

RTLisp

This document will be more readable if you resize the window so that the string of dashes below lies entirely on one text line. -----

Intro

RTLisp runs on David Betz's XLisp and is comprised of a set of object class and function definitions which provide a Lisp interface to Paul Lansky's NeXT-based, real-time audio mixing software, `rt`. The `rt` user arranges audio material temporally, dynamically and spatially by means of a relatively simple score grammar which is parsed and executed by `rt`'s audio driver program. **RT** is wonderful in that it allows the composer to process, shuffle and mix concrete sounds interactively and often in real-time. However, the simplicity of `rt`'s grammar can make it tedious to use directly, especially for the realization of complex scores. Also, there is no provision for the programmatic generation of audio events, so every event must be specified in detail by hand.

RTLisp was designed to facilitate the process of `rt` score creation. It equips the composer with a powerful, interactive, high-level programming environment (Lisp), and lends some intelligence to tasks such as temporal placement and grouping of audio events, control of dynamics and `rt` 'track' assignment. **RTLisp** can be thought of as an `rt` score compiler. It can run stand-alone in a shell window, controlling the `rt` audio driver directly, or it can run in conjunction with **rt.app**, Mr. Lansky's NextStep interface to the driver.

For the purposes of this discussion, it is assumed that the reader is familiar with Lisp (Common Lisp in particular) and is at least vaguely familiar with basic concepts of object-oriented programming. It's also assumed that the reader has a working knowledge of the `rt` application.

General

RTLisp is basically built around three object classes: **notes**, **tracks** and **sounds**. **Notes** are translated as audio events ('playnotes' in `rt` parlance) after being assigned properties that include: the soundfile from which the event is extracted, starting time, duration, position in the stereo field, pitch transposition and dynamics. **Notes** are placed in **tracks**, which may

impart dynamic and panning characteristics of their own, and may be shifted in time or selectively enabled or disabled for a given mix. Each **sound** object is associated with a particular soundfile and contains information about that soundfile, such as its format and duration. Each **note** object refers to a **sound** object as its source of audio material.

The **RTLisp** user normally begins a program or session by declaring the soundfiles that will be used in the score, thus creating the sound objects that refer to these files. Then, tracks are created and notes are placed in them. Tracks and notes are specified individually and/or generated programmatically. Finally the **mix** function is called, which resolves note and track synchronizations, determines which notes to play and sorts them by start time, computes amplitude envelopes, assigns physical **rt** audio tracks and produces an **rt** score. This score may be sent directly to the **rt** driver program, to the pasteboard, to the **rt** NextStep interface (rt.app), or piped to any Unix filter.

Sound Objects

The two primary functions associated with sound objects are **sounds-from** and **sound**.

(**sounds-from** "directory1" "directory2" ...)

builds a search path through which to search for soundfiles named with relative paths (no leading /'s or .'s). The directories in the search path are scanned in the order in which they're specified in the **sounds-from** function. The directory names must be quoted. The default search path is ".", the current directory.

(**sound** <name> [[is] "pathname"])

creates a sound object and assigns it to the symbol <name>. If "pathname" is not specified, the actual soundfile name is assumed to be <name>.snd (all alpha lowercase) and is scanned for in the search path specified by **sounds-from**. If "pathname" is given (the 'is' is optional), the search path is similarly checked unless the path begins with '/' or '.', in which case it's taken as absolute. The pathname must be quoted. All sound objects are kept in a list assigned to global variable ***sounds***. Examples:

(**sound** thump) ; creates a sound object named `thump' which refers to soundfile "thump.snd"

(**sound** beep is "Long_soundfile-name.snd") ; caps in pathname necessitate using this form

If the pathname does not include the ``.snd'` suffix, **RTLisp** will supply one. A useful sound function that also applies to note and track objects is **info**:

(**info** beep [**notes**])

displays info on sound object ``beep'`. If **notes** is specified, the display will include a brief summary of all note objects that use the sound object.

(**info sounds**)

displays brief summaries of all declared sound objects. The **dur** function:

(**dur** beep)

returns the duration in seconds of the soundfile associated with ``beep'`. You can play the soundfile associated with a sound object with:

(**play** mysound)

Type ctrl-C to interrupt the playing.

Note Objects

Note objects are the basic audio event units in **RTLisp**. A note can be thought of as a slice of audio data, extracted from a soundfile and altered by applying various modulating parameters and control signals. Notes directly translate into **rt** ``playnote'` commands.

Notes are defined thusly:

(note	:is <symbol>	; tag, required if note is to be edited or otherwise referenced
	:track <track-obj>	; track object to which note is assigned, default = *current-track*
	:at <time>	; start time relative to beginning of track, default = 0
	:sync <rule>	; alternative to :at
	:snd <sound-obj>	; sound object providing audio source data, required
	:skip <time>	; offset into soundfile, default = 0
	:dur <time>	; duration, default = duration of soundfile - skip value
	:transp <control>	; pitch transposition, in semitones
	:pan <control>	; placement in stereo field, 0 = right, 1 = left
	:gain <value>	; amplitude multiplier, default = 1
	:env <envelope>	; amplitude envelope
	:comment "<string>"	
)	

Only the **:snd** parameter is required. All time values are in seconds. An <envelope> item is one that evaluates to a list of time/value pairs. That is, the first number in a pair specifies time relative to the beginning of a note and the second number specifies the amplitude of the envelope at that time. Example: '(0 0 1 0.5 2 1 9 1 10 0)'. As in the **rt** 'playnote' command, envelope time values are interpreted proportionally to the total duration of the note, not as absolute values in seconds. A <control> item may evaluate to a simple numeric value or an envelope. For example, a note may be statically fixed in the stereo field, (... **:pan** 0.2), or it may be panned from left to right over its entire duration, (... **:pan** '(0 1 1 0)). It is suggested that the user always specify amplitude envelopes (**:env**) as normalized (i.e. maximum amplitude = 1.0), and that the actual peak amplitude factor be controlled by the **:gain** parameter. This way, the 'contour' of a note is maintained independently of its actual volume in the mix. Pan and amplitude envelopes are used in conjunction to automatically generate separate amplitude envelopes for left and right stereo channels. The **:is** parameter serves to assign the note to a symbol (and to give it a printable name) for later identification. When a note object is created it is added to the track object whose identity is held in global variable ***current-track*** (more on this later). This can be overridden by explicitly assigning a track via the **:track** parameter. The **:comment** parameter, if specified, is echoed in the **rt** score, preceded by '/' (rt's comment delimiter), directly above the note with which it is associated.

An alternative to the **:at** parameter for setting a note object's start time is the **:sync** parameter. **Sync** allows the user to

specify that the note's start time is to be computed relative to the start time of some other note. The reference note is normally in the same track, though not necessarily. Sync grammar looks like this:

```
:sync [time [:from-start | :from-end]] [[:to] note [time [:from-start | :from-end]]]
```

It basically reads "synchronize some particular moment in this note to a moment in another note." The first time value (in seconds) is the reference time for the note being synchronized. If **:from-start** is specified, the reference time is taken to be relative to the start time of the note. If **:from-end**, relative to the termination time. If neither is specified, the reference time is taken to be an absolute time value in the soundfile associated with the note object. This is especially useful when you're using **RTLisp** in conjunction with a graphic sound editing application that displays a time scale. If no reference time is specified, the default is taken to be the start time of the note being synchronized. The **:to** parameter is optional, and is offered simply to improve readability. The note object following **:to** (if present) is the note to synchronize to. If this note is not specified, the default action is to synchronize to the note object created immediately prior to the note being synchronized. The subsequent reference time specification (if present) has the same meaning for the reference note as that for the note being synchronized. Examples:

```
(note ... :sync 0.1 :from-start :to wham -2.5 :from-end ...)
    ; sync 0.1 seconds from start of this note to 2.5 seconds before end of note `wham'
(note ... :sync ...)
    ; sync this note to play simultaneously with note created immediately prior
(note ... :sync 3.2 :to kaboom ...)
    ; sync this note such that the audio data 3.2 seconds into note's soundfile plays
    ; simultaneously with note `kaboom'. It's expected that `3.2' lies in the interior
    ; of the current note, but it's not strictly necessary.
```

An abbreviated form of **:sync** is

```
(note ... :splice <intvl> ...)
```

The created note is set to begin <intvl> seconds before the end of the previous note. As of this writing, splicing does not

effect any sort of automatic cross fading. If a cross fade is desired it's up to the user to provide appropriate amplitude envelopes.

When the start time of a note is set with **:sync** or **:splice**, the start time is actually stored as a Lisp expression rather than a simple number, and the computation of the absolute start time is deferred until it's time to generate the **rt** score. This insures that synchronization is maintained even if a reference note is shifted in time.

Once a note is created, it can later be edited but only if it is assigned a tag with the **:is** parameter. The **edit** function looks just like the **note** function...

```
(edit <tag> <param> <value> <param> <value> ...)
```

...except that it requires <tag> to be specified so that it knows which note object is to be edited. Only the values of parameters appearing in the function call are changed:

```
(edit gong :pan 0.8 :gain 0.4) ; change pan and gain values, leave other parameters alone
```

As with sound objects, you can get information on note objects with the **info** function:

```
(info <note-tag>) ; info on particular note  
(info notes) ; brief info on all notes
```

Track Objects

The track object class is the mechanism by which note objects may be grouped into phrases, sections, voices or whatever structures the composer finds useful. (It is important that the reader understands that the **RTLisp** track object does not necessarily correspond to **rt**'s notion of a track. It's probably best that these two are thought of as 'virtual' and 'physical', respectively.) When a note object is created it is placed in a track, either by default or explicitly via the **:track** parameter. Later, the note may be moved to another track with the **edit** function. A track is created or edited with the **track** function:

```

(track      <name>                ; tag, required
  :state <t | nil>                 ; [dis|en]able track, default = enabled
  :reserve <track#>               ; reserve rt track
  :gain <value>                   ; gain factor, default = 1
  :at <time>                      ; start time for track, default = 0
  :env <envelope>                 ; amplitude envelope
  :pan <control>                  ; track pan
  :comment "<string>"
)

```

If a track called <name> does not exist, the **track** function will create it. Otherwise, the **track** function edits the values of parameters present in the call. Editing can also be done with the **edit** function, just as for note objects. The track's **:state** determines whether or not it is played in the mix. Enabling or disabling the track enables or disables all the notes in that track. The **:gain** parameter scales the gain values of the track's constituent notes, allowing whole groups of notes to be raised or lowered in volume under control of a single value. Similarly, the track's amplitude **:envelope** (if present) is superimposed on each note's amplitude envelope, thus providing a means for effecting, say, a crescendo over the total duration of all of the track's notes. The **:pan** parameter works in much the same way, except that it preempts the individual note's pan control instead of modulating it. As with notes, the **:pan** value may evaluate to a simple number or to a time/value envelope. When **RTLisp** generates the **rt** score for a mix, the track objects will assign physical **rt** tracks to their notes in such a way that no two note objects ever overlap in time on a given track. This avoids the note preemption behavior inherent in the **rt** driver. However, it may be desirable to 'hard-wire' a track object to a physical **rt** track, perhaps to take advantage of the gain sliders in **rt**'s tracks window. This hard-wiring can be achieved by setting the **:reserve** parameter to the desired **rt** track number. Any number of track objects may reserve the same **rt** track. Track objects that do not reserve an **rt** track are prevented from using reserved tracks. Their tracks are assigned instead from a free track pool. There is no checking for note preemption for track objects that reserve **rt** tracks. That's left to the user. If the **:at** parameter is specified, all notes in the track object are time-shifted uniformly according to the time value (seconds) given. Remember that each note's start time is interpreted relative to its containing track's start time. This makes it easy to shift a whole section of music by changing a single number. Like note objects, tracks accept the **:sync** specification in lieu of **:at**, but the syntax is slightly different:

:sync [[note] time [:from-start | :from-end]] [[:to] object [time [:from-start | :from-end]]]

If `note' is provided, the first time specification is taken to be relative to the note object (which, presumably, is a constituent of the track being synchronized). Otherwise, the time refers to the start time and duration of the track itself. The `object' (which is required) may be either a note or another track.

(**track** frogs ... **:sync** 0.5 **:from-start** **:to** crickets 3.0 **:from-end** ...)
; frogs track starts 3.5 seconds before crickets track ends

When **RTLisp** is started it creates a track called **main**, which is the initial default track. The default track is changed with:

(**on-track** <track>)

If <track> does not exist, **on-track** will create it with default settings. In any case, <track> remains the default track until the next invocation of **on-track**. Each subsequently created note object is assigned to the default track unless the note's creation includes a **:track** specification. Track objects may be selectively enabled or disabled. Notes belonging to disabled tracks are not included in the mix and thus are not written into the **rt** score. Tracks may be turned off or on in a number of ways:

(**on** track track ...)
(**off** track track ...)
(**on** all)
(**off** all)
(**on** all but track track ...)
(**off** all but track track ...)

Functions **reserve** and **release** deal with reserving and freeing physical **rt** tracks:

(**reserve** **:track** <n> **:for** track track ...)
(**release** track)

:track and **:for** are optional and are provided only to enhance readability. The **reserve** function simply reserves physical track <n> (where <n> evaluates to a legal **rt** track number) for all of the tracks specified. The **release** function accepts either an **rt** track number or a track object. If a number, all track objects which had previously reserved this track number are unbound from it and revert to getting their physical tracks from the free track pool. If the argument is a track object, then only that object is unbound.

The Mix Function

Mix does the actual work of compiling all the information specified in note, track and sound objects to generate an **rt** score suitable for parsing and execution by the **rt** audio driver process. **Mix**'s call format looks like this:

```
(mix      :from <x:0>           ; start mixing from time...
          :to {or :for} <x:9999>; ...till (or for duration x)
          :soundfile "name"     ; write soundfile instead of mixing  live
          :nosort               ; don't pre-sort notes
          :dump                 ; write rt code to screen            (don't run rt.driver)
          :copy                 ; write rt code to pasteboard      "
          :rtapp                ; write rt code to Lansky's rt interface  "
          :pipe <"cmd">         ; write rt code to unix pipe
          :gain <g:1>           ; set output gain for both channels
          :tick <t:0.01>        ; set time resolution (rt `timescale')
        )
```

All parameters are optional. The defaults for the **:from**, **:to**, **:gain** and **:tick** parameters are 0, 9999, 1 and 0.01, respectively. The **:from** and **:to** parameters may alternatively specify note objects instead of time values, in which case the mix begins with the beginning of the **:from** note and ends at the finish of the **:end** note. If **:dump**, **:copy**, **:rtapp** or **:pipe** are specified, the **rt** score is generated but no actual mix is done, either live or to a soundfile (unless the score is **:piped** to the **rt** driver). If none of these four options is selected, the **rt** driver is started and the **rt** score is piped to it, resulting either in a real-time mix (if there are no errors in the score) or a mix to the specified soundfile. It seems that the **rt** driver is better

behaved when notes are sorted by their start times. However, sorting does take time. If you're careful to generate note objects in chronological order, sorting can be switched off with the **:nosort** option. The **:rtapp** option first causes the **rt** score to be written to the NeXT Workspace's public pasteboard object. **Rt.app** is then signalled to pick up what's in the pasteboard and process it just as if it were reading an **rt** script from a file. Soundfile names appear in **rt.app**'s soundfile window, **RTLisp**-generated notes appear in the playnote window and mix start and end times appear in their text forms in the control window. Presently, in order for this to work, **rt.app** must already have been launched. Theoretically, this shouldn't be necessary, but it is. This appears to be a bug and NeXT is looking into it.

Other Functions

(**ld** ["filename"])
; load "filename". If filename is not given, reload file previously loaded.

(**clear**)
; initialize **RTLisp**: terminate active pipe process, set current track to **main**, discard all note objects

(**clear-notes** [track track...])
; clear specified tracks' note lists. If no tracks specified, erase all notes.

Installation

Included with this document should be the **XLISP** executable binary and the **RTLisp** `workspace'. An **XLISP** workspace is basically an image of the **XLISP** environment at the time of a call to the **save** function. It's assumed that the **rt** driver program, **rt.driver**, resides at /LocalApps/rt.app. If this is not the case, you'll need to change **rt-cmd** in the RTLisp source and rebuild the RTLisp workspace. Suggested installation procedure:

- login as (or su) root
cp xlisp /usr/local/bin
mkdir /usr/local/lib/xlisp

```
cp rtlisp.wks /usr/local/lib/xlisp
echo 'xlisp -w/usr/local/lib/xlisp/rtlisp.wks $*' > /usr/local/bin/rtlisp
chmod 755 /usr/local/bin/rtlisp
```

Example

Here's a very simple RTLisp program which should run on any NeXT system:

```
(clear)                                ; initialize system

(sounds-from "/NextLibrary/Sounds")    ; set soundfile search path
(sound basso is "Basso.snd")           ; declare sounds to be used with note objects
(sound frog is "Frog.snd")
(sound pop is "Pop.snd")
(sound tink is "Tink.snd")
(sound bonk is "Bonk.snd")
(sound funk is "Funk.snd")

; pick-sound: randomly pick a sound object from global variable *sounds*
(defun pick-sound () (nth (random (length *sounds*)) *sounds*))

; notes: create `n' note objects (by default, assigned to track main). Each note's sound object is picked randomly and
;         it's start time is 0.1 seconds later than the previous note.
(defun notes (n)
  (clear-notes)
  (do ((i 0 (1+ i))                ; for i = 0; i < n; i++
       (st 0.0 (+ st 0.1)))        ; st = 0.0; st += 0.1
      ((= i n) nil)
      (note :snd (pick-sound) :at st))) ; pick sound, assign it to created note, start at `st'
```

; The code above does not generate an **rt** score, it simply generates a bunch of note objects and puts them in track **main**.
; To see the **rt** score, try:

```
(mix :dump)
```

; To hear the mix (assuming the rt.driver is properly installed), do:

```
(mix)
```

Peter M. Yadlowsky
Academic Computing Center
University of Virginia
pmy@virginia.edu

XLISP version 2.1c, Copyright (c) 1988, by David Betz
rt by Paul Lansky and Kent Dickey, Princeton University

12/4/91