ The Burst settings are easier to understand hands-on than in theory. This section of the manual goes hand in hand with the file

'07_burst_settings.pd", and it is recommended that you refer to this file as you read to see the burst settings in action.

BASIC SETTINGS

Within the variety of Burst settings, there are three states that are most commonly used: **all**, **linear** and **random**. These can be easily accessed by the `:L` command, or by using the menu (*burst → presets*). Simple though they are, these three settings account for a great many possible uses.

- **ALL** (`:L 0`) means that all open burst toggles will fire. This can multiply the messages in a network, hence a Context set to `:L 0` can be used to control the density of activity.

- **LINEAR** (`:L 1`) means that one toggle will fire each cycle—specifically the next one in line. This can be used to define structure within a network, by assigning a certain path the first time around, another one the second time around, etc.

- **RANDOM** (`:L 2`) means that one random toggle will fire each cycle. This can be used to determine random structures, and to set open loops, etc.

➢ See file '08_burst_and_structure.pd' for examples of how the burst presets can be used in a network.

ADVANCED SETTINGS

There are six main parameters controlling the burst toggles: three for **quantity** and three for **selection**. They are easier to understand if you trace the order that they are executed in:
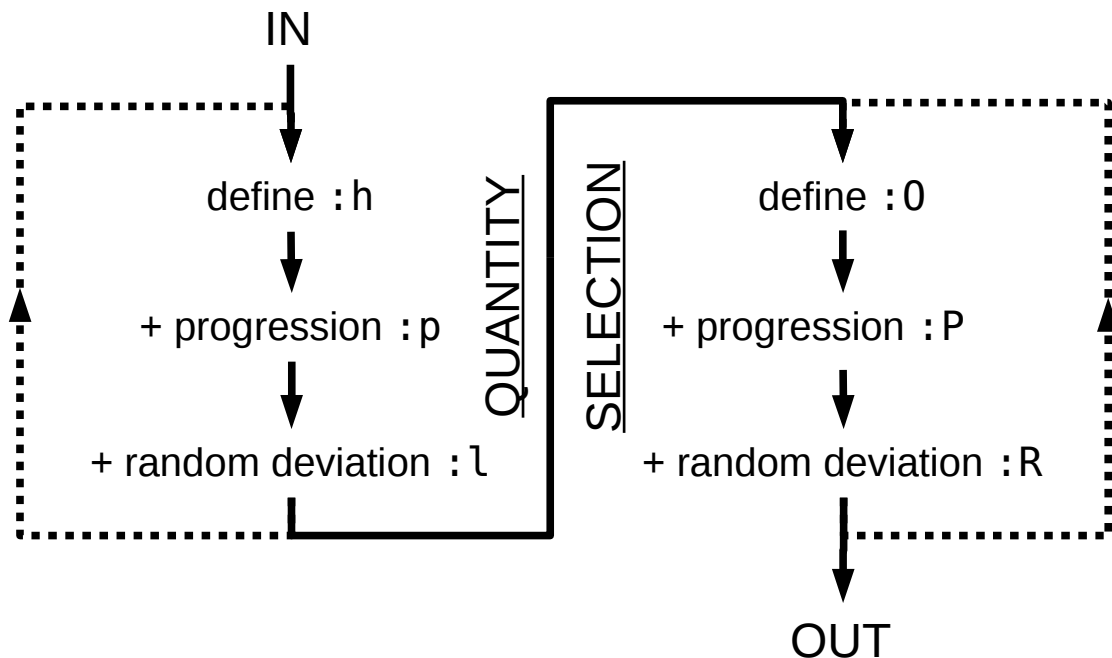
IN

define `:h`   define `:0`

+ progression `:p`   + progression `:P`

QUANTITY   SELECTION

+ random deviation `:l`   + random deviation `:R`

OUT

Illustration 23: order of operations for burst toggle selection

1. First, a starting quantity is set, `:h`. This can be any number between 1 and the total number of burst toggles. 0 wraps around to the highest number, so it effectively means ALL. 0 is the default setting.

2. Next, a number constant `:p` is added to `:h`. `:p` can be anything between 0 and the total number of burst toggles. 0 means "no addition" while the second highest number is effectively -1. Numbers higher than the total number of burst toggles will wrap around.

3. Then a random number is added to sum. The distribution for the random number has many options (see RANDOM DEVIATION below).

4. The resulting number is then passed on to two different places:

    1. It is stored as the new `:h` value, to be used as the base for the next progression and random deviation.

    2. The number is passed on to the **selection** parameters, determining **how many** toggles will be selected.

5. The **selection** parameters are identical to the **quantity** parameters. A starting point is set (or remembered), a progression is added, following by a random deviation.

6. The resulting array of numbers are passed on as the finished **burst selection**, and are stored in an array for the next cycle. Because they are stored in an array, each toggle can follow its own path through the burst.

MANUAL DEFINE (`:o`)

It is possible to define exactly a set of toggles which will fire at any time using the `:o` setting. This should be thought of as a starting point, as the selection will change when the burst advances, according to the other burst settings. The `:h` setting is automatically overwritten by `:o`—if `:o 5 6 7` is set, `:h` will become 3, since there are three items in the list.

As a sort of a shortcut to the `:o` setting, you can manually define the next set of toggles which will fire by selecting them while Context is not cycling. Clicking a toggle on with the mouse pushes it to the front of the list (it is best to select the right toggles quickly, since they are recalculated within one second). Note that unlike `:o`, this method will not update `:h`, and it will not save.

RANDOM DEVIATION

There is very fine control over the random deviation (both quantity `:l` and selection `:R`). The settings go from 0 to 101, covering four distinct phases:

•   0 means determinate; there is no random deviation.

•   < 100 sets the random deviation to Gaussian distrbution. The value determines the sigma of the function, so the higher the number, the more uncertain the deviation. The median point of the distribution is always the stored value `:h` or `:o`, meaning that there is equal opportunity for the random deviation to go up or down.

•   100 selects even distribution. All numbers are equally likely, and any progression `:p` or `:P` is effectively redundant.

•   101 selects Unique Random Number distribution (urn). All random numbers in the "bag" are selected one by one. Once the last one is gone, the "bag" is refilled.

Besides the `:l` and `:R` values, there are two more settings which effect the random deviation.

•   A wrap setting (`:w` and `:W`) can be turned on or off to determine whether random deviations will wrap from the top of the burst to the bottom. For instance, in a Context of length `:X 15`, a toggle stuch at position 15 might jump back down to 1 in one step if `:w 1` is set. If `:w 0` is set, the toggle would have to jump 14 positions to get to 1.

- If the progression (`:p` or `:P`) is set to -1, then the origin point (`:h` or `:o`) is frozen and will not update each cycle. This means that the random deviations will stay around a center point without moving. By contrast, `:p 0` or `:P 0` allow for random deviations to make a 'drunken walk'.

➢ See file '08_burst_settings.pd' for working examples of all the burst settings.

WHY DOES THE BURST WORK LIKE THIS?

The burst process might seem unduly complicated, and indeed its behaviour may remain mysterious even to experienced users. By way of justification, it will perhaps help to know that the burst process is an attempt to connect the three preset states, ALL, SEQUENCE and RANDOM in a continuum, with as few variables as possible. Given the array of options, the preset states states are no longer discrete 'modes', but just configurations, with a whole topology of different options lying in between. The behaviour of the burst can be altered at any time, meaning that the nature of the randomness within a network can itself be dynamic.

If it all seems a bit much, just stick to the presets—they are dynamic enough to be useful in most situations.

12. MESSAGE VARIABLES (OUTPUT LANGUAGE)

SUMMARY

- Context messages are interpreted by the Output Language before they are sent
- Arithmetic is evaluated (inside brackets)
- Various other system values are substituted for special characters
- This can be integrated with the Pattern as a useful way of designing melodies and envelopes

Section 9: MESSAGE DATABASES covers how to get messages into the databases and how to use them while sequencing. This chapter section covers the content of database messages.

In principle, a message can be anything you want: '440', 'A#' and 'launch the rocket' are all valid Context messages which can be sent and interpreted by external objects or patches. However, there are some characters which are interpreted by Context before the message is sent. These include arithmetic symbols and variables. Together, these are known as "message variables" or the "output language". They are fewer and more intuitive than the input language, but no less powerful.

EXAMPLE

Let's start with a simple example. Create a Context and give it the message ' (5 + 20) '

```
:a (5 + 20)
```

When a toggle is opened and Context is started, it will send the message '25'. This is not very useful, since you could have just typed in the message '25' in the first place. Now, type in the message ' (@ + 20) '

```
:a (@ + 20)
```

The @ symbol is a variable which is determined by the Context's position on the canvas. You will probably see a number between 21 and 29, but you can change it by moving the Context away from the origin point of the canvas (usually the top left corner). Any other arithmetic would work as well, so ' (@*2 + 20) ' would probably yield a number between 22 and 38.

Message variables can be combined in almost any way to produce custom patterns and commands.

HOW TO GET MESSAGES OUT OF CONTEXT

Messages are stored in the message databases :a and :b and sent every time a toggle fires. The easiest way to test this out is to type a message into the symbol atom, create the objects [receive cx-] (where x is the Context's ID number), open a few toggles and start the Context (double clicking on its canvas). If this doesn't work, it may be because the database limits :e or :f are set too high. See Section 9: MESSAGE DATABASES for more details.

	CHARACTERS	FUNCTION
ARITHMETIC	+ - * /	*perform arithmetic on numbers*
VARIABLE WITHOUT ARGUMENT	_ = ¬ @ $	*substitute number*
VARIABLE WITH ARGUMENT	^ > ! ` ? # [ ] % ~	*substitute number for argument*
DIRECTOR	( ) < <o & .	*instructions for how to interpret the message*

Table 4: list of message variable characters and categories

ARITHMETIC

 + - * and / work just as you would expect them to (addition, subtraction, multiplication and division), taking one argument before and one argument after the character. The syntax works just about the same as on a scientific cauculator, so

```
4 (5-2) - (5-3)      →      10
3(10(2+3)) [10]       →      150
```

the first important rule of Context messages is that **only arithmetic expressions surrounded by brackets will be evaluated**. So the examples above actually would not work as written—the first would evaluate to "4 3 2" and the second to "3 50". To make them work, you would need to rewrite them as

```
(4 (5-2) – (5-3))

(3(10(2+3)))
```

The bracket rule exists to allow + - * / and % to be used as symbols themselves. If the bracket rule weren't enforced, you could never use these characters in a Context message without them being evaluated.

---

10 Note that Context is very good at interpreting spaces: '3(10(2+3))' is the same as '3 (10 (2 + 3))'.

➢ See files '09_message_variable_examples_1.pd' and '10_message_variable_examples_2.pd' for working examples.

VARIABLES WITHOUT ARGUMENTS

—	The underscore character substitutes the toggle number of whichever toggle triggers the message. For isntance, if the first, third and fifth toggle on the pattern-axis are open, the _ will evaluate to 1, 3 and then 5. _ is useful for for creating a quick line of numbers if you need to evaluate another expression.
=	This has nothing to do with equals! The = character reads the vale of the toggle float from whichever toggle triggers the message. It will not read float-connects (ie. "c4")—it will only read the plain toggle floats, obtained by clicking on a toggle and scrolling *down*. This turns Context into a simple array storage and playback device. Since the toggle floats are so easily set, the = character is the best way of storing simple melodies and other patterns in Context. See Section 5: FLOAT CONNECTIONS) for how to access toggle floats.
¬	The Not sign calculates the distance between the current toggle and the next open toggle. For instance, if the 1st, 2nd and 5th toggles are open on a Context of length 10, ¬ would evaluate to 1, 3, 5 (the last one wraps around to find the first toggle again). Since it looks ahead, ¬ is very useful for calculating the decay function of a note or a drum beat. It can be used to automatically space the values, so that a note will have a shorter decay if it is followed quickly by another note. The ¬ value might need to be scaled to optimize this function (ie. ' (¬ * 20 + 30) '). Note that the ¬ variable doesn't really work for the last toggle—it just outputs 1 (see Section 6: THE LAST TOGGLE).
@	The at sign substitutes the Context's position on the canvas. This is exactly the same scalar as is used in positional timing mode and can be scaled using the same settings. (see Section 10: POSITIONAL MODE). @ brings a physical dimension into the Context network, since a note or other value can be determined by the Context's relative position.
$	The dollar sign substitutes a custom variable, as defined by the :$ input command. This is useful if you want a variable which you can quickly define or change from outside of Context, a bit like the cold inlet of an arithmetic object.

VARIABLES WITH ARGUMENTS

~	The tilde character takes a snapshot of an embedded Content array at the given cursor point. It is extremely useful for parameter control, since the array can be redrawn and moved so easily. See Section 17: CONTENT to learn about Content arrays.
arguments	~ takes a symbol argument to tell it which Content array to read. The Content array name is displayed in the top left corner—usually ~1 or a filename. If there is only one array embedded, you can shortcut this step by using a period, ie. ' ~ . '.
^	The caret character returns a Context ID number chosen from a given set. The set is defined by tags (see Section 7: ID AND CHANNELS). This is useful when sending commands to other Contexts using the > prefix—it means that you don't have to fiddle around to get the right ID number.
arguments	^ takes one symbol argument, which must correspond to a tag used at least once in the network. A period (.) means the variable will choose from the set of all open Contexts. If ^ receives a float as an argument, it will turn it into an ID number, ie. '4' becomes 'c4'.
settings	The ID numbers ^ selects can either be chosen in sequence or at random. The :nr setting determines this.
#	The hash character tranposes numbers into frequencies is a scale. It does this with the help of [scaler], and is no use unless this object is created in a patch. For instance, if the patch contains [scaler minor C 5], ' #2 ' would evaluate to 293.663, the 2nd note of the C minor scale in the 5th octave (D). See Section 17: SCALER).
arguments	# takes a single float as an argument. If the float is not a whole number, [scaler] will interpolate it.
>	Right angle-bracket returns values based on the current Context state. For instance, ' > d ' would return ' 4 ' if the cycle time (:d) were currently set to 4, and ' > x ' would return ' 1 2 3 ' if the :x toggles 1 2 3 were open. This is useful for creating conditional rules (see Section 15: RULES).
arguments	> takes a maximum of four arguments, but only the second one is required. 1. float: The first argument is the ID number of the Context you wish to query, ie. 'c4'. If this is blank then Context will query itself. 2. symbol: The second argument is the context command you wish to query, but without the colon. For example, ' > X ' will query the :X value (length). This argument is mandatory. 3. float: The third argument isolates one of the terms in a list. For instance, if the x-axis has toggle 3 4 5 6 open, ' > x 3 ' returns 5, the 3rd item in the list. 4. float: The fourth argument is only used for querying database message, to help you jump from one line to another. If two float arguments are present, the first one will search for line numbers held

	between bars (' | '), while the second one will search for terms inside that line number. So if the :a database contained the text ' 1 2 3 | 4 5 6 | 7 8 9 ', ' > a 2 3 ' would return '6', the third term of the fourth line. > has a habit of generating errors when used in conjunction with other variables, because of its variable number of arguments. If this happens, put a period ' . ' to indicate where the argument ends and the rest of the message begins.
`	The grave accent (not an inverted comma) lets you perform a custom function on one variable. The function is housed in the overlay, in the last outlet / inlet pair of [ols] / [olr] (see Section 14: Error: Reference source not found). It can consist of any (series of) PD objects.
arguments	` surrounds any number or arguments, ie. " ` 1 2 3 a b c ` ".
[ ]	Square brackets let you process a list. There are three ways of doing this, as indicated by the first argument outside the bracket. The first option one item from the list, based on its position. The second counts the number of items in a list. and the third removed an item from the list. This is useful for conditional rules (see Section 15: RULES), ie. by counting the number toggles that are open in an array.
argumentss	[and] take one float argument on the outside and any number of arguments between them. 1. A positive float argument will take the nth term of the list. Ie., ' 3 [4 5 6 7] ' would return '6', the 3rd item of the list. 2. A 0 will count the items in the list. Ie. ' 0 [4 5 6 7] ' would retirn '4', since the list has 4 items. 3. A negative float removes the positive equivalent of that number from the list. Ie. ' -5 [4 5 6 7] ' would return '4 6 7', since 5 has been removed from the list. Unfortunately, it is not possible to remove a negative number from a list.
?	The question-mark is the most commonly used variable. It produces a random number, or random array (series). Along with the burst (see Secion 11: BURST SETTINGS), it is the main way of creating generative content in a Context network. A single random number is excellent for creating random notes, and a random array works well for setting toggles and toggle floats.
arguments	? can take one, two or no float argguments. 1. One argument (the most common) returns a single random number, with the argument being the maximum. For instance, ' ? 4 ' returns 0 1 2 or 3. Negative arguments will yiled negative results, ie. ' ? -4 ' returns 0 -1 -2 or -3. 2. Two arguments returns an array, with the first argument being the maximum and the second the number of terms. For instance, ' ? 4 3 ' would return ' x x x ', where x = any number between 0 and 4.

	3. No arguments returns the last random number / array, without choosing a new one.
settings	There are three settings associated with ? 1. `:ro` gives an offset, adding a given value to the random number or each number in the array. This is useful in weeding out zeros if you want to use ? to send data to toggles, as ' `:x 0` ' is not often a useful command to accidentally send to a toggle array. 2. `:rm` determines whether ? selects a new random value every time or not. When It is off (`:rm 0`), ? will output the same value(s) until it is told to do otherwise. 3. `bump m` forces a new value when `:rm` is off (`:rm 0`).
!	The exclamation mark finds a value from another Context's message, specifically the Context that started this one. For instance, if this Context was started by a message ' c 4 ', then ' ! c ' would return 4.[11] This is useful in conjunction with arithmetic for creating relative notes and values in a network. For instance, the ! variable can determine that this Context will play a note two notes higher than the last one.
arguments	! takes one argument, a symbol which will be searched for in the incoming message. Since ! will remove the symbol once it evaluates, care should be taken to put it back in if you want to pass the value on to another Context.
%	The percentage sign finds a value from one of this Context's messages, specifically the last one which fired. For instance, if the last message that was sent by this Context was ' d 5 ', then ' `% d` ' would return 5. This is useful for creating relative melodies, ie. melodies which can be transposed into any key.
arguments	`%` takes one argument, a symbol which will be searched for in the last message. Since `%` will remove the symbol once it evaluates, care should be taken to put it back in if you want to pass the value on to another Context.

DIRECTION CHARACTERS

( )	Brackets are used to define order of operations in messages. Variables and arithmetic there are inside brackets will be evaluated before whatever lies without. Note that arithmetic will only be performed if it is inside brackets, to allow for messages that contain arithmetic symbols. So ' 2 + 3 ' would not be altered by the output language, whereas ' (2 + 3) ' would become '5'. Brackets are very important for designing complex messages. When in doubt, use them.

---

11 Every toggle can store a message and can be connected to another Context. "The message that started this Context" means *the message associated with the toggle which hit this Context*. But how does Context know what this message is? Connecting a [print] object to any Context outlet will reveal the answer. A toggle outlet always sends a message "! my message". The exclamation mark starts a Context, but also stores the rest of the list for retrieval by the ! variable.

&	Ampersand is used as a separator for sending more than one message simultaneously. For example, 'message one & message two' would become 'message one' and 'message two', sent one immediately after the other in that order. This is useful for sending a note and another parameter in the same message.
< (prefix)	Left angle-bracket direct the message to some other send channel, as set by the first argument it encounters. For instance, the message ' < drums c 1 ' would send the message 'c 1' to be received by the [receive drums] object (see Section 7: ID AND CHANNELS). Note that this will only work if < is the first character in the message. Putting it directly after an & also works.
<< (prefix)	Two left angle-brackets do the same as one, but bypass the ID channels and 'context' (all) channels.
<o (prefix)	This character directs the message towards the outlets instead of the send channels. This can be useful for sending command messages to other Contexts or controling other PD objects. Like <, it must be at the beginning of a message.[12]
. (prefix)	A period prefix directs the message towards this own Context's input. This is useful for making a Context modify itself using the command language (see Section 8: CONTEXT COMMANDS (INPUT LANGUAGE)). Note that you cannot type a message starting with a period into the symbol atom, because it will send the message immediately, rather than storing it in the database (see Section 8: SENDING COMMANDS). Instead, you must send the message via a command, or open the database editing tool (see Section 9: MESSAGE DATABASES). (Note also that if you are sending a command message into the message database, you need to surround it with quotation marks so that it doesn't get interpreted as input language (see Section 8: QUOTATION MARKS)).[13]
¤	The currency symbol is used to escape a special character from being interpreted, similar for backslah (\) in a command prompt. It is used *after* the character you want to escape, not before. For example, if you wanted to send the message ' # 4 ', but didn't want # to be interpreted by the Output language, you would write ' #¤ 4 '. BUG: The ¤ character is obtined by pressing ESCAPE, but can <u>only</u> be typed into the symbol atom. PD message boxes and the database text box will not accept this character. Hopefully, a better solution will be found for escaping characters one day. In the mean time, you can put ¤ into a message box by using a $1 variable and then sending input to it from a sybmol atom.

---

12 Note that Context always sends its message to the outlets prepended with an exclamation mark, to enable the ! variable to store values. So technically all the <o prefix does is strip the ! character off the beginning of the list.

13 A period is also used to indicate the end of the arguments to the > and ? commands, in case they are shorter than required.

➢ It is possible to use message variables in the input language as well, but you must start with the character '##' For instance, `[context ## :x ? 4 2]` will create a Context with a random number of x-axis toggles.

➢ See files '09_message_variable_examples_1.pd' and '10_message_variable_examples_2.pd' for working examples.

13. MEMORY BANK

SUMMARY

•   The y-axis toggles double as a memory bank
•   To access the bank, hold the mouse over Context, then double click and hold Shift

Most conventional sequencers have the ability to store more than one pattern and jump between them at the press of a button. With physical hardware, this is an important way of saving space and money, but the convention has proven to be useful in software sequencers as well (see for example [Propellerhead's Rebirth]).

Context has a build in bank for exactly the same purpose. By default, it stores the x-axis toggles and toggle floats (:x and :i), but it can be programmed to store any Context setting (see CHOOSING WHICH SETTINGS ARE STORED). Stored Context states are referred to here as "terms", since they are stored as terms in a text object. The bank works via the command language, but can be conveniently controlled via the GUI.

ACCESSING THE BANK

To control the bank via the GUI, first move the mouse cursor over the Context you which to control and then double click and hold the Shift key. The y-axis is then re-purposed into a selector, with each toggle representing one term of the bank. Clicking on the toggles (while still holding the shift key) will load a new Context state, and all changes made to Context will be saved to that slot. Letting go of the shift key will bring back the normal y-axis view.

To control the bank via commands, simply send the message `:U x`, where x is the term number you wish to jump to. (See Section 8: SENDING COMMANDS).

With the GUI control, the number of terms in the bank is limited to the number of toggles on the y-axis. This limit can be overcome using the command controls—if there are 4 y-axis toggles, simply send the message `:U 5`. Notice however that you cannot skip the line; choosing `:U 20` would fall back to `:U 5`.

Using the bank doubles down on memory storage, so it is recommended that you don't use it frequently and on every Context in the network, but only when needed.

BANK SETTINGS

There are settings associated with the bank.

- **:A** determines whether the bank accepts changes to its terms or not. If you turn this off, then Context will recall a certain state but not update it. This is useful if you want to return to a fixed state without it changing.

- **:A bang** saves the memory bank to the patch. This will not happen automatically.

- **:A open** lets you edit the terms of the bank list, using the text editor tool. Bank saving generally takes care of itself, so this isn't strictly necessary, but it can be useful, especially for saving a Context state other than those allowed by the default settings. For instance, you could load two particular cycle time settings (**:k** and **:E**) into two different bank terms and then switch between them easily, without having to edit them manually each time.

CHOOSING WHICH SETTINGS ARE STORED

By default, the x-axis toggle and toggle float settings (:x and :i) are tracked and stored by the bank. This can be changed using the `[preset]` object. Simply create `[preset a b c]` in the overlay, where a b and c are the command terms you wish to be tracked (without the colon). Note that the default commands will be erased from the tracking list if `[preset]` is present, so you will have to add them back if you want them.

See the next section (14: USING THE OVERLAY) to understand what is meant by "create `[preset a b c]` in the overlay."

14. USING THE OVERLAY

SUMMARY
- The overlay is an editable part of the Context patch
- To open the overlay, right click on a Context and select *open*
- To save the overlay, menu-save the popup overlay patch and then menu-save the parent patch
- The overlay can be used for hacking Context and embedding objects

The overlay is a part of the inner working of each Context that is accessible to the user. It is a simple sub-patch that can be called and edited at any time from the parent patch. Besides editing, it can also be saved to the parent patch. This allows for modifications to an individual Context beyond the limitations of the command input parameters. This will not be a familiar concept to most PD users, so it is worth repeating. *The internal workings of a Context can be rewired by the user, and these alterations can be saved to the parent patch.*
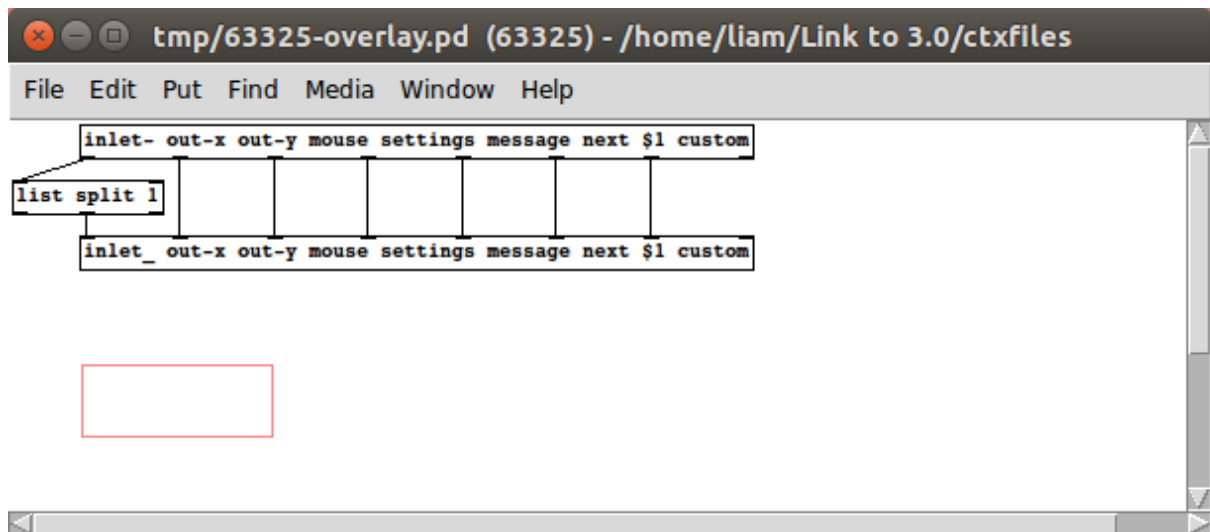
Illustration 24: A standard Overlay patch, featuring the hacking area (top objects) and embedding area (GOP box at the bottom)

ACCESSING AND SAVING THE OVERLAY

Accessing the Overlay is very easy—simply right click on any Context and click o*pen*. The patch that opens is not the main Context patch, as it normally the case in PD. It is the Overlay, a sub-patch within the main Context patch, and you can edit it like any other.

Saving the Overlay is also very easy—just press Control + s or *file → save*, as usual. The important thing, however, is that the parent patch must be saved after the Overlay is saved. This goes for every overlay that you wish to save, and without it, changes to the overlay will not persist the next time you open the patch. Double saving is very easy to do, but just as easy to forget. There is a shortcut to overlay saving with the "save" argument of the [mover] object (see Section 17: MOVER). More information about overlay saving is given in the OVERLAY SAVING section.

➢ WARNING: Do not use sub-patches in the overlay, and do not use objects with commas (including messages and comments). Do not re-size objects or comment boxes. Any of this will break the overlay saving mechanism and / or crash PD.

There are three main things that the overlay is used for:

1. **"Hacking" Context**. Most of the important messages and triggers internal to Context are routed through the overlay. Intercepting them and modifying them with other PD objects can achieve new and customized behaviours for a Context instance;

2. **Custom GUI elements** can be added to the overlay GOP. For instance, if you plan to use the `:0` setting a lot, you could place a float atom on the overlay GOP for swift editing of this command. This is covered in CUSTOM GUI'S;

3. **Embedding objects**, most importantly, Content. Embedding objects means placing them on the timeline, so that they will be played back as a part of the Context cycle.

➢ See '12_overlay_examples.pd' for examples of these techniques.

HACKING

At the top of the Overlay (sometimes off the screen) you will see two objects connected to each other in 7 different places. Most of Context's internal commands are routed through here, and can be intercepted and modified in the overlay. The object names and arguments are there to indicate what each channel holds, as seen in Illustration 24. The first line carries messages from the inlets, before they have been interpreted as commands; the second and third lines carry messages to the cable / float-send outputs; the fourth caries information from the mouse; the fifth all input from commands and settings; and the sixth the next burst selection. The final eight channel, which is not connected by default, is not a central system, but is used for the % message variable (see Section 12: VARIABLES WITH ARGUMENTS).

To hack Context, you will need to intercept any of the outlet channels from the top "inlet-" object. Most of the channels have more than one line flowing through them; you will need to use [route] to identify and separate them. You can then manipulate them in any way and feed it back into the inlets of the second "inlet_" object. Deleting the existing connections will break that part of the Context system, ie. deleting the 'out-x" connection will mean that nothing is sent to the outlets. Proceed with caution here!

This manual does not go into detail about how and what to hack; this is left to the user to do for themselves. Two things, however, are worthy of note:

1. Any PD object can be hosted in the overlay, including Context itself. This way, you can use Context to hack itself.

2. The 'inlet' channel gives the inlet number, followed by whatever message the inlet has received. The inlet number is split off, rendering all inlets identical (see Illustration 24). However, this befaviour can easily be changed by hacking the overlay, letting the user customize different Context inlets for different functions.

3. The overlay has all DSP functions switched off using [switch~] while Context is idle. They are switched on only when the cycle starts. For this reason, you shouldn't host DSP objects inside the overlay unless you are OK with them only functioning while Context is on.

CUSTOM GUI'S

Context has a mimimal GUI, with most of its accessibility being through the menu. However, it is possible to extend the GUI in a custom way by adding elements to the overlay. This is because the overlay contain a Graph On Parent (GOP) area which is visible from the main Context GUI. For instance, if you plan to change the Burst settings a lot on a particular Context, you can add sliders to the GOP area which control the relevant settings. An example of this can be found in the file 'overlay_examples.pd'.

There are some limitations to custom GUIs. First of all, they don't work so well if you wish to re-size Context beyond the boundaries of the GUI objects. Secondly, they might interfere with the mouse control, ie. if you click and hold on a slider, you might end up stopping Context (since this is the usualy function of clicking and holding on the Context canvas). A solution here is to use the Control key, which bypasses the click-and-hold stop. However, this has its own hazards, since the mouse can then interfere with the Context resizing if it strays too far. Perhaps this problem will be solved one day, but for now it must be suffered.

EMBEDDING OBJECTS

The main function for the Context overlay is the embedding. Embedding turns the Context canvas into a timeline, similar to the function of a Digital Audio Workstation (DAW). This is useful for sample playback and enveloping, as well as linear arrangement of music compositions (see [Goodacre]).
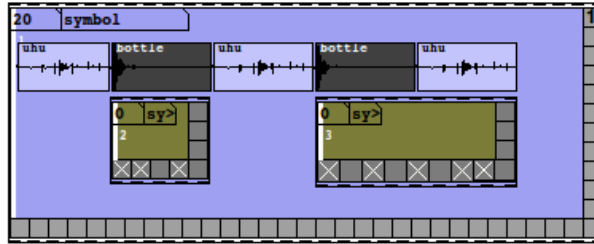
Illustration 25: Context with embedded objects

There are three objects which can be embedded on the timeline:

1. **[content]**, a glorified version of PD's native array object, which can be edited and re-sized for convenient sample playback.

2. **[marker]**, a simple object which pins a custom message to a particular point on the timeline, and directs start and stop messages (see Section 17: MARKER).

3. **[context]** itself. When a Context is embedded, it is connected to the timeline, such that it automatically starts and stops when the cursor crosses its boundary, and loops continuously while the cursor is over it. This simulates the behaviour of a DAW tracker. (Note: this behaviour will only work if DSP is switched on!)

In addition to these three objects, there is a handy **[mover]** object which is used to move and resize objects embedded on the Context canvas.

➢ The behaviour of all of these objects is explained in more detail in Section 17: OTHER OBJECTS, and examples are given in the file '13_embedding.pd' (warning: this patch takes several minutes to load).

OVERLAY SAVING

Interally, the overlay saving is a complicated procedure, and not without its hazards. It has been designed to take care of itself as far as possible, so the information written in this chapter shouldn't be necessary for ordinary use, but it may prove useful, for instance if you want to edit .pd files by hand.

Each overlay patch is first saved to a temporary folder, ctxfiles/tmp. If you open this folder, you will see all the old overlay saves you have ever made. These are only necessary as temporary files, so it is perfectly safe to empty the folder at any time.

The saved overlay patch is then written into the creation argument of the Context object. It is always saved at the end of the creation argument, after all of the commands. A special command, `:H`, always at the beginning of the list, tells Context where the commands end and the overlay saving beggins. Ie. ' `:H 6` ' means that there are 6 command terms, including the ':H' and the '6', before the overlay saving portion begins. Tampering with `:H` will break a Context quite easily, so this setting is best left alone.

Another setting, `:io`, pertains to overlay saving, specifically for saving [ctx inlet~] and [ctx outlet~] objects (see Section 17: INLET~ AND OUTLET~). This helps Context dynamically create the inlet~ and outlet~ objects on loading, and again does not require (or tolerate) any input from the user.

Overlay loading is a heavy process, which noticeably slows down the creation of Context and the loading of the patch (see 20: LOAD TIME). For this reason, it is best to keep the patches small, and to use custom-built abstractions if you intend to hose something larger.

Finally, the above warning should be repeated: **do not use sub-patches [pd …] or commas in the overlay**. In addition to this, it is important not to edit or even move the two hacking objects (inlet- and inlet_) in the overlay. They can, however, be reconnected.

15. RULES

SUMMARY

- Rules govern Context behaviour logically, defining *if-then* clauses
- Rules can be triggered by incoming messages (commands) or outgoing messages (from the database)
- Rules are stored in a special text-edit box (and ultimately the `:ex` command)
- Each rule has two parts: a condition and a consequence, which must be written according to a special syntax

Context deals with messages. It receives messages in the command language command language, which controls all of Context's variables and settings in a deterministic way (see Section 8: CONTEXT COMMANDS (INPUT LANGUAGE)). It sends messages from the message database (see Section 9: MESSAGE DATABASES), which can either be directed inwards towards the network itself or outwards to other PD instruments. All of this activity takes place *sequentially*, according to the Pattern and Timing, and *randomly*, according to the Burst.

Rules govern Context's activity *logically*. They are a way of determining outcomes based on the messages that context sends or receives, apart from the meaning that those messages have. The easiest way to understand rules is to think of them as *if-then* clauses. *If the cycle time is greater than 4,* *then* *open the second toggle*, or *if Context sends the message "'c#',* *then* *send the message 'e'* are both valid examples of rules.

There are two types of rules: those which are triggered by incoming messages commands (see Section 8) and those that are triggered by outgoing database messages (see Section 12). Rules can be very powerful in a Context network:

- They allow for complex relationships between Context objects. For instance the toggles of one can be tied to another;

- They are defined textually, allowing for a great deal of versatility and power;

- They are triggered conditionally, based on the messages that Context sends or receives.

Writing rules involves both the input and the output language, so it is best to be familiar with the material in Sections 8 and 12 before proceeding further here. Besides this, rules have their own set of commands and procedures which are described below.

ACCESSING RULES

Rules are edited and stored in a text object. Accessing them is similar to accessing the bank list in that it can be opened and saved through the menu (*other → rules*) or by dedicated commands, `:I -open` and `:I -save`. Unlike the bank, there is no GUI shortcut; rules must be edited in the text editor. In addition, rules can be turned on or off with `:I 1` and `:I 0`. `:I 0` means that all rules will be disregaraded. Rules are off by default and should be kept that way unless they are needed, since they cost the CPU.

Unlike the bank, rules must be saved manually. Go to *properties → other → rules → save* and then menu save the patch if you want them to persist when you next open the patch. Using the `:I -save` command works just as well.

BASIC SYNTAX OF RULES

Each rule takes up two lines. The first line is a condition and the second line is a consequence. If the condition is met by any incoming command, the consequence is executed. The two lines must be conjascent and must be terminated with semi-colons, or else the rule will be invalid. One Context can any number of rules. If there are two commands pertaining to the same command, they must be numbered in order (see CONDITION TERMS).

	SYNTAX	EXAMPLE 1	EXAMPLE 2
CONDITION	NUMBER, TERM,{]}, QUERY,{VALUES};	0 :d ;	0 :x] q> 3 ;
CONSEQUENCE	INDICATOR,{DIRECTOR},{COMMAND},{ARGUMENTS};	c- . :d 4 ;	1 c* . :y def > x ;
DESCRIPTION		*The :d value (cycle time) is always 4.*	*If the x-axis has more than three open toggles then the y-axis should match the x-axis toggles.*

Table 5: Basic syntax of rules. {Curly brackets} indicate optional terms; commas are not required.

CONDITION TERMS

- **NUMBER**: Each rule must have a number as its first term. Usually this is 0 for an incoming rule and 1 for an outgoing rule. The numbers serve two purposes.

    - It determines whether the rule applies to incoming or outgoing messages. Positive numbers indicate incoming while negative indicate outgoing.

    - It determines the order that rules will be searched for. If a single message triggers multiple rules, it will trigger them in the order that they are numbered. Note that you only need to order the numbers if you have more than one rule pertaining to the same command. Otherwise, they should all be 0 (outgoing) or -1 (incoming).

- **TERM**: This is the term that will be searched against the incoming / outgoing messages. For incoming rules, it will always by a Context command, complete with semi-colons, as described in Section 8: CONTEXT COMMANDS (INPUT LANGUAGE) and 'context_commands_all.pd'. For outgoing rules, it can be anything.

- **QUERY**: The query determines what type of condition is to be met in the arguments of the command. For instance, it could ask if the arguments are *greater than* a certain value, or whether they *contain* a certain term. There are seven possible queries, listed below (see QUERIES).

- **]**: The square bracket indicates how you want the arguments of the previously metioned command are to be queried. Without the character, the arguments will be taken as they are; with the character, the arguments will be counted. For instance, `:x contains 3` asks whether the 3rd toggle on the x-axis is open, whereas `:x ] greaterthan 3` asks whether there are more than 3 toggles open.

- **VALUES**: These offer the value(s) that must be met by the query of the command terms. Usually there is just one argument (see QUERIES below).

CONSEQUENCE TERMS

- **INDICATOR**: The indicator indicates what should happen if a matching condition is found. There are four indicators: `c*`, `c+`, `c-` and `c/`. See INDICATORS below for a descriptionof their function (they have nothing to do with arithmetic). If the indicator is missing then the rule will be ignored and the incoming command will pass as usual.

- **DIRECTOR**: The director determines where the message will be sent. These are the same set of characters as in 12: DIRECTION CHARACTERS. A period (`.`) is used to direct the message to itself. If no director is specified then the message will proceed to the send channels (see Section 7: ID AND CHANNELS).

- **COMMAND, ARGUMENTS**: These form the new, outgoing command that should be sent if a match on in incoming command is met. They must adhere to the normal Command language (see Section 8: CONTEXT COMMANDS (INPUT LANGUAGE)). The Command and argument are optional because the consequence of a matching condition could be "no acation". Ie. if the consequence term is simply the indicator ' - ', then the incoming command will be 'swallowed'.

QUERIES

All queries are written as a 'q' plus logic symbol, altough each one has aliases. Each query returns a boolian response, 1 or 0, which reports whether the condition has been met.

QUERY	ALIAS	DESCRIPTION	NOTE
q<	less, lessthan, less-than	Checks whether the incoming command's argument is less than the given value.	1
q>	greater, greaterthan, greater-than	Checks whether the incoming command's argument is greater than the given value.	1
q<=	lessor, less-or	Checks whether the incoming command's argument is less than or equal to the given value.	1
q>=	greateror, greater-or	Checks whether the incoming command's argument is greater than or equal to the given value.	1
q==	equals	Checks whether the incoming command's argument is equal to the given value.	1
qn==	equalsnot, equals-not	Checks whether the incoming command's argument is not equal to the given value.	1
qn=	containsnot, contains-not	Checks whether the incoming command's argument does not contain the given term(s).	2
q=	contains	Checks whether the incoming command's argument contains the given term(s).	2 If the query term is left blank, q= will be selected

			as default.

1: These queries compare two values. They make most sense in conjunction with a command that has only a single float argument, such as `:d` or `:k`. They also work well with the `]` operator, which counts the number arguments, ie. `:x ] q> 3` means, *are there more than three toggles open?*

2: These queries compare a list to another list, which can be of any legth. They are useful for querying toggles, toggle floats and message databases.

INDICATORS

INDICATOR	ALIAS	DESCRIPTION
`c-`	`replace`	This indicates that the incoming condition-matching command should be replaced by the consequence
`c*`	`after`	This indicates that the incoming condition-matching command should be executed after the consequence
`c+`	`before`	This indicates that the incoming condition-matching command should be executed before the consequence.
`c/`	`goto`	This indicator refers to another consequence in the same rule list. It takes a single float argument, which is the line number, starting from 0. Ie. ' `c/ 5` ' refers to the 6th line. If there is no consequence on the given line, the rule will fail.

➢ Rules are non-interactive with each other, ie. the consequence of one rule cannot trigger the condition of another.

➢ Besides the terms described here, a rule can contain any variable from the output language in its arguments. See 14_rules.pd for examples.

16. HOLDING MESSAGES

SUMMARY

- The `:N` command puts the next Context input in a queue
- The first argument specifies the length of the queue while the second specifies the trigger
- `:N` will hold the next command, even if it is received through the GUI.

The `:N` command has a special function of queuing other incoming messages. It is like a traffic warden that signals out particular messages and holds them for a specified period. The period is not defined in time, but in input messages. For instance, the the `:N` command could hold a `start` message until it the next time it

receives input starting with `:x`. Or it could make an `:x` command wait until Context cycles another 3 times. As with rules (section 15), this is another way of controling Context logically.

Holding messages is easier than using rules, since it is carried out entirely in input commands (see Section 8: CONTEXT COMMANDS (INPUT LANGUAGE)). It takes two arguments, the second one being optional. The first argument is a float, which indicates the number of steps that the following command should be held. The second indicates the input that will trigger the advancement of the message. This must be an input command, minus the colon (eg. 'x' for '`:x`', 's' for '`start`', etc.). If this argument is blank, then any input will trigger the message to be advanced.

The `:N` command <u>always puts the next single received command in the queue</u>. Usually this next message is contained in the same message as `:N`. Ie. the message:

```
:N 3 :x 2
```

tells Context to hold the command :x 2 and then release it after any 3 successful input commands are received. This means that the second x-axis toggle will open (or close) only after Context receives 3 more inputs (perhaps 'start' commands).

If :N has no further commands attached to it, then Context will wait to hold the next input that it receives. This includes GUI control as well as command input. For example, if you snd the message

```
:N 18 y
```

and then opened the first x-axis toggle, then this toggle would be put in the queue. Furthermore, the x-axis toggle would only advance towards to front of the queue when Context receives a y-input command (since the `:N` command had the y argument). The result is that the first x-axis toggle will open (or close) only after the 18th `:y` command received by that Context.

Note that when queuing `:x` `:y` or `:d` messages, the GUI will reflect the anticipated, not the current, state. Ie. if you send the message ' `:N 3 :x 1` ', the first toggle will appear open on the screen, but will refuse to send any output until 3 more inputs are received. Playing around with `:N` commands should make this clear.

➤ See 'context_commands_all.pd' for working examples of message queues.

17.   OTHER OBJECTS

Context is unusual as a piece of software in that it puts so much emphasis on a single object. It does, however, come with a small library of other objects that enhance its behaviour, a bit like plugins. Most of these objects are mean to be created in the Overlay (see Section 14: USING THE OVERLAY) and will not be useful outside of Context. However, some of them perform significant functions (especially Content), and it is worth getting acquainted with them, especially the ones presented here first. Hopefully the list of plugin objects will grow in the future.

CONTENT

SUMMARY

- [content] stores arrays and plays them back in time with the cycle
- Arrays can either be loaded from an external file (:lo) or drawn in the canvas
- Content has a hidden audio outlet~ at the bottom left
- Content has its own menu and command system
- Small arrays can be saved to the patch

[content] is Context's best friend. It is a modified PD "array" object which holds an array of numbers and stores them for playback. Content is integrated into the Overlay such that its playback synchs with the position of the cursor. It can be moved and resized easily on the canvas using the [mover] object (see the next section MOVER).

Content is sophisticated enough to have its own commands and menu system (see CONTENT SETTINGS below). This works in much the same way as Context's and is not very complicated, so you will pick them up easily if you have made it this far.
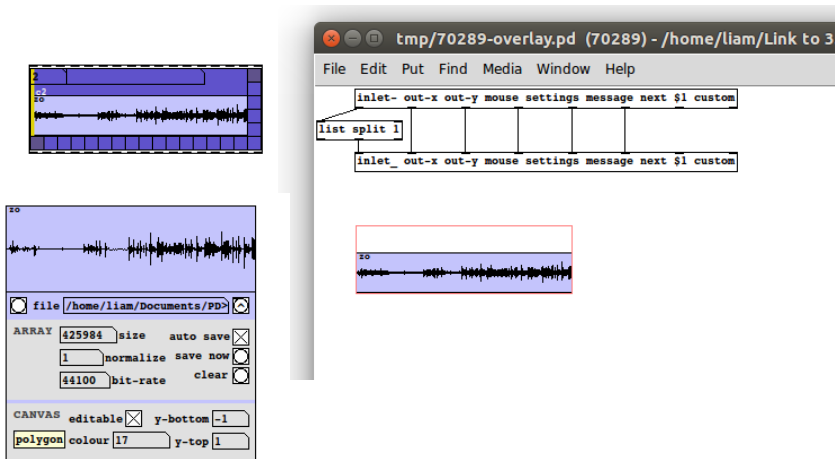
Illustration 26: Three views of content: embedded (top left), overlay (right), and properties view (bottom left)

CREATING CONTENT

Content must be loaded in the Overlay of a Context; it is not very useful outside of this. To access the Overlay of any Context, right click it and select 'open'. Then create the object [content] and move it into the small GOP box (see Section 14: USING THE OVERLAY). If all its boundaries lie within the GOP box, it will appear on the Context canvas as soon as you exit the overlay.

Content has one inlet and one outlet, both hidden on the canvas. The inlet receives messages, while the outlet sends signals. So you should immediately connect Content to another DSP object if you wish to use its signal (ie. [dac~] or [send~]).

The larger task is usually getting Content to the right size. This is acomplished using the [mover] object; see subsection MOVER below.

LOADING ARRAYS

Content can either load audio files using [soundfiler], or can have its values set / drawn by the user.

- To load an array, send Content the following command `:lo path/name.wav` .

- Alternatively, right click on Content and select 'open' to get the 'open file' panel and select from there.

- The `:lo` command without any argument will also open this panel, so creating [content :lo] might be a faster way of loading a file.

- If the `:lo` command receives an argument that doesn't have a file type (ie. does not contain .xxx), then it will search for a local array of the same name. For instance, '`:lo array1`' would mean that Content reads from array1, if it is present, instead of its own array. This is useful in case you want to load the same sample twice, to save on memory.

- To draw an array, simply draw using the mouse, or use the `set` command (see CONTENT SETTINGS below).

GETTING OUTPUT

If Content is embedded in the Overlay of another Context, it will automatically play during the Context cycle. The position of the playback is determined by the position of the cursor on the canvas. To get signal out of Content, simply use its audio outlet. You can then send it anywhere in the patch using [send~] or [throw~] objects. You can also use the [ctx] object to create a signal outlet on Context if you want to work in the parent patch (see INLET~ AND OUTLET~ below).

Note that you can also get non-signal output out of Content using a message variable. The procedure for this is very easy—simply type tilde (~) into any Context message and it will automatically read the array in place of the character. This will give a snapshot of the array at a particular point, as defined by the pattern toggles, and is extremely useful for setting envelopes and modulation parameters. See Section 12: VARIABLES WITH ARGUMENTS and '10_message_variable_examples_2.pd' for more examples.
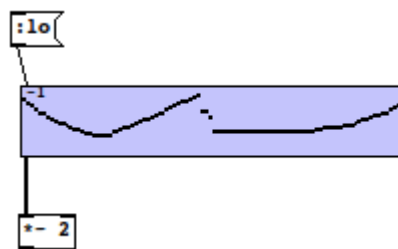
Illustration 27: Content has one message inlet and one signal outlet, both hidden from view.

CONTENT SETTINGS

Like Context, Content has its own settings menu. Right click and select 'properties' to view it. Here you will see the familiar PD array settings, plus a few extras.

SETTING	COMMAND	DEFAULT	DESCRIPTION
load	`:lo`	blank	This loads a file into the Content array. It should take a file path, absolute or relative. Alternatively, it can take the name of a local array, ie ' `:lo array 1` '. Sending `:lo` on its own summons the *open-file* panel.
size	`n/a`	100	This is the size of the array in samples (not its physical dimenstions).
normalize	`:nm`	1	Normalizes the whole array to a given value around 1.
bitrate	`:br`	44100	If the bitrate of the audio file differs from 44100 samples per second, you should type it in here.
array type	`:li`	0 (polygon)	This is the way the array is displayed on the canvas: as a polygon (0), points (1) or a curve (2).
editable	`:ed`	0 if file loaded, else 1	This is a useful feature that turns off the mouse editing of the array. Set this if you have loaded a sample and don't want to accidentally redraw it in the parent patch while using Context.
colour	`:c`	17	These are the same colours as Context.
y-bottom	`:s1`	-1	This is the value of the bottom bar of the graph. It can set the scale of the graph, together with `:s2`.
y-top	`:s2`	1	This is the value of the top bar of the graph. It can set the scale of the graph, together with `:s1`.
save now	`save`	n/a	This saves the Content array to the creation argument. This is only advisable with smaller arrays (~128 samples), ie. not with samples.
auto save	`:as`	0	If this is on then the Contents of the array will automatically save to the creation argument. This is only advisable with smaller arrays (~128 samples), ie. not with samples.
clear	`clear`	n/a	This clears the array to a constant 0.
print state	`print`	n/a	Print the current Content state.
set array	`set`	n/a	Set values in the array. The values are sent stepwise, ie. `set y1 x1 y2 x2…` . Or, with the * preset, they are sent as a list, ie. `set * x1 x2 x3…`
set position (absolute)	`:x1 :x2 :y1 :y2`	0	Set the position of Content manually. `:x1` is the left boundary, `:x2` the right boundary, `:y1` the top and `:y2` the bottom. The values are in seconds, relative to the cycle time of the host Context.
set position (relative)	`:x1 +`	0	You can nudge the position of Content using the + command, ie. ' `:x1 + .5` ' would move the sample ½ a second to the

			right. This works for all of the positioning commands.

SAVING

Content has two options for saving. It can eiter save the filename or array name in reference to an exeternal file (faster). This is done using the `:lo` command, as above. Alternatively, it can save the values themselves to the creation argument. This is very useful for envelope arrays, as it can be done entirely within the patch, but it does take longer to load.

There are three steps necessary for saving an array to the creation argument.

1. Use the `save` command, or go to *properties → save.*

2. Save the Overlay patch. (This just means Ctr+S in the Overlay patch—see Section 14: OVERLAY SAVING).

3. Save the parent patch.

Step 1 can be bypassed if you have turned on auto-save, `:as 1`.

➢ WARNING: For now, Saving arrays is a perilous business in Context, and it is not uncommon to open a patch and find that an embedded array has loaded all wrong. Hopefully this will improve with time, but for now

MOVER

> ### **SUMMARY**
> - [mover] is used to move embedded [content], [context] and [marker] objects
> - Mover must be created in the Overlay but works in the parent patch
> - Holding the Shift key turns marker on
> - Double pressing and holding it flips the rotation
> - Move objects by dragging the left / top edges
> - Resize objects by dragging the rigth / bottom edges
> - Mover can move objects normaly, fast or slow by clicking, double clicking or tripple clicking
> - Mover can take an optional argument to auto-save the Overlay

The object [mover] makes embedding objects much easier. It's job is simply to move overlay objects around the GOP area, and to resize them. This functionality approaches the behaviour of a DAW, where samples and instruments can be moved around a spacial environment to locate their position in time. In Context this process is not as smooth as with professional software, but it is still a very handy feature for working with arrays.

There are three objects that [mover] can move: Content (see 17: CONTENT, Context, and Marker (see 17: MARKER). It moves them all in more or less the same way, as described below.

Only one [mover] object needs to be created per Context; it is placed in the overlay (it's initial position is incidental). However, Mover will only function from the parent patch; you won't be able to use it while the Overlay patch is active.

MOVING OBJECTS WITH MOVER

To move an object on the overlay:

1. Make sure that (only) one object [mover] is created in the Overlay;

2. Close the overlay and hold the mouse over the Context in the main patch;

3. Press and hold the Shift key. Now you will see Mover showing up in the Context canvas. It tracks the mouse position as you move it;

4. Bring it to the left-hand edge of a movable object and wait for the line to go black;

5. Click and drag the object horizontally;

6. Now, bring Mover to the right hand edge of a movable object;

7. Click and drag to re-size it;

8. Lift the Shift key once you have finished. Voila!

9. To move an object vertically, double press and hold Shift. Then follow the same instructions as before; the top edge moves while the bottom edge re-sizes.

➢ WARNING: Moving and resizing larger arrays can take a lot of time and may cause lag and audio dropouts. This is because the array must be deleted and redrawn on the canvas every time it is moved (which is happening 5 times per second when Mover is active. Because of this, moving arrays is best kept for designing networks, not for performance.

➢ See Example 3 in the file '13_embedding.pd' to practice using Mover. (Warning: this patch takes several minutes to load).

PRECISION MOVING

Mover has three "speeds" of moving an object:

1. **NORMAL**: One pixel at a time;[14]

2. **FAST**: One toggle box at a time;

3. **SLOW**: 10 miliseconds at a time (this is variable: see CREATION ARGUMENTS).[15]

Choosing between these modes is easy—At step 5 or 7 above, simply double click against the edge of the boundary for fast and tripple click for slow. Normal is achieved by single clicking.

Note that fast moving is the best option if you want the sample to align with a boundary of the Context canvas. It is also best for moving larger arrays, as it minimizes lag.

---

14 The time equivalent of one pixel can vary depending on the size of the Context host and its cycle time (`:d`); with more extreme settings, one pixel could become a noticeable amount of time.

15 Note that with this setting, the moving object will not move across the screen at the same rate as the mouse—indeed, it might not move visibly at all. But the time index is updated whether you see it or not.

CREATION ARGUMENTS

[mover] has two possible creation arguments:

1. The flag '`save`'. This triggers an Overlay save shortly after each successful object movement. This is extremely handy if you are editing an embedded array frequently, as it bypasses the need to open the Overlay and save it manually each time.

2. A float argument which overrides the 10ms resolution and replaces it with another value for slow moving, in miliseconds. The argument is given in seconds, not miliseconds.
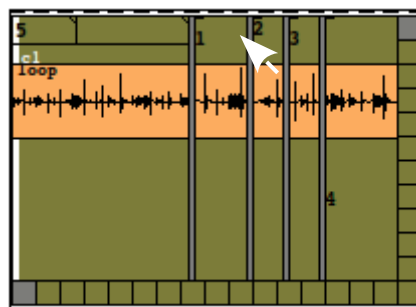
MARKER

SUMMARY

- [marker] marks a point for Context to start or stop during playback
- Markers must be created in the overlay
- Markers number themselves
- Markers can store Context messages and send them during playback

Marker's job is to guide start and stop commands in an embedded Context to a specific place. This is a bit like setting up boundaries at the start and end of a waveform. It can be used for beat-slicing samples, and for creating custom paths through the Context cycle.

To use Marker, create a [marker] object in the Overlay (see Section 14: USING THE OVERLAY). Marker will align itself automatically to the GOP area, and can be moved horizonally within that space. Marker works in conjunction with Mover, so it is best to create this object too, as well a Content sample. Markers commonly come in pairs; create as many as you want.

Each Marker has a unique ID number. Unlike Context ID numbers, these are fixed to the position, not to the object—the first from the left is always 1, the second from the left 2, etc. Marker ID numbers are used for sending information to Marker, and for referencing with Context `start` commands (see START AND STOP below).

Illustration 28: Four markers embedded in a Context, with an arrow showing the approximate place for moving.

MOVING

Markers can be moved from the main patch using [mover]. This does not seem to work as well as it does for moving [content], but the trick is to aim for the area just <u>below</u> the number label (see Illustration 28). The label itself can be moved up or down the marker using the y-shift mode over Mover (double press and hold shift; see Section 17: MOVER). This, however, has to be done from slightly <u>above</u> the label—at the very top of the Context GUI object, in fact. This is not as intuitive as it should be, but still quite convenient once you get the hang of it.

Markers can also be moved manually with commands: see MESSAGES below.

START AND STOP

Marker synchronizes with the `start` commands of its host Context. For example, `start m 1` will jump to marker number 1 and start the cycle from there.

You can also schedule a stop using the same command. For instance, `start m 1 2` will start at the first marker and stop at the second marker. This is a very easy way of beatslicing in Context.

➢ See 'marker-help.pd' for a working demonstration of Marker staring and stopping.

MESSAGES

Besides directing `start` commands, Marker can also store and send a custom message (one per marker). It is sort of like a guest member of the `:a` database (see Section 9: MESSAGE DATABASES), in that it fires at a defined time during the Pattern playback, except that its position is completely flexible.

To set the Marker message, simply use the `:a` command. For instance, ' `:a here I am` ' will send the message 'here I am' when the cursor crosses that point in the cycle. **Note that DSP must be switched on in order to send marker messages**, as it depends on a DSP signal to trigger the message.

Marker messages are dynamic and can use any terms from the Output language (see Section 12: MESSAGE VARIABLES (OUTPUT LANGUAGE)). This means that random numbers, toggle floats, etc., are all eligible to fit in Marker messages. However, two variables differ from regular context messaeges:

1. The @ symbol gives the Marker ID number instead of the Context position;

2. The @@ symbol gives the time index.

Note that the ~ message variable is especially useful for using with markers, since it takes a snaphot of the array at a custom time index.

It is also possible to use Marker messages to send commands to Context, using the period (`.`) director, or custom send director (`<`). See Setion 12: DIRECTION CHARACTERS.

➢ See 'marker-help.pd' for a working demonstration of Marker messaging.

COMMANDS

Marker has a small number of commands and a saving system built in. Unlike Context and Content, it doesn't have a menu-system—commands must be sent to it by text. There are two options for doing this:

1. Via the inlet on the top (this can only be accessed from the Overlay;

2. By sending commands through Context.

The latter is done using the '`marker`' command, which takes the Marker number as an argument. For instance. `marker 1 ' :a my message '` would set Marker number 1's message (`:a`) to ' my message '.

Note the important quotation marks around the Marker command—this is there to distinguish it from Context's own `:a` command. For more information, see Section 8: QUOTATION MARKS)).

There are only three commands for Marker:

SETTING	COMMAND	DESCRIPTION
x-position	`:x1`	This specifies the Marker's position along the x-axis, in seconds.
label position	`:y1`	This specifies the vertical position of the label on the marker, also, strangely, in seconds.
message	`:a`	This stores the marker message which will be sent when the cursor matches the Marker's position.

INLET~ AND OUTLET~

SUMMARY

- [ctx inlet~] and [ctx outlet~] create inlets~ and outlets~ on the Context GUI leading to the Overlay

The object [ctx inlet~] and [ctx outlet~] are used to create audio inlets and outlets from the Overlay to the main Context object. The [ctx] objects can be created anywhere in the Overlay patch; the new in/outlets~ appear as the rightmost on the Context GUI (the others in the row will move over to make space for them.) You can see this in action in the patch 'embedding.pd' (warning: this file takes several minutes to load).

[ctx] objects should save as normal in the Overlay—see Section 14: OVERLAY SAVING).

SCALER

SUMMARY

- [scaler] turns numbers into notes on a scale
- It lieases with the # message variable

[scaler] is used to turn numbers into notes on a scale. It makes most sense when seen with its cretion arguments. Ie. the object [scaler minor Ab 3] turns the number 2 into 116.5, which is the 2nd note of the 3rd octave of the Ab minor scale (Bb). Scaler communicates internally with Context—it needn't be connected to anything. The # message variable will automatically take a value from any [scaler] object open in the patch (see Section12: VARIABLES WITH ARGUMENTS). Scaler may be placed in the Overlay, or in the parent patch. In the Overlay, it will be attached to only that one

Context, allowing each one to have a different scale. In the parent patch, it will communicate with all open Contexts, allowing for global scales.

You can also use the in/outlets of Scaler to achieve the same thing. This is the only object in the library that is useful outside of Context.

- **Eligible mode arguments**: minor, aeolian, locrian, majo,r ionian, dorian, phrygian, lydian, myxolydian, melodic ascending minor, phrygian raised sixth, lydian raised fifth, lydian myxolydian, major minor, locrian sharp, altered, pentatonic major, pentatonic minor, pentatonic egyptian, blues minor, blues major

- **Eligible key arguments**: a, a#, ab, b, b#, bb… gb.

- **Eligible octave argument**: any float.

UNDO

<div>

SUMMARY

- [undo] is used for tracking changes to Context and undoing them from the GUI

</div>

[undo] allows you to create an undo stack within Context. It tracks all changes made to Context and allows you to roll back to them at a later date.
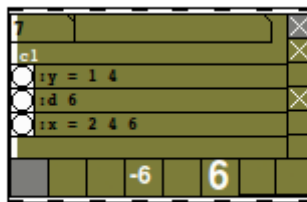
Illustration 29: An example of an undo stack. The lines of text in the middle of the Context canvas were states and can be reselected by pressing the button.

TO CREATE AN UNDO STACK

1. Open the Overlay

2. Create the object [undo]. It will align itself horizontally to the GOP area automatically; you need to align it vertically to where you want it. (If the Undo box is too short then resize the Context x-axis and it should reset itself).

3. Create as many other [undo] objects beneath it. as you want. Make sure to stack them in the same order as they are created.

4. Save the Overlay and exit. The [undo] objects will automatically track changes made to the context.

Every time you make a change, to the Context state, the previous state will be sent to the top Undo objet. Previous entries will move down the list. (Note: they travel according to creation order, not position, so it is up to you to stack them in the right way). To revert to a previous state, simply click on the button.

Undo stacks are particularly useful for Contexts which are receiveing random or dynamic commands, in case something goes wrong and you want to quickly revert back. However, the do take up extra memory, so don't use them unless you need them.

KEYLIST

<div style="border:1px solid; background:#5e2850; padding:10px;">

SUMMARY

- [keylist] binds keyboard keys to specific Context commands
- It must be created in the Overlay
- It can take numerical keyboard input or input through its inlet

</div>

The [keylist] object is used to send specific commands to Context at the press of a single key. It is a bit like a shortcut bank for rapid sending of a pre-designed message. For instance, it can enable keyboard triggering of starting the Context cycle at a mid-point (see Section 3: STARTING AND STOPPING CONTEXT) or to send a hold command (see Section 16: HOLDING MESSAGES). Note that the message is sent when the key is <u>lifted</u>, not pressed, for reasons described below.

Keylist must be created in the Overlay to work (see Section 14: USING THE OVERLAY). It is best understood by looking at its arguments. For example, `[keylist a start]` would trigger a `start` command when the 'a' key is pressed.

The creation argument for Keylist works in the following order:

1. The first term is a keyname, typically a character key, ie. 'e' (number keys and modifier keys like Shift Control don't work well);

2. An optional flag `**` (described below);

3. Any command of any length (ie ' `:x 3 5 6` ').

In addition to this, there are two variables that may be included anywhere in the command list:

1. The $ sign takes a value from Keylist's inlet. A number atom can be placed on the overlay for this purpose if you wish;

2. The % sign waits for numerical key input while the trigger key is depressed. For instance, with a Keylist object called `[keylist c :x %]`, pressing and holding the 'c' key, pressing 7, and then letting go of 'x' would result in the message ' `:x 7` ', ie. it would open the 7th toggle.

Usually, Keylist will only be active while the mouse cursor hovers over Context, to avoid cross-channel communication. This requirement can be overriden with the `**` flag, Ie. `[keylist c ** :x %]` would work whenever the c key is pressed, regardless of where the mouse cursor is.

18. PRESETS

SUMMARY
- Preset commands are shortcuts to speed up writing complex Context states
- Presets are summoned using a `::` symbol
- Presets are user defined and can be edited in the '`load-presets.txt`' file.

The Context abstraction can be manipulated in many different ways through commands and GUI manipulation, and as a result, one Context can behave completely differently than another. Each configuration of Context is called a "Context state", and consists of a series of commands (see Section 8: CONTEXT COMMANDS (INPUT LANGUAGE)), optionally followed by an overlay state save (see Section 14: OVERLAY SAVING). Commands can achieve any particular state, but typing them out can be just as time consuming as manipulating it by hand using the menu. **Presets** are functions used to speed this up and give the user easier access to commonly used Context states. For example, if you want to access a Context with a loaded Content sample, you can type the command:

```
context :: sample
```

This is much easier than typing the full Context state:

```
:H 12 :X 9 :Y 6 :d 1 :Z 13 :ol 1 13 obj 0 24 content :x2 1 :y1
0.4 :y2 1 :li 1 :lo
```

A preset is indicated by a double colon (`::`), followed by a single identifier argument. It can be given as creation arguments or as regular commands (see Section 8: SENDING COMMANDS).

The presets are stored in the 'load-presets.txt' file in the main Context folder. This file comes populated with a few fewquently used options, including larger Contexts, but it is also an editable file, so you can add your own shortcuts. The first term of every line is an identifier symbol, and the rest of the line is the desired Context state. Make sure you finish every line with a semi-colon. The Context state will be saved to the patch in its long form, so there is no danger of reducing compatibility by creating your own presets.

➢As well as the 'load-presets.txt' file, 'burst-presets.txt' is also editable. Use this file to create your own presets for the burst, besides 'All', 'Random' and 'Sequence'.

BUG: The first Context you create in a patch will not respond to presets. There has to be at least one other Context open somewhere in order for them to work.

19.   MISCELLANEOUS

FLOAT CONNECT LOCALITY

Like the [send] object, a float connect sends through a specific channel without specifying which objects will receive the message, if any. This can lead to unexpected interference: if more than one context share the same ID number, they will both fire. Context ID numbers are assigned automatically so as to not repeat within a given network, so this generally isn't a problem within a single network, but the trouble begins when two patches are open at the same time, both containing context ID number 1. In this scenario, the two patches might interfere with each other every time a float connect 1 fires.

The solution to this problem is the `:J` setting. With this on (it is on by default), a toggle float will only send to a context of the same ID number in the same patch, ignoring those in any other patch.[16] If you have a network which spans several patches, you can turn `:J` off to allow contexts to communicate with each other.

---

16 Note that it uses the $0 variable to achieve this—with __ on, a toggle float will send a message to '$0-ID'; whereas with __ off, it will send simply to 'ID'.

OUTLET MESSAGES

If you connect a [print] object to the outlet of a context, you will probably wonder what the text it prints means and why there is an exclamation mark in it. The short answer is that an exclamation mark (followed by anything) starts a context, and you don't need to worry about it. The long answer is that is has to do with relative message variables, as follows:

When a toggle fires, context sends its database message to the send channels, and the same message, prepended by an exclamation mark, to the outlet or float connect. The exclamation mark starts the next context as normal, but not before it saves the rest of the message. This saved message is used by the ' ! ' variable to create relative messages.

If you want to use the outlets and get rid of the exclamation mark, you can use the director <o at the beginning of the database message (see Section 12: MESSAGE VARIABLES (OUTPUT LANGUAGE)). This can easily be written in the message prepend fields (see Section 9: MESSAGE PREPEND AND APPEND).

20.   PROBLEMS / LIMITATIONS

LOAD TIME

Context is a large program. It has been built with efficiency in mind, and every care has been taken to ensure that a large number of Contexts can function together without taxing the memory of CPU[17]. The drawback, however, is load time. Each Context takes a fraction if a second to load, due too the integrated state loading and heavy dynamic patching procedures It follows on initiation. This has two undesirable consequences:

OPENING PATCHES

Larger patches can take a very long time to open. 'A long time' means as much as 5 minutes, and presumably more for even larger patches. The biggest load hogs are custom overlay saves—especially embedded Content objects with saved arrays (see Section 14: OVERLAY SAVING). This is followed by Contexts with a large number of toggles, each of which has to be created individually on loading. Therefore, if you want to cut down on load time, you should keep overlay time and large Contexts to a minimum.

A partial solution to the problem of load times is to turn DSP off while loading heavy patches. This seems to cut the load time in half. A long load time is much more acceptable than having a long save time, and hence it is considered to be an operational cost rather than a bug. Nevertheless, it is hoped that the load time can be optimized in the future, simply to make Context more convenient.

CREATING NEW CONTEXTS

A similar and more immediately frustrating problem is that creating new Contexts in a patch will lead to audio drop-outs. This effectively rules out the possibility of "live coding" a Context network during a performance, which seems like a very high cost.

There are two immediate solutions to this problem:

1. You can create a large stack of unused Contexts and then drag them on screen instead of creating new instances. This is crude, but effective.

2. You can run the Context network on one CPU core and the audio on another. The [pd~] object is built for this, although I still experienced drop-outs when I tried it. What does work is running two instances of PD and then using [netsend] and [netreceive] to communicate. You can then

---

17 Preliminary tests show that it is possible to create several hundred Contexts without it having noticeable effects on the system.

create new Contexts in the "master" PD instance without hearing any adverse audio effects. Of course this restricts the use of [content], since you can't send signal data over the network. But there is nothing stopping you from having a few more Contexts in the "client" instance and loading samples there.

Despite the workability of these solutions, finding a way to load Context that doesn't lead to audio drop-outs is a top priority for the future development of Context.

NOTE ON AUDIO DROP-OUTS

Besides creation, there are a few other processes that create audio drop-outs:

1. Deleteing Contexts (see DELETING CONTEXTS below);

2. Re-sizing Context (ie. creating or destroying new toggles), which is a real pain, since this seems like a desirable thing to do during a performance;

3. Adding and removing tags, since this triggers changes in the send and receive channels (see Section 7: ID AND CHANNELS);

4. Moving and re-sizing [content] arrays (see Section 17: CONTENT);

5. When a [content] array is embedded, returning from the menu to the main GUI face (see Section 8: SENDING COMMANDS).

Until solutions are found, these actions should be considered as part of the setup and design of a network, and not a good option for performances.

TEMPO / BPM

is a highly decentralized system, with no Context having absolute authority over another. This offers a great deal of flexibility when designing a network, and hence is considered to be a strength. However, it does come with the drawback that setting and changing tempo (BPM) is much harder than on other systems, since every Context has its own time cycle. It is possible to change the tempo globally, using the `:te` command. `:te` defaults to 100 and works as a simple percentage; raising the number increases the cycle time by that much, and hence slows Context down. (See tempo_change.pd for a working example of this). Unfortunately, there is no easy way to make Context work in terms of BPM, without performing calculations and rules.

DELETING CONTEXTS

Deleting Contexts has several hazzards associated with it:

1. It leaves a hold in the network which might not be automatically filled

2. It tends to create audio drop-outs

3. It will sometimes clear some of the internal text files that other Contexts are depending on.

Number 3 is a bug and will hopefully get fixed one day. In the mean time, saving and re-opening the patch should restore the normal behaviour. To avoid the problem, try deleting Contexts one by one rather than in batches.

ZOOMING AND RESIZING

Since PD 0.47, it is possible to zoom in on the canvas through the menu. Unfortunately, this will render most of the Cotnext GUI features dysfunctional, since they depend on x-y coordinates which are not updated with the zooming. Context will continue to play, but it won't interact with the mouse the way you expect.

Likewise, resizing a Context object with the native PD edit-mode resize arrow is bad news,

Context has a way of resizing and its own zoom function, which you should use if you want more visibility. See Section 4: RESIZING CONTEXT.

21. TROUBLESHOOTING

PROBLEM	SOLUTION
There are "… can't create" messages all over the console when Context loads.	1. Do the uncreated objects look like normal externals? Then those externals aren't being found by your system. Try declaring zexy before creating Context. All other libraries are declared through the [library/object] method, so make sure that the paths to your externals folder is declared. 2. Do the unreated objects look like custom Context abstractions? (ie. 'am.pd', 'pf0.pd'…). If so, then make sure that the main Context folder is declared.
The symbol atom at the top disappears when I enter the properties menu.	This happens because the text is too bold. Open PD with a special flag "-font-size normal", or add the same text to the startup flags (Edit → preferences → Startup → Startput Path).
I can't get the toggle floats to scroll with the mouse.	This problem is specific to Cyclone version 0.3. Download another version of Cyclone and it should be fine. Later releases should work, and the old v 0.0.extended versions work also.
I found another problem which I think might be a bug…	Email liam.goodacre@gmail.com

22.   COMPLETE SCHEMATIC OF CONTEXT PROCESS

**INPUT
(INSTANTANEOUS)**

INLETS   RECEIVE CHANNEL   GUI
MANIPULATION   MENU   CREATION
ARGUMENT

SAVING

INTERPRET
(INPUT LANGUAGE)

INCOMING RULES
(RULE LANGUAGE)

SEQUENCE

messages

DATABASES

OUTGOING RULES
(RULE LANGUAGE)

connections

OUTPUT
LANGUAGE

OUTLETS   FLOAT CONNECT

'start'

SEND CHANNEL

<

**OUTPUT
(TIMED)**

CONTEXT
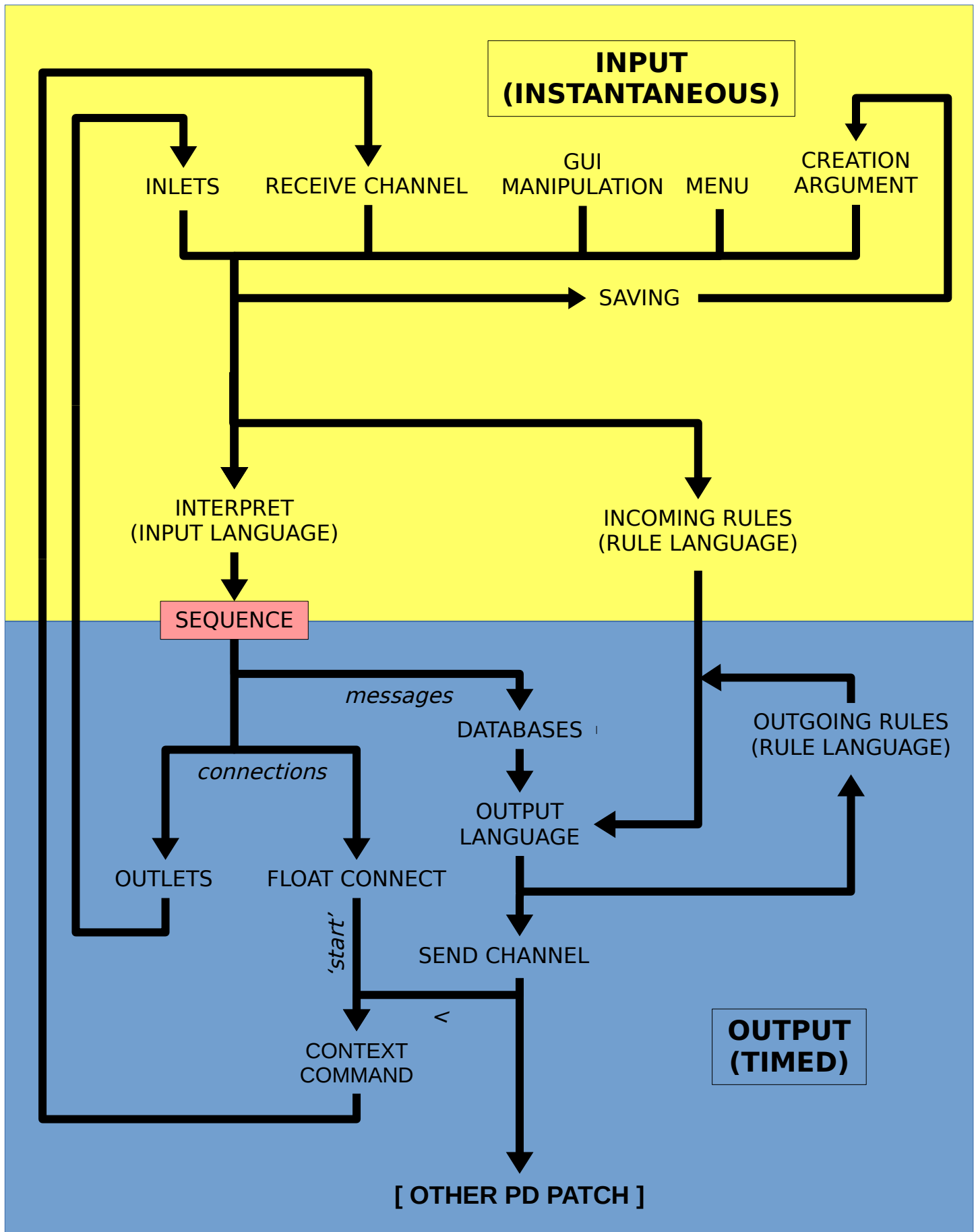COMMAND

[OTHER PD PATCH]

Illustration 30: Context schematic

23. GLOSSARY OR TERMS

Note: This glossary does not name all of Context's features, but instead defines the custom vocabulary surrounding Context and Context networks as used in this manual.

BANK: A storage bank that is used to store and recall multiple Context states.

BURST: A set of toggles, by default the y-axis toggles, which all fire together at the end of the cycle. The burst is typically used to control the logic of a context network, ie. by determining what happens after the PATTERN is finished. There are here are 8 different parameters controlling exactly which toggles will fire.

CONTEXT, A: A single instance of the [context] abstraction, created on a PD canvas.

CONTEXT CANVAS: The coloured area not taken up by TOGGLES in the middle of a context.

CONTEXT COMMAND: An text based instruction sent to context to determine a CONTEXT SETTING. Context commands form a rudimentary language, as detailed in the 'context_commands_all.pd' help file.

CONTEXT NETWORK: A group of contexts connected together, either with PD cable connections or FLOAT CONNECTIONS. Context networks are used to store compositions.

CONTEXT SETTING: An parameter which changes context's behaviour in some way. Settings can either be set by a CONTEXT COMMAND, the GUI or by the MENU. They are detailed in the 'context_commands_all.pd' help file.

CONTEXT STATE: A particular configuration of CONTEXT SETTINGS assigned to a context, given and saved as a series of CONTEXT COMMANDS.

CURSOR: The timeline indicator which moves across the CANVAS, tracking the progress of the CONTEXT CYCLE.

CYCLE / CONTEXT CYCLE: The main process which a context executes, namely playing the PATTERN followed by the BURST. The time taken for this process, the CYCLE DURATION, is often referred to simply as the CYCLE.

DATABASE: A collection of messages which are stored and played on output. Each context has two databases, the PATTERN DATABASE and the BURST DATABASE, which operate identically but are acceded by the two different TOGGLE ARRAYS.

EMBEDDING: Placing other objects, typically the [content] abstraction or another [context], in the OVERLAY and hence on the CONTEXT CANVAS.

EXCEPTIONS / EXCEPTION LIST: A list of user specified terms (typically CONTEXT COMMANDS) which will be rejected as INPUT.

FIRE: When a selected TOGGLE is triggered by the CONTEXT CYCLE and sends a MESSAGE.

FLOAT ATOM: The number box found at the top-left hand corner of every context, left of the SYMBOL ATOM, used to determine the CYCLE TIME.

FLOAT CONNECTION: A positive TOGGLE FLOAT, which is used as an alternative to a PD cable to form a connection between two contexts. Float connections correspond to ID NUMBERS.

GUI (Graphical User Interface): The main face of context seen and interacted with by the user, comprising the FLOAT ATOM, SYMBOL ATOM, X-AXIS TOGGLE ARRAY, Y-AXIS TOGGLE ARRAY, CURSOR, and CANVAS.

ID: A mandatory number and optional symbols assigned to a context to distinguish it from others, and to aid in INPUT and OUTPUT. Each ID term creates one dedicated send and receive channel.

INLETS: The PD inlets found at the top of a context. Inlets are used to receive CONTEXT COMMANDS and to interconnect contexts to form a CONTEXT NETWORK. In contrast to the PD convention, all context inlets are identical and equivalent to each other.

INPUT: Any message received by context, typically a CONTEXT COMMAND. Input is received through the INLETS, through a FLOAT CONNECT, or through the ID RECEIVE CHANNELS.

LAST TOGGLE: The toggle on the bottom-right corner of any context, which acts as a special point between the pattern and the burst.

MENU: The alternative context GUI face, used to access most CONTEXT SETTINGS. The menu is opened by right clicking on context and choosing 'properties'.

MESSAGE: This one word means (too) many different things in this manual. In general, there are two types of messages in Context: Incoming messages, which are used to control Context, and Outgoing messages, which Context generates and uses to control other things.

MESSAGE (INPUT / COMMAND): The same as a CONTEXT COMMAND.

MESSAGE (OUTPUT / CONTEXT / DATABASE): Any message stored in the DATABASE and sent as OUTPUT by the PATTERN or the BURST.

MESSAGE VARIABLE: A special character used in an OUTPUT MESSAGE that is interpreted by context in a particular way. Message variables form a rudimentary language, as detailed in Section 12: MESSAGE VARIABLES (OUTPUT LANGUAGE).

OUTLETS: The PD outlets found at the bottom of a context. Context outlets correspond individually to the x-axis toggles, and are synched to its TOGGLE ARRAY, by default the PATTERN ARRAY. They are used to interconnect contexts to form a CONTEXT NETWORK.

OUTPUT: Any message, or collection of messages, sent by the context BURST or PATTERN to an outside object, through a RECEIVE CHANNEL, or the context OUTLETS.

OVERLAY: An internal sub-patch within context which is accessible, and saveable, from the parent patch. The overlay can be used for two things: EMBEDDING custom objects on the CONTEXT CANVAS and HACKING.

PARENT PATCH: The main patch that you work in to create a Context network.

RANDOM SERIES: A series of numbers (:d values) which are allowed by the random settings (:q and :D) as possible cycle times in random mode (:E 1).

SEND / RECEIVE CHANNEL: The argument of a normal [send] or [receive] object, ie. [send foo]. Each context has its own send and receive channels corresponding to its ID, which are used for sending INPUT and receiving OUTPUT. In this documentation, SEND CHANNEL means a channel that context sends, and RECEIVE CHANNEL means a channel that context receives.

SEQUENED MESSAGES: MESSAGES sent from the MESSAGE DATABASE in a timed pattern, according to the PATTERN or BURST. Typically, sequenced messages are musical events that will be interpreted or played by other PD objects external to context.

STORAGE BANK / BANK: A database within context that records and retrieves various different CONTEXT STATES. The bank can be used like the bank on a step-sequencer to store multiple patterns in parallel.

SYMBOL ATOM: The text box found at the top-left corner of every context, right of the FLOAT ATOM, used to view and edit the MESSAGE DATABASE.

TAGS: The optional symbol component of the context ID.

TERM: A list / line of text from a text storage object.

TOGGLE: An individual toggle box found on the x- or y-axis.

TOGGLE ARRAY: The set of all toggles on the x- or y-axis (ie. The PATTERN or the BURST). Sometimes, TOGGLE ARRAY refers to only those toggles in the array which are currently selected.

TOGGLE FLOAT: An arbitrary number assigned to an individual toggle, distinct from the state of that toggle (ie. 1, 0). There are two types of toggle float: positive and negative. Positive toggle floats are FLOAT CONNECTIONS; negative floats are used as a MESSAGE VARIABLE.

TOGGLE FLOAT ARRAY / FLOAT ARRAY: The set of all toggle floats on the x- or y-axis.

24. LICENSE

Context is Licensed under the GPL v3. See 'license.txt' for more details.