# Chino – a framework for scripted meta-applications

**David ADLER**
david.jo.adler@gmail.com

## Abstract

Chino is presented, a framework for creating meta-applications from Linux audio and Midi tools. It provides command line options to create or open sessions, a runtime user interface for adding, restarting or removing applications and a hand-editable file format to which sessions are saved. Graphviz is used to optionally display the layout of a session.

Chino itself is a Bash script that just provides generic functionality, users can create presets to implement what is desired for their use cases. Presets are prototypes for sessions, multiple sessions can be derived from a preset.

A preset is made up of a number of applications, each defined as a program together with its usage. For every application, the preset contains required application files and a library file that, via variables and functions, defines how the program is to be started and interconnected. Defining applications together with their connections results in dependencies, which are tied via user-defined port-groups.

In this paper, we will explain the architecture of Chino and take a look at some implications and limitations of this session management model.

## Keywords

Linux audio, Bash, session management

## 1 Introduction

Chino[1] is yet another approach to session management for Linux audio. It is geared towards applications using the Jack Audio Connection Kit (Jack) for audio and either Jack or the Alsa Sequencer for Midi.

The modularity of UNIX/Linux software is proverbial and usually well received. In the realm of Linux audio, however, this appreciation hat its limits, as manually restoring modular sessions quickly becomes prohibitively complex. Consequently, users often complain about the lack of a comprehensive session management system.[2]

Regardless of complaints, some progress is taking place. Jack Session, the LADCCA/LASH/LADISH lineage and the Non Session Manager are currently coexisting[3] and the number of applications supporting one or more of them is steadily increasing.

Those session managers are capable of storing and restoring an arbitrary setup, as long as the applications involved are supported in some way. A Chino session, in contrast, is restricted to a limited number of setups prepared by the user via presets. No support for a protocol by applications is required, any application capable of restoring a previous state by command line options and/or file loading can be used.

Once a preset is prepared, usage is dead simple. No manual connection making is involved and all files belonging to a session are automatically placed in an ordered manner below one base directory.

In section 2 we will go through some underlying concepts, followed by a description in section 3 of how they are implemented. Those two sections together explain the architecture of Chino; providing useful knowledge for creating custom presets. While we can create sessions based on an existing preset without such a level of understanding, restricting oneself in that way neglects one of the main features of Chino, which is customisability. In section 4, eventually, we will take a look at some of the implications and limitations of the presented design.

[1] The application, online documentation and an example preset are available from `http://www.chino.tuxfamily.org`.

[2] In February 2013, Dave Phillips started a thread on the Linux Audio Developer (LAD) mailing list with the subject line *"So what do you think sucks about Linux audio?"* that pretty much confirms existence of those complaints. `https://lists.linuxaudio.org/mailarchive/lad/2013/2/5/196481`

[3] Dave Phillips' article *"A brief survey of Linux audio session managers"* from January 2013 on LWN.net gives a good overview. `https://lwn.net/Articles/533594/`

## 2 Concepts

This section covers a number of concepts underlying the design of Chino, without going too much into detail. Having these concepts in mind will aid us in understanding the subsequent section covering implementation.

### 2.1 Sessions

While Chino does manage *sessions*, the term session management is somewhat misleading. As stated in the introduction, Chino lacks the ability to just save and restore any setup involving supported applications. Sessions in Chino might be better described as "instances of a meta-application" or as "patch files" to which the meta-application saves its state.

### 2.2 Presets

A *Preset* defines the meta-application of which the sessions are instances. Chino, the core script, only provides generic functionality. Whenever running a session, it needs to be pointed to a preset. Figure 1 shows the relations between Chino, presets and sessions.

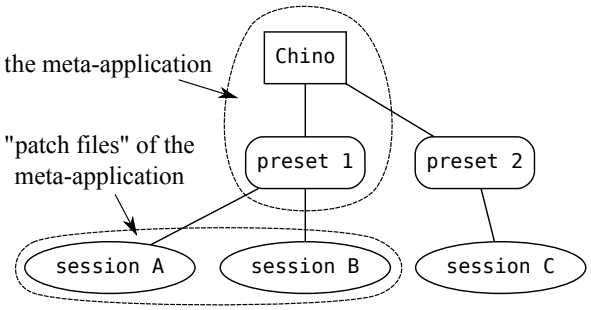


Figure 1: A preset—together with Chino—can be viewed as a meta-application. Sessions derived from that preset then are the "patch files" saved by the meta-application.

It is up to the user to create one or more presets in order to cover desired use cases, a default preset is provided as an example or starting point.[4]

### 2.3 Applications

A preset defines a set of *applications*. An application—in Chino—consists of three things:

1. the actual *program* used (like amSynth or Pure Data);
2. *application files* belonging to the program (patch files, configuration files and the like, if any);
3. an *application library*, a text file in Bash syntax defining how the program is to be started and interconnected.

While the programs—of course—are installed to the operating system, application files and application libraries are part of the preset.

One program can serve as several applications. We could define two applications `amssynth` and `amsfilter`, both using the program AlsaModularSynth—in one case as a synthesiser and in the other case as a filter.

Henceforth, we will use that distinction between the terms *program* and *application* throughout this document.

### 2.4 Methods

Applications are grouped into *methods*, categories for applications that can be handled in similar ways. Methods are defined in *method libraries* that are also part of the preset.

Two *method types* are hard-coded into Chino: *unique methods* and *channel methods* (see figure 2).

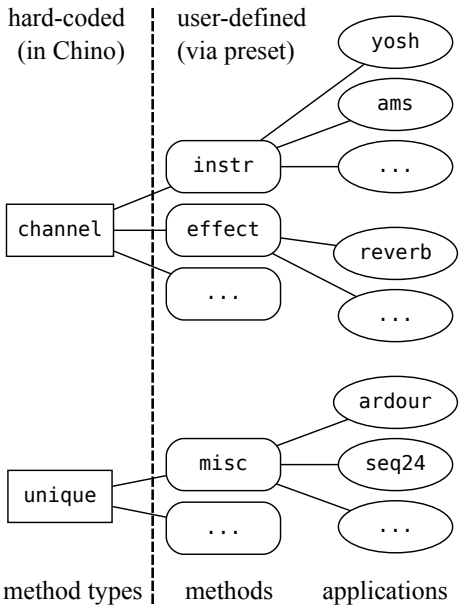


Figure 2: Method types, methods and applications.

- For **unique methods**, application names are assigned to a consecutively filled array of variable length. Entries must be unique, thus the name.

[4] The default preset is documented on `http://chino.tuxfamily.org/preset.html`.

- For **channel methods**, application names are assigned to indices of a fixed-sized array, where indices may be left empty. Applications need not occur uniquely within a channel method. An obvious use case (though not the only one possible) is using the index to connect an application to a certain audio or Midi channel, thus the name.

Introducing that extra layer of methods has two main advantages. First, methods simplify adding support for applications to a preset, as it is often the method library that does most of the work. Secondly, having channel methods allows for some desirable flexibility in arrangement of the connection graph.

### 2.5 Templates

In addition to pointing to a preset, we may optionally point to a *template session*. Any session derived from that same preset can serve as a template. To illustrate this, figure 3 shows a version of figure 1, modified to include a session that points to a template.

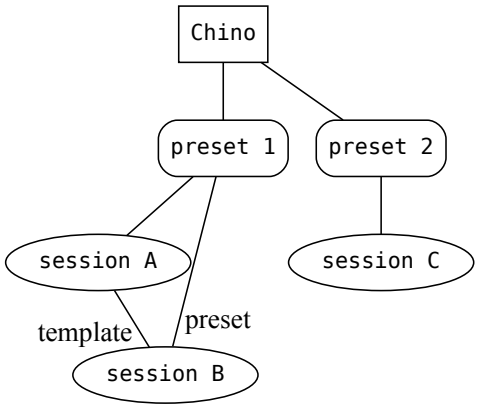


Figure 3: `session B` uses `preset 1` as a preset and `session A` as a template.

### 2.6 Inheritance...

#### 2.6.1 ...of application files

On running a session, missing application files are copied from the template or preset, in that order of precedence.

#### 2.6.2 ...of libraries

Libraries are sourced from the local session, the template or the preset, in that order of precedence.

Per default, application libraries will remain with the preset. Unlike application files, they will not get copied to the local session.

For custom application behavior on a per-session base, an option exists to "localise" a library. For making a session self-contained, an option exists to localise all used libraries. Self-contained sessions point to themselves as a preset.

Libraries local to the template, however, will get localised automatically; to not disrupt matching pairs of application files and application libraries for future child sessions.

### 2.7 Session hierarchy

Since applications are defined together with their connections, they may *depend* on other applications *providing* for certain ports to connect to. This leads to a hierarchical session layout, a dependency tree.

Ports provided by the sound card and Midi hardware (those devices are made applications as well) will usually form the root of the hierarchy. More layers can then be added to form a virtual studio to the user's liking. A mixer might for instance form the layer on top of the hardware ports, instruments and effects can then be connected to the mixer.

Applications do not depend on other applications, they rather depend on or provide for user defined *port-groups*. A port-group is just a name for a set of *port-variables* to which the real Jack/Alsa port names are assigned.

That way, applications providing for the same port-group can be exchanged without breaking the session. We could define two applications `seq24` and `nonseq` both providing for a port-group `SEQ`; then either can be used as a sequencer without making any further changes to the session.

Just as applications, methods may also depend on and and provide for port-groups. In the context of dependencies, we will sometimes use the term *nodes* when referring to anything that can depend or provide, i.e. either methods or applications.

Dependencies are handled separately for audio and Midi, so the place of a node in the dependency tree is defined by four lists of port-groups: *audio depends*, *Midi depends*, *audio provides* and *Midi provides*. Collectively, we will refer to them as *anchors*. We can make depends optional by prefixing them with a colon, this just suppresses the warnings otherwise displayed for unsatisfied depends.

Figure 4 shows the way a node is represented

in the session graph (or dependency graph) that Chino displays using the Graphviz software, figure 5 shows the graph of a small session.

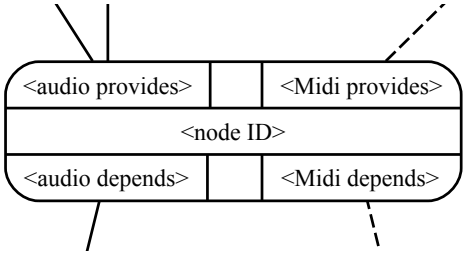


Figure 4: Representation of a node and its anchors in the dependency graph. Solid lines represent audio dependencies, dashed lines represent Midi dependencies.

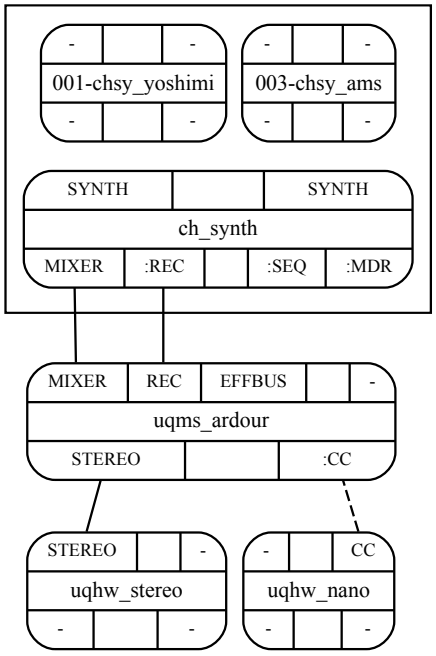


Figure 5: A session graph. Three unique method applications are started: `uqhw_stereo` (a stereo sound card), `uqhw_nano` (a Midi controller) and `uqms_ardour`. The channel method `ch_synth` has anchors, so it gets its own node in the graph and its applications `chsy_yoshimi` and `chsy_ams` drawn inside a cluster.

## 3 Implementation

Chino is written in the Bash scripting language. This section goes into some detail on how it is implemented.

### 3.1 Names

Rather strict rules exist regarding names of sessions, methods, applications and port-groups. The rules are ruthlessly enforced, as those names are part of file paths, variables or functions. Allowing special characters would just not work and would be potentially harmful.

- *Session names* may only consist of letters, numbers and underscores.
- *port-group names* may only consist of letters and numbers. They must be unique within each connection type (audio and Midi).
- *Method names* may only consist of letters and numbers. The first two characters of a method name must be unique within each method type, those characters make up the *method acronym* (consequently, method names must be at least two characters long). *Method IDs* are the method names prefixed with `uq_` for unique methods and `ch_` for channel methods.
- *Application names* may only contain letters and numbers, names must be unique within their method. *Application IDs* are the application names prefixed with `uq<method_acronym>_` for unique methods and `ch<method_acronym>_` for channel methods.

Given that nomenclature, all method IDs and application IDs will be unique strings within the set of all methods and applications.

### 3.2 The session directory tree

To facilitate automated copying of application files on creating or expanding a session, we must make sure the directories and files of a preset are named according to some rules.[5]

When running a session, its directory tree will automatically get filled and supplemented with files from the preset or template. At that point, thus, we don't need to meticulously follow naming rules anymore—Chino does it for us.

The base directory of a preset or session holds one subdirectory `<application_ID>` for every application. For presets, that is one for every implemented application; for sessions, that is one for every application that is or ever was part

[5]Naming rules are comprehensively covered in the online documentation. `http://chino.tuxfamily.org/documentation.html#file`

of the session (obsolete files do not get deleted by Chino, as they might not be obsolete from a user's perspective). Those application subdirectories hold the application files; applications not needing files do not get such a directory.

The base directory of a session also holds the *session definition file* `<session_name>.sdef` to which Chino saves the session.

### 3.3 Libraries

A `libs` directory, below the base directory of a preset, contains all libraries.

A file `libs/<session_name>-listlib` is the "root-library" of a preset. It holds arrays of allowed methods and applications and sets the initial array sizes for channel methods.

Methods and applications listed in `listlib` are each defined in their own library file, method libraries or application libraries respectively.

#### 3.3.1 Variables in libraries

Via variables, all libraries define anchors of the node they represent.

Method libraries additionally have a variable that allows to give them a custom option in the rutime user interface. In the default preset, the `ch_dssi` method uses that feature to let users split the configuration of ghostess into the the parts belonging to the single DSSI plugins.

As shown in the example below, application libraries additionally have variables defining a number of properties: whether the application comes with application files; whether a function is required for adapting application files to their new names after copying; what Midi system to use and whether to disconnect autoconnected ports.

```
FILE_uqms_seq24='true'
MOVE_uqms_seq24=''

APRO_uqms_seq24=''
ADEP_uqms_seq24=''
MPRO_uqms_seq24='SEQ'
MDEP_uqms_seq24=':KBD :CC'

MIDI_uqms_seq24='alsa'
AUTO_uqms_seq24=''
```

#### 3.3.2 Functions in libraries

Via functions, methods and applications accomplish the rest of the work; like a solid-state Bash session script ripped into pieces, those pieces then being called on demand by Chino.

Method libraries must provide a number of mandatory functions; requirements for application libraries are largely determined by their method.

If an application or a method has any anchors defined, this triggers the requirement for additional functions to be present. Details on those functions will follow in sections 3.6.3 and 3.6.4.

### 3.4 Steps

To illustrate what *steps* are about, it is useful to picture the "lifecycle" of an application (not in terms of code development but in terms of running the application).

When manually running an application, the single steps to be accomplished will be something along the line of:

- starting the program;
- establishing audio connections;
- establishing Midi connections;
- making music (the "tweak-and-save loop");
- quitting the program.

The attempt to adapt those steps to match the requirements of Chino led to the following list of steps:

- `assign` for assigning an application to an array and sourcing application libraries;
- `check` for checking whether an application file is present, if applicable;
- `list` for displaying a summary to the user;
- `copy` for copying and renaming the application file if the above check was negative;
- `start` for starting the program, includes assignment of port-variables;
- `acn` for establishing audio connections using the assigned port-variables;
- `mcn` for establishing midi connections using the assigned port-variables;
- the "tweak-and-save loop"(not a step);
- `unassign` for unassigning and killing an application when removed using the runtime user interface, or `kill` for killing an application on quitting the session.

For each of those steps, a method library must provide a so-called *step function* `s_<method_ID>_<step_name>()`. Chino simply calls the appropriate step function from the method library whenever a step needs to be done for one of the applications belonging to that method.

### 3.5 Tasks

A *task* is just a series of steps that accomplishes something useful.

Tasks may be vertical, calling a number of steps for one application, e.g. for adding an application to a session or for restarting an application (see table 1).

| step | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| assign | | | s2 | |
| check | | | s3 | |
| list | | | s4 | |
| copy | | | s5 | |
| start | | | s6 | |
| acn | | | s7 | |
| mcn | | | s8 | |
| unassign | | | s1 | |
| kill | | | | |

Table 1: Illustration of a vertical task. An application a3 is restarted in a session consisting of applications a1 to a4. The task runs steps `s1` to `s8`.

Tasks may be horizontal, calling one step for all applications, e.g. for re-establishing all audio connections (see table 2).

| step | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| assign | | | | |
| check | | | | |
| list | | | | |
| copy | | | | |
| start | | | | |
| acn | s1 | s2 | s3 | s4 |
| mcn | | | | |
| unassign | | | | |
| kill | | | | |

Table 2: Illustration of a horizontal task. Audio connections are re-established for all applications a1 to a4. The task runs steps `s1` to `s4`.

Tasks may be both horizontal and vertical, e.g. for opening a session (see table 3).

Tasks are part of Chino, so users won't be bothered with them. The appropriate tasks get called whenever a session is opened or closed or when applications are added, restarted or removed.

### 3.6 Helper functions

*Helper functions*, prefixed with `h_`, exist for every step except `unassign`. Helper functions are never called by Chino itself, they are just tools

| step | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| assign | s1 | s2 | s3 | s4 |
| check | s5 | s6 | s7 | s8 |
| list | s9 | s10 | s11 | s12 |
| copy | s13 | s14 | s15 | s16 |
| start | s17 | s18 | s19 | s20 |
| acn | s21 | s22 | s23 | s24 |
| mcn | s25 | s26 | s27 | s28 |
| unassign | | | | |
| kill | | | | |

Table 3: Illustration of a task that is both horizontal and vertical. A session consisting of four applications a1 - a4 is opened. The task runs steps `s1` to `s28`.

available to accomplish steps in standardised ways; to be called from the step functions inside the method libraries.

The design of Chino attempts to find a reasonable balance between being as generic as possible and making implementation of presets as simple as possible. The combination of methods and helper functions serves that goal.

In most cases, using helper functions is desirable, resulting in a method doing barely more than calling the appropriate helper functions with appropriate arguments inside its step functions, as seen in the following example.

```
s_ch_synth_copy()
{
    declare -ri chan=$1
    h_copy ch_synth $chan
}
```

Nevertheless, it is also desirable to have the freedom of not using helper functions or using them in non-standard ways, to adapt to non-standard use cases. Three of the methods implemented in the default preset are non-standard in that sense:

- the unique method `hw`, for hardware devices, neither using `h_copy` nor `h_start()` as those are not applicable to hardware;
- the channel method `dssi`, which uses one instance of ghostess to harbour all DSSI plugins assigned as applications to its channels;
- the channel method `senv` (from **s**ynthesis **env**ironments) for instruments capable of multi-channel audio and Midi, in which the program is started just once for all instances of the same application.

A few helper functions deserve a closer look, since they implement essential functionality within Chino.

#### 3.6.1 `h_assign`

`h_assign()` sources libraries, thereby implementing the inheritance rules for application libraries described in section 2.6.2.

#### 3.6.2 `h_copy`

`h_copy()` copies application files, thereby implementing the inheritance rules for application files described in section 2.6.1. Application files may be single files or directories containing files.

#### 3.6.3 `h_start`

`h_start()` starts the program and calls the appropriate *assignment functions* from the method libraries and application libraries.

In the assignment functions, we assign real Jack/Alsa port names to the appropriate port-variables, a prerequisite for later having the connection graph established.

For both anchor types—audio and Midi—a respective assignment function is required whenever the node has any anchors of that type.

To illustrate how `h_start()` works, let us look at a case where it gets called for an application that has audio anchors, thereby triggering the necessity for Jack audio port assignment.

1. a snapshot of audio ports is taken via `jack_lsp`;
2. an `<application_ID>_start()` function gets called from the application library, starting the program;
3. After the new ports have appeared, another snapshot of audio ports is taken.
4. Two newline separated lists—of new Jack audio input and output port names—are retrieved from a diff of the two snapshots;
5. The appropriate *assignment function* from the application library is called with the two lists as arguments.

Accordingly, new Jack-Midi and Alsa-Midi ports are retrieved and passed to the functions for assignment.

In the case of programs with many ports—like a mixer with inputs, outputs, sends and returns—it is helpful to make up a suitable naming scheme for ports within the application, then using tools like `grep` or `sed` for variable assignment.

#### 3.6.4 `h_acn` and `h_mcn`

`h_acn()` and `h_mcn()` call one *connect-function* for each depend of the method and one for each depend of the application.

In those connect-functions, we must establish connections using the port-variables assigned during the `start` step.

Two functions are available to aid in establishing connections:

- `msaudioconnect()` for mono/stereo-agnostic audio connections.
- `ajmidiconnect()` for Alsa/Jack-agnostic Midi connections. Whenever required, Chino will launch `a2jmidid` to facilitate translation.

Both functions require certain suffixes being part of the port-variable names, documented in detail in the comments inside the libraries of the dummy preset that comes with Chino for documentation purposes.

If a depend is unsatisfied—i.e. either not provided or ambiguously provided—no attempt will be made to establish connections.

Connect-functions are not exclusively called during the `acn` and `mcn` steps. The tasks for adding or restarting applications will, after having completed all steps, call connect-functions for all nodes depending on newly provided port-groups. That way the connection graph is kept sane regardless of application launch order.

### 3.7 User interface

The user interface consists of the configuration file, command line options and arguments, the hand-editable session definition file and a runtime user interface.

To give an examplary session definition file, here is what a file defining the session shown in figure 5 would look like.

```
NAME=graph
PRESET=/path/to/preset:preset_name
UQMETHS=hw msc
uq_hw=stereo nano
uq_msc=ardour
CHMETHS=synth
ch_synth-CH-001=yoshimi
ch_synth-CH-003=ams
```

Via command line, new sessions can be created and existing ones can be opened. For convenience, some more options exist: for writing a prototype session definition file and for creating new libraries by using existing methods or applications as a template.

Whenever running a session, its base directory must be the present working directory. For creating a new session, at least a session name must be given.

```
$ chino -n name_of_session
```

To open an existing session, we point Chino to the session definition file.

```
$ chino -o name_of_session.sdef
```

Once a session is running, the runtime user interface offers keybindings to add, remove or restart applications, to re-establish connections, to localise libraries, to check dependencies, to display the dependency graph and to save the session state. The latter will only save the current setup, state of the involved applications needs to be manually saved to the appropriate application files.

## 4 Conclusions

### 4.1 Field of application

Due to said differences, Chino does not so much compete with the other session managers. Being a command line tool that requires some editing of Bash scripts for customisation, it will certainly not fulfil the desire of many users for a comprehensive session manager with a graphical user interface.

Chino just attempts to fill a niche in the ecosystem of session managers by embracing modularity and customisability. One of its strong points is the use of presets and templates, although other session managers also offer features in that direction.

Not being able to store any setup clearly is a disadvantage, though one that is somewhat mitigated once the following assumptions are made:

- Any one user will only use a small subset of all possible setups;
- The user will use that subset repeatedly.

Admittedly, those assumptions do not apply to someone in the phase of exploring the variety of Linux audio applications. For someone who is comfortable with Bash and knows which set of applications to use for what purpose, however, Chino might be a convenient tool.

### 4.2 Session portability

Sessions turn out to be somewhat portable. Limitations that come to mind are:

1. the session must either be self-contained or its preset must be present on the host system;
2. we might run into incompatibility-issues when versions of programs are mismatching;
3. hardware requirements of the sessions, like audio channel counts, must be met;
4. depending on the programs used, matching sample rates might be required;
5. hardware applications might need to be adapted to Jack/Alsa port names of the host system;
6. program behaviour might differ due to local configuration files.

While points (1) to (4) are mere facts, point (5) can be resolved by agreeing on a naming scheme for port-groups the hardware applications provide for, host systems can then use their own hardware applications.

Point (6) can be mitigated if applications make use of as many command line flags as possible, to override local settings. If the program allows to specify a configuration file on the command line, we can include one as part of the application.

### 4.3 Known issues

Due to the fairly small user base (consisting of just the author himself), this list is most likely incomplete.

- Establishing connections takes rather long for large sessions.
- It takes time and care to build a preset (though once that is accomplished, Chino doesn't get in the way anymore).
- It's a crude hack still in development.

Given the last point, Chino actually runs surprisingly well.

## 5 Acknowledgements

Sincere thanks go to all Linux (audio) developers, collectively constituting the giant's shoulders upon which this little script resides.

Thanks go also to the entire Linux audio community. Especially the mailing lists have provided some highly educational reading matter over the years.