Realtime extension for the sound editor SND.

Kjetil S. Matheussen

8th July 2005

Contents

1 Introduction **4**

2 Quick Start **5**

2.1 Setting up SND and Emacs 5

2.2 Two short examples 5

3 The RT Engine **7**

3.1 Features 7

3.2 Guile Functions to control the Realtime Engine 7

3.3 Global variables 8

4 The RT compiler **9**

4.1 Features 9

4.2 Macros and functions to compile and run rt-functions 9

4.3 The ≪realtime≫ class 10

4.4 Various 11

5 The RT programming language **12**

5.1 Features 12

5.2 Limitations 12

5.3 Types 13

5.4 Closures 14

5.5 Functions 14

5.6 Macros 14

5.7 Reading and writing rt-variables from the guile-side 15

5.8 Shared variables 16

5.9 Midi 16

5.10 Ladspa plugins 16

5.11 Routing signals 17

5.11.1 Buses 17

5.11.2 The in and out macros 17

5.11.3 The syntax for in and out 18

5.11.4 The syntax for definstrument 20

5.12 Using CLM 20
5.13 Lockfree Ringbuffer 22
5.13.1 ringbuffer 23
5.13.2 ringbuffer-location 23
5.14 Provided Functions, Macros and Special Forms 24
5.14.1 Blocks 24
5.14.2 Control Flow 24
5.14.3 Functions 24
5.14.4 Variables 24
5.14.5 Conditionals 25
5.14.6 Iteration 26
5.14.7 Logical Operators 27
5.14.8 Mathematics 27
5.14.9 Lists, pairs, vectors and VCT's 28
5.14.10 Others 29
5.15 Various 30

6 Missing functionality **32**
6.1 Order of execution 32

7 History **33**

8 Notes **34**

1 Introduction

The RT extension for the sound editor SND consists of two parts:

The RT Engine - An engine for doing realtime signal processing.

The RT Compiler - A compiler for a scheme-like programming language to generate realtime-safe code understood by the RT Engine.

As far as possible I have tried to make the language behave and look like scheme. However, there are no support for consing, creating closures, or other operations that can trigger a garbage collection, so its not really a very schemish language although it visually looks a lot like scheme. Perhaps consing and more advanced stuff will be implemented later, but it should not be necessary as the code blends very fine into Guile. If you need to create lists or closures, you have to do that in Guile. So, actually, technically, the language is more like C than Scheme.

Acknowledgments

Thanks to the guile development team for creating Guile. Thanks to the gcc development team for creating gcc. Thanks to Bil Schottstaedt for creating SND. Thanks to Notam and the Art Council Norway for economical support making this software.

2 Quick Start

2.1 Setting up SND and Emacs

This is how I work. Perhaps you'll find this setup comfortable.

1. Download and unpack the latest version of snd-ls from http://www.notam02.no/arkiv/src/snd/
2. Compile and install it like this:

   ```
   ./build
   ./install
   ```

3. Link a "scheme" executable so that it points to the snd-ls binary

   ln -s ‘which *snd-ls*‘ ~/bin/scheme

4. Start the jack server
5. Start emacs
6. Maximize Emacs so that it fills the whole screen
7. Load the file called "rt-examples.scm" into Emacs.
8. Split your window in two parts: C-x 2
9. Move the cursor to lower part window: C-x o
10. Run snd-ls: M-x run-scheme
11. Hide the SND window behind the Emacs window. We don't need to look at SND.
12. Start the engine and compiler by writing (*load-from-path* "rt-compiler.scm")
13. Move the cursor to the upper part window: C-x o
14. Evaluate blocks of code by placing the cursor inside a block and press Ctrl+Alt+x

2.2 Two short examples

1.

```
(let ((osc (make-oscil)))
  (<rt-play> 0 3
             (lambda ()
               (out (* 0.8
                       (oscil osc))))))
```

A sinus is/should be heard for three seconds.

2.

```
(define-rt2 (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(fib 30)
-> 832040.0
```

(There are more examples in the file rt-examples.scm)

3 The RT Engine

The main purpose of the RT Engine is to receive <*realtime*> objects and provide ways to control exactly when a <*realtime*> objects processing function is being called. The API for doing this is hidden from the user, and is accessed by calling methods in the <*realtime*> class.

3.1 Features

- Realtime safe
- Jack driver
- Unlimited number of input and outputs
- The RT Engine is a class, and unlimited number of <*RT-Engine*>-instances can run simultaneously. [1]
- Protection mechanism provided to avoid locking up the computer when using too much CPU.
- Properly made priority queue system for events to ensure a stable CPU load (At least, I hope so).
- Controlled entirely from scheme using Guile

3.2 Guile Functions to control the Realtime Engine

(*rte-pause***)**
Pause the engine.

(*rte-continue***)**
Starts running the engine after being paused.

(*rte-restart* #:key *num-input-ports num-output-ports***)**
Stops and starts engine. If you only have stuck sounds, use (*rte-silence!*) instead. (Warning, theres a huge memory-leak when using this function.)

(*rte-silence!***)**
Call this function if you have stuck sounds or <*realtime*> instances running. (This function is equivilant to the *freeAll* OSC message for Supercollider3)

(*rte-time***)**
Returns the time in seconds since the engine was started.
(*rte-time*) = (*rte-frames*) / (*rte-samplerate*)

(*rte-samplerate***)**
Returns the samplerate.

(*rte-frames***)**
Returns the number of frames since the engine was started.

[1] Well, actually just 16.

(*rte-is-running?*)
Returns true if engine is running. (Ie. not paused)

(*rte-max-cpu-usage*)
This is a setter-function. The engine will stop calling further <*realtime*> instances if the cpu-usage for the jack client is higher than (*rte-max-cpu-usage*). This is a safety mechanism to avoid locking up the computer or jack to kill the client, which can be unfortunate or embarrassing in certain situations. Note however that if exactly 1 <*realtime*> instance is using too much cpu, this mechanism will not work. The default value is set to 80. At least on my machine, this is a comfortable value, because the machine is still quite responsive at 80%.

(*rte-info*)
Returns a list of 7 elements:

1. Current size of the priority queue.
2. Size of the priority queue.
3. Number of lost events because the priority queue was full.
4. Number of events waiting to be run.
5. Number of <*realtime*> instances currently running.
6. CPU load reported by the jack client.
7. Number of times the execution of <*realtime*> instances have been stopped because the cpu-usage was higher than (*rte-max-cpu-usage*). This number is mostly useful when being compared to a previous value, or to check whether its higher than 0.

3.3 Global variables

∗*rt-engine*∗
The main object, an instance of <*rt-engine*>. Evaluate (-> ∗*rt-engine*∗ *dir*) to get the name of all the methods and subclasses.

∗*rt-num-input-ports*∗
Number of input jack ports. You can define this variable before loading rt-compiler.scm or rt-engine.scm to override the default value of 8.

∗*rt-num-output-ports*∗
Number of output jack ports. You can define this variable before loading rt-compiler.scm or rt-engine.scm to override the default value of 8.

∗*out-bus*∗
The bus connected to the soundcard output.

∗*in-bus*∗
The bus connected to the soundcard input.

4 The RT compiler

The RT compiler works by translating the Scheme-like input-code into s-expression based C-code, which is further translated into normal C-code by the *eval-c* macro. The *eval-c* macro is found in the file eval-c.scm. *eval-c* does also the compiling and linking by calling gcc.

4.1 Features

- Compilation of simple lisp functions into machine code.
- All compiled code should normally be hard real time safe.
- The compiled code should be so fast, that theres normally nothing to gain by writing a function in C. At least, thats the plan, its currently not always true, although not far from.

4.2 Macros and functions to compile and run rt-functions

(*rt-compile* ...) Macro

```
(define a (rt-compile (lambda (b c)
                        (* b c))))
```

(*rt-c* ...) Macro

Same as rt-compile

(*rt-funcall* ...) Macro

```
(rt-funcall a 2 3)
=> 6.0
```

(*rt-func* ...) Macro

```
(define a (rt-func (lambda (b c)
                     (* b c))))
(a 2 3)
=> 6.0
```

(*rt-safety* ...) Setter function

rt-safety is a setter function. If set to 0, no runtime error checking is performed. Its safe to set (*rt-safety*) to 0 if you don't get any "RT RUNTIME ERROR" error messages to stderr when running your function, and you are sure thats impossible to happen. On the other hand, if you do see an "RT RUNTIME ERROR" message printed to stderr when running your function, theres a good chance you will lock up your machine by setting (*rt-safety*) to 0.

For operations on lists, pairs and vectors, this could have an impact on the performance. But, generally, don't expect to see a big improvement in the performance by setting it to 0.

(<*rt*> ...) Macro

Creates a subclass of <*realtime*> :

```
(define a (<rt> (lambda ()
                  (out (oscil osc)))))
```

(<*rt-play*> ...) Macro

Creates a subclass of <*realtime*> :

```
(<rt-play> (lambda ()
             (out (- (random 1.8) 0.9))))
```
[*white noise is being generated*]

```
(<rt-play> 1
           (lambda ()
             (out (- (random 1.8) 0.9))))
```
[*one second later*, *white noise is being generated*]

```
(<rt-play> 1 10
           (lambda ()
             (out (- (random 1.8) 0.9))))
```
[*one second later*, *white noise is being generated for ten seconds*]

<*rt-play*> is a macro that calls <*rt*>.

(<*rt-play-abs*> ...) Macro

Same behavior as rt-play, except that the start-time is absolute time: (<*rt-play*> 1 *func*) is the same as (<*rt-play-abs*> (+ 1 (*rte-time*)) *func*)

(*rt-clear-cache!*) Function

Compiled rt-functions are cached into memory (currently not to disk). (*rt-clear-cache!*) clears the cache.

4.3 The «realtime» class

The <*realtime*> class provides the following methods:

- (*play-abs* [*start*] [*duration*]) Method
  Starts playing at the absolute time *start*, stopping at the absolute time (+ *start duration*). Default value for *start* is the current time. If *duration* is not specified, a stop command is not scheduled.

- (*stop-abs* [*end*]) Method
  Stops playing at the absolute time *end*. Default value for *end* is the current time.

- (*play* [*start*] [*duration*]) Method
  Starts playing *start* seconds into the future from the current time, stopping at (+ *start duration*) seconds into the future from the current time. Default value for *start* 0. If *end* is not specified, a stop command is not scheduled.

- (*stop* [*end*]) Method
  Stops playing *end* seconds into the future from the current time. Default value for *end* is 0

4.4 Various

- For define-rt, I have the following lines in my .emacs file:

```
(font-lock-add-keywords
 'scheme-mode
 '(("(\\(define-rt\\)\\>\\s-*(?\\(\\sw+\\)?"
    (1 font-lock-keyword-face)
    (2 (cond ((match-beginning 1) font-lock-function-name-face)
             ((match-beginning 2) font-lock-variable-name-face)
             (t font-lock-type-face))
       nil t))))
```

- The definstrument-macro is implemented so that any rt-code is compiled when the instruments definition is being evaluated, and not when being called.

- To extend the RT language with your own functions written in C, look at how the <*rt-func*> function works in rt-compiler.scm. The *rt-renamefunc* macro may also be of large use. Beware though that the API for <*rt-func*> and *rt-renamefunc* might still change.

5 The RT programming language

5.1 Features

- Provides a subset of the scheme r5rs standard
- Most of CLM is supported as well as various other functions specific for snd, sndlib and guile.
- Lisp macros
- Types are automatically determined, but the common lisp operators *declare*[3] and *the* can be used on numeric variables and expressions to help the compiler produce more efficient code.
- Guile can both read and write variables which is used inside the compiled functions.
- Possible to read Guile variables. [2]
- It should not be possible to cause a segfault by running a compiled functions. But for now, I know that at least when dividing or moduling by 0, you will get a segfault. I don't know how to handle that situation yet. There are also probably a lot of other situations that might cause a segfault. Please send me code that cause segfaults.
- Error checking. If there is an error in your code that cause the compilation to stop, you sometimes get a human readable explanation about it, if you are lucky.

5.2 Limitations

- A variable can not change type. No dynamic typing.
- No allocation (consing, vectors, etc.)
- No closures
- No optional arguments or keyword arguments. (Optional arguments are supported with the help of macros though.)
- No boolean type or symbols: #f=0, #t=1
- Not possible to call Guile functions.
- Not possible to set Guile variables. (There are ways around this though)
- Tail-recursiveness is not guaranteed.
- The function to determine types is wrongly designed, so you sometimes have to manually set the types for variables by using *declare* or *the*. (its a bug that should be fixed, but theres other more important tasks placed earlier in the queue.)

[2] Writing Guile variables only half-worked, and sometimes made guile segfault, so I removed it.

5.3 Types

- The rt-language automatically detects the types for variables.
- A variable can not change type.
- There is no boolean type, so #f=0 and #t=1.
- To improve the performance, use *declare* and *the* to specify types, just like in common lisp. See below for usage of *declare* and *the*. [3]
- It is no point to declare non-numeric variables. But it won't hurt either, unless wrongly declared, which will only make the compilation stop.
- Supported numeric types: <*int*>, <*float*> and <*double*> . These are directly mapped to the int, float and double C-types.
- If there are more alternative types than one for a variable, and its type has not been declared with *declare*, the type will be determined based on the following rules for merging different types:

<*int*>	+	<*float*>	− >	<*float*>
<*float*>	+	<*double*>	− >	<*double*>
<*int*>	+	<*double*>	− >	<*double*>
<*void*>	+	Any type	− >	<*void*>

  Everything else is illegal.

- I guess there can be a need for an int type that is guaranteed to be at least, or exactly, 64 bits wide. Please tell me if you need such a type, and what its name should be.
- Guile variables (ie. of type <*SCM*>) are automatically converted on the fly to the proper types:

(**let** ((*a* (*the* <*int*> (*vector-ref vec* 2))))
  a)

  Will result in code that works like this:

(**let** ((*a* (*scm_to_int* (*vector-ref vec* 2))))
  a)

  Without using the *the* operator, it would have worked like this:

(**let** ((*a* (*the* <*SCM*> (*vector-ref vec* 2))))
  a)

  For the first example, if (*vector-ref vec* 2) hadn't been a numeric value, or there aren't as many as 3 elements in *vec*, an error had been caught, and the evaluation of the compiled rt-code would stop.

[3] Note also that although the compiler tries to determine the most fitted type for a numeric variable, it often fails. So in some situations you have to declare numeric variables although it shouldn't have been necesarry. For an example, study the C-code generated by the instrument called *extremely-simple-delay* found in the file rt-examples.scm. Two of the local float variables should have been ints. This is because the compiler is buggy. It will hopefully be fixed though.

5.4 Closures

Closures are not supported. And worse, there is currently no checking whether the code is safe in a language that doesn't support closures.

The following code:

```
(define a (rt-compile (lambda ()
                        (let ((a (lambda (b)
                                   (declare (<int> b))
                                   (lambda ()
                                     b))))
                          ((a 50))))))
(rt-funcall a)
```

...returns 0. [4]

(Note, I manually had to add (*declare* (<*int*> *b*)) to make it compile because of a bug in the compiler.)

5.5 Functions

```
(define-rt (add-really a b)
  (+ a b))

(define-rt (add a b)
  (add-really a b))

(rt-funcall (rt-c (lambda ()
                    (add 2 3))))
=> 5
```

Optional, rest or keyword arguments are not supported.

5.6 Macros

Macros are straight forward:

```
(define-rt-macro (add . args)
  `(+ ,@args))

(rt-funcall (rt-compile (lambda (a b c)
                          (add a b c)))
            2 3 4)
=> 9
```

And keyword arguments:

```
(define-rt-macro (add a1 a2 (#:a3 3) (#:a4 4) (#:a5 5))
  `(+ ,a1 ,a2 ,a3 ,a4 ,a5))
```

[4] The current behavior guarantee the return value to be 0. But that behavior might change.

```
(rt-funcall (rt-compile (lambda ()
                          (add 1 2 #:a4 9))))
=> 20
```

[1+2+3+9+5]

The function *rt-macroexpand* works the same as *macroexpand*, but for rt-macros. It can be used inside other rt-macros, and is currently used in the **if**, *min*, *max*, **and** and **or** macros to speed up some situations.

When letting a variable name start with the prefix *expand/*, like this:

```
(define-rt-macro (add expand/a expand/b)
  `(+ a b))
```

..a and b are macroexpanded automatically. In some situations, this can cause increased performance. (But not in the short add macro above though.)

For the define-rt-macro macro, I have the following lines in my .emacs file:

```
(font-lock-add-keywords
 'scheme-mode
 '(("(\\(define-rt-macro\\)\\>\\s-*(?\\(\\sw+\\)?"
    (1 font-lock-keyword-face)
    (2 font-lock-constant-face
       nil t))))
```

5.7 Reading and writing rt-variables from the guile-side

```
(definstrument (instrument)
  (let ((osc (make-oscil))
        (vol 0.8))
    (<rt-play> 0 10
               (lambda ()
                 (out (* (oscil osc)
                         vol))))))

(define i (instrument))

(-> i vol)
=> 0.8

(-> i osc)
=> #<oscil freq: 440.000Hz, phase: 0.256>
```

To change the volume:

```
(set! (-> i vol) 0.2)
(-> i vol)
=> 0.2
```

To change the frequency:

```
(set! (mus-frequency (-> i osc)) 200)
=> 200
```

This will return an error:

```
(set! (-> i osc) (make-oscil))
```

...because only numbers and buses can be set.

5.8 Shared variables

The guile-function *make-var* (with an optional value argument for value) allocates a variable that can be both read from and written to both from the guile and the rt side with the functions *read-var* and *write-var*:

```
(let ((vol (make-var)))
  (<rt-play> 0 10
             (lambda ()
               (out 0 (* (read-var vol) (in 0)))))
  (<rt-play> 0 10
             (lambda ()
               (out 1 (* (read-var vol) (in 1)))))
  (write-var vol 0.2)
  (in 5000
      (lambda ()
        (write-var vol 1.0))))
```

(API might change)

make-glide-var, *read-glide-var* and *write-glide-var* are functions which interpolates the read values to avoid large jumps. Check out source code for syntax.

5.9 Midi

Receiving alsa midi is supported. Check out rt-examples.scm for a quite large example. If you don't want alsa-midi (for example if you're running osx), set **use-alsa-midi** to #f before loading rt-compiler.scm.

```
(<rt-play> (lambda ()
             (receive-midi (lambda (control data1 data2)
                             (declare (<int> control data1 data2))
                             (printf "gakk! %x %x %x\\n" control data1 data2)))))
```

5.10 Ladspa plugins

Ladspa-plugins are supported, altough a bit inefficiently. Using ladspa-plugins might also trigger bugs in some plugins because the framesize is currently always 1. The three plugins I have tried so far have

worked fine though. [5]

make-ladspa is a guile function that creates a plugin object. First argument is filename, and second is the name of the label.

ladspa-set! is implemented both for guile and rt, and sets an input control-number.

ladspa-run is an rt function. First argument is the plugin object, and second argument is a *vct* holding sound-data. The function returns a *vct*.

make-ladspa-gui is a guile function that automatically makes a gui for the ladspa object. This one should hopefully be convenient to use to find default values. (see rt-examples.scm for an example)

```
(definstrument (ladspatest)
  (let ((am-pitchshift (make-ladspa "am_pitchshift_1433" "amPitchshift")))
    (<rt-play> (lambda ()
                 (out (ladspa-run am-pitchshift
                                  (vct (in 0))))))))

(define l (ladspatest))
(ladspa-set! (-> l am-pitchshift) 0 0.5)
(make-ladspa-gui l)
(-> l stop)
```

(API might change)

5.11 Routing signals

5.11.1 Buses

To create a bus, use *make-bus*. *make-bus* takes one optional argument, which is number of channels. Default is 1.

To read from a bus, use *read-bus*. To write to a bus, use *write-bus*.

When writing to a bus, you add your signal if the bus had been written to in the current cycle. If not, the old value is overwritten.

When reading, you always get the current value, unless the bus hadn't been written to for two or more cycles. Then you'll get zero.

This behaviour is a bit different from Supercollider, but I think its better. I migh change the behaviour later though.

5.11.2 The in and out macros

The *in* and *out* macros are supposed to provide a convenient interface to the bus system. The basic way to play a sound to the soundcard is like this: (*write-bus* ∗*out-bus*∗ *sound*). Instead, you can just write (*out sound*).

[5] I think at least all swh-plugins should work fine because they seem to use a common ringbuffer to buffer up sound-data.

But thats not all. *in* and *out* also tries to automatically find out whether you are playing a VCT or just a single value, and which channel(s) to write to or read from.

But thats not all either. The compiler automatically creates two special bus-variables called *out-bus* and *in-bus*, which, if *out-bus in-bus* aren't already declared locally [6] is set to ∗*out-bus*∗ or ∗*in-bus*∗. And since buses are settable, you can redefine outputs and inputs the way you like:

```
(definstrument (oscillator)
  (let ((osc (make-oscil)))
    (<rt-play> (lambda ()
                 (out (oscil osc))))))
(define o (oscillator))
(define bus (make-bus 2))
(set! (-> o out-bus) bus)
(definstrument (volume vol)
  (<rt-play> (lambda ()
               (out (* vol (in))))))
(volume 0.5 #:in-bus bus)
(rte-silence!)
```

This way, you very seldom should have the need to use *read-bus* and *write-bus* directly.

In my .emacs file, I have the following lines to colorize *out* and *in*:

```
(font-lock-add-keywords
 'scheme-mode
 '(("(\\(out\\)\\>\\s-*(?\\(\\sw+\\)?"
    (1 font-lock-keyword-face
       nil t))))
(font-lock-add-keywords
 'scheme-mode
 '(("(\\(in\\)\\>\\s-*(?\\(\\sw+\\)?"
    (1 font-lock-keyword-face
       nil t))))
```

5.11.3 The syntax for in and out

This simple function will software monitor the two first channels for 10 seconds:

```
(<rt-play> 0 10
           (lambda ()
             (out 0 (in 0))
             (out 1 (in 1))))
```

This function does the same, but swaps the channels:

```
(<rt-play> 0 10
           (lambda ()
             (out 0 (in 1))
```

[6] Never declare *in-bus* and *out-bus* in the toplevel!

```
    (out 1 (in 0))))
```

This function does the same, but will mix both input-channels before sending the result to both channel 0 and 1.

```
(<rt-play> 0 10
           (lambda ()
             (out 0 1 (in 0 1))))
```

This function does exactly the same as the one above, but by using a shorter syntax:

```
(<rt-play> 0 10
           (lambda ()
             (out (in))))
```

This function will send the sum of the first two input-channels to the 10 first even-numbered output-channels:

```
(<rt-play> 0 10
           (lambda ()
             (out 0 2 4 6 8 10 12 14 16 18 (in))))
```

This last argument for *out* can also be a vct. The following function will software monitor the two first channels for 10 seconds:

```
(<rt-play> 0 10
           (lambda ()
             (out (vct (in 0)
                       (in 1)))))
```

This function does the same, but for channel 2, 3 and 4:

```
(<rt-play> 0 10
           (lambda ()
             (out 2 (vct (in 2)
                         (in 3)
                         (in 4)))))
```

This function does the same as the one above, but halves the volume:

```
(<rt-play> 0 10
           (lambda ()
             (out 2 (vct-scale! (vct (in 2)
                                     (in 3)
                                     (in 4))
                                0.5))))
```

Note that the API for out and in might change. (although probably not too radically...)

5.11.4 The syntax for definstrument

When using the functions *in* or *out* inside a *definstrument* block, you can call the instrument with the hidden key-word arguments *out-bus* and *in-bus*.

Example:

1. First create an instrument, a simple oscillator.

```
(definstrument (oscillator)
  (let ((osc (make-oscil)))
    (<rt-play> (lambda ()
                 (out (oscil osc))))))
```

2. Make a bus with two channels

```
(define bus (make-bus 2))
```

3. Let the oscillator play to the bus

```
(oscillator #:out-bus bus)
```

   [No sound in the loudspeakers]

4. Connect the output from the oscillator to the soundcard

```
(<rt-play> (lambda ()
             (out (read-bus bus))))
```

5.12 Using CLM

Almost all CLM classes are supported, as well as all their methods, and other functions. Most things should work as expected, hopefully.

Exceptions:

- CLM constructors are not supported:

```
(define func (rt (lambda ()
                   (let* ((osc (make-oscil :frequency 440)))
                     (oscil osc)))))
```

  [*error*]

- For all the generators that may require an input-function argument, (that is convolve, granulate, phase-vocoder and src), the input-function argument is not optional but must be supplied:

```
(src s
     (lambda (direction)
       (readin file)))
```

- (*mus-srate*) returns the samplerate specified by the current rt-driver (ie jack), not what SND reports. To avoid different values for mus-srate reported by snd and rt, (**set!** (*mus-srate*) (*rte-samplerate*)) is called in the init-process of rt-engine. If you set (*mus-srate*) later (in Guile), you might get unexpected results.

- (*mus-srate*) is not settable.

- The CLM generators in-any and out-any are not available. You probably want to use read-bus and write-bus instead.

- readin has mostly been rewritten to be able to buffer the whole sound first instead of reading from harddisk while playing. The new readin also remembers which buffers are currently in use, so playing the same file many time simultaneously will not cause extra memory usage.

  There is another thing to be aware of though: While the following block should work as expected:

```
(let ((rs (make-readin "1.wav")))
  (<rt-play> 0 10
             (lambda ()
               (out (readin rs)))))
```

  The following block will not:

```
(let ((rs (vector (make-readin "1.wav")
                  (make-readin "2.wav"))))
  (<rt-play> 0 10
             (lambda ()
               (out 0 (readin (vector-ref 0 rs)))
               (out 1 (readin (vector-ref 1 rs))))))
```

  [A run-time error-checker will make the function exit before doing anything, and no sound will be heard.]

  Instead you have to do:

```
(let ((rs (vector (make-rt-readin "1.wav")
                  (make-rt-readin "2.wav"))))
  (<rt-play> 0 10
             (lambda ()
               (out 0 (readin (vector-ref 0 rs)))
               (out 1 (readin (vector-ref 1 rs))))))
```

- Short example of the use of readin, here's a file-player running in an endless loop:

```
(let ((rs (make-readin "/home/kjetil/t1.wav")))
  (<rt-play> (lambda ()
               (if (>= (mus-location rs) (mus-length rs))
                   (set! (mus-location rs) 0))
               (out (readin rs)))))
```

- Reverb for the locsig generator is not implemented. I'm a bit confused about locsig actually. I'm not sure the rt-implementation is correct...
- Non of the frames/mixers/sound IO functions are supported, as they require disk-access, which shouldn't be done inside the audio thread.
- Only *hz->radians* is implemented from the *Useful functions* section of the CLM manual. (Most of them probably only requires a 2-3 lines long macro to be supported though.)
- array-in, dot-product, sine-bank, edot-product, contrast-enchancement, ring-modulate, amplitude-modulate, fft, multiply-arrays, *rectangular->polar*, *rectangular->polar*, spectrum and convolution is not implemented. (Most of these probably only requires 6-10 lines of wrapping-code to be supported.)

Note that calling the CLM methods are not very efficient (that is, the (*mus-*∗) functions). This will hopefully change, but until then, you can in certain situation significantly improve the efficiency of your code by avoid using CLM methods as far as possible. For example,

```
(let ((das-env (make-env '(0 400 1 500)
                         #:duration 5)))
  (<rt-play> (lambda ()
               (let ((envval (env das-env)))
                 (if (>= envval 500)
                     (remove-me))))))
```

use more than reasonable less CPU than:

```
(let ((das-env (make-env '(0 400 1 500)
                         #:duration 5)))
  (<rt-play> (lambda ()
               (let ((envval (env das-env)))
                 (if (>= (mus-location das-env) (mus-length das-env))
                     (remove-me))))))
```

The only exception is the methods for the rt-readin class, which access the attributes directly instead of doing indirect jumps.
So this is as efficient as possible:

```
(let ((rs (make-readin "/home/kjetil/t1.wav")))
  (<rt-play> (lambda ()
               (if (>= (mus-location rs) (mus-length rs))
                   (set! (mus-location rs) 0))
               (out (readin rs)))))
```

5.13 Lockfree Ringbuffer

(not implemented)

Use the ringbuffer clm-like generators to exchange data-streams between guile and the realtime thread.

5.13.1 ringbuffer

```
(define osc (make-oscil))
(define rb (make-ringbuffer (* 8192 256)      ;; Number of samples to buffer. This one should be huge to avoid clicking.
                            (lambda ()
                              (oscil osc))))
(<rt-play> 0 10
           (lambda ()
             (out (* 0.8 (ringbuffer rb)))))
```

The above example is not very good, because you can run oscil directly in the realtime thread. A better example is below, because you can't call readin in the realtime thread. This is how you can play a file without buffering the whole file into memory, which the rt-version of readin does:

```
(define file (make-readin "/home/kjetil/t1.wav"))
(define rb (make-ringbuffer (* 8192 256)
                            (lambda ()
                              (readin file))))
(<rt-play> 0 10
           (lambda ()
             (out (* 0.8 (ringbuffer rb)))))
```

5.13.2 ringbuffer-location

Assumes that location doesn't change to radically, only 0 or 1 steps more or less compared to the last one. It can receive request for any step though, but it might not be able to catch the value in time.

```
(define file (file->sample "/home/kjetil/t1.wav"))
(define rb (make-ringbuffer-location (* 8192 256)
                                     (lambda (location)
                                       (file->sample file location))))
(define position 0)
(<rt-play> 0 100
           (lambda ()
             (out (* 0.8 (ringbuffer-location rb position))) ;; If data is not available, a value from the buffer is returned ir
             (set! position (1+ position))))
```

To delay playing until data is available:

```
(<rt-play> 0 100
           (lambda ()
             (if (ringbuffer-location? rb position)      ;; ringbuffer-location? whether data at the position is available. If
                 (begin
                   (out (* 0.8 (ringbuffer-location rb position)))
                   (set! position (1+ position))))))
```

5.14 Provided Functions, Macros and Special Forms

5.14.1 Blocks

(begin <*body*>**)** Special form

Works as in scheme

5.14.2 Control Flow

(*call-with-current-continuation proc***)** Special form

I think it works as in scheme, but I'm surprised how simple it was to implement...

(Not a very efficient function)

5.14.3 Functions

(lambda . . .) Special form

Works as in scheme, except:

- Functions might not be tail-recursive if possible. Its not very difficult to guarantee a function to be tail-recursive for single functions, but I think gcc already supports tail-recursive functions, so I hope its not necessary to add it explicitly. But I might be wrong!
- Rest argument is not supported: (**lambda** (*a* . *rest*) . . .) (error) (You can work around this to a certain degree by using macros with keywords or optional arguments)

(let . . .) Macro

named let is implemented as a macro.

5.14.4 Variables

(define . . .) Special form

Works nearly as in scheme, but unlike scheme it can be placed anywhere in a block. For example:

```
(begin
  (set! a 2)
  (define d 9)
  (+ a d))
```

...is legal.

(let ...) Special form

Works as in scheme

(let* ...) Special form

Works as in scheme.

(letrec ...) Special form

Works as in scheme.

(*letrec ...)** Special form

Like let*, but with the functions available everywhere:

```
(rt-funcall (rt-compile (lambda ()
                          (letrec* ((a 2)
                                    (b (lambda ()
                                         (c)))
                                    (c (lambda ()
                                         a)))
                            (b)))))
=> 2.0
```

(There is also a letrec* macro for guile in oo.scm.)

(set! ...) Special form

Works as in scheme, except that setting Guile variables will not affect the Guile side:

```
(let* ((a 5)
       (b (rt-compile (lambda ()
                        (set! a 9)
                        a)))
       (c (rt-funcall b)))
  (list a c))
=> (5 9.0)
```

(For setting a large number of variables to be visible from the Guile-side, you can use *vct-set!*)

5.14.5 Conditionals

(if <*test*> <*consequent*> <*alternate*>**)**

(if <*test*> <*consequent*>**)**

Works as in scheme. But beware that there is no boolean type, and #f=0 and #t=1. Therefore, the following expression will return 1, which is not the case for scheme: (**if** 0 0 1)

(case ...) Macro

works as in scheme, except that = is used for testing instead of eqv?

(cond ...) Macro

works as in sheme, but => is not supported

(< ...) Macro

works as in scheme

(> ...) Macro

works as in scheme

(<= ...) Macro

works as in scheme

(= ...) Macro

works as in scheme

(>= ...) Macro

works as in scheme

(= ...) Macro

works as in scheme

(*not* ...) Function

works as in scheme

5.14.6 Iteration

(*while* <*test*> <*body*>) Special form

Works as in Guile, including both break and continue. (Does not expand to a recursive function.)

(*break*) Special form

Used to break out of a while loop. (Not a very efficient function)

(*continue*) Special form

goto the top of a while loop. (Not a very efficient function)

(do . . .) Macro

works as in scheme (Using while)

(*range* . . .) Macro

```
(range i 5 10
       (printf "%d " i))
=> 5 6 7 8 9
(range i 10 5
       (printf "%d " i))
=> 10 9 8 7 6
```

(Using while)

5.14.7 Logical Operators

(and . . .) Special form

works as in scheme

(or . . .) Special form

works as in scheme

5.14.8 Mathematics

(+ . . .) Function

works as in scheme

(− . . .) Macro

works as in scheme

(∗ . . .) Function

works as in scheme

(*/ ...*) Macro

works as in scheme

(*1+ ...*) Function

works as in Guile

(*1− ...*) Function

works as in Guile

(*min ...*) Macro

works as in scheme

(*max ...*) Macro

works as in scheme

And:

sin cos tan abs log exp expt acos asin atan sqrt asinh acosh atanh cosh sinh tanh atan2 (see "man atan2") hypot (see "man hypot")

zero? positive? negative? odd? even? remainder modulo quotient

floor ceiling truncate round truncate

logand logior lognot logxor ash random

5.14.9 Lists, pairs, vectors and VCT's

VCT's are extremely efficiently implemented.

Pairs, lists and vectors are not, and should be avoided.

The provided functions are:

vct make-vct vct-map! vct-length vct-ref vct-set! vct-scale! vct-offset! vct-fill!

vector? vector-length vector-ref

pair? null? car cdr cadr caddr cadddr caddddr cddr cdddr cddddr cdddddr cdar cdadr cdaddr cdadddr caar caadr caaddr caadddr list-ref for-each

5.14.10 Others

(*declare* ...) Special form

Works as in common lisp, except that the name of the types are different: <*int*>, <*float*> and <*double*> .

```
(define-rt (int-fib n)
  (declare (<int> n))
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

...which is the same as:

```
(define-rt (int-fib n)
  (declare (<int> n))
  (the <int>
       (if (< n 2)
           n
           (+ (fib (- n 1))
              (fib (- n 2))))))
```

As you see, the compiler is a little bit intelligent when determine types, so it should not be necessary to use *declare* on all numeric variables and *the* for every expression, (although it shouldn't hurt either).

(*is-type?* <*type*> <*variable*>) Special form

Mostly for internal use:

```
(let ((a 5))
  (is-type? <int> a))
=> 1
(let ((a 5))
  (is-type? <float> a))
=> 0
```

The first argument must be a type, and the third argument must be a variable-name. This is not legal: (*is-type?* <*int*> (+ 2 3)) Used to implement *exact?/inexact?*/etc., and for various optimizations.

(*the* ...) Special form

Works as in common lisp, except that the name of the types are different: <*int*>, <*float*> and <*double*> are the currently supported numeric types.

```
(define-rt (int-cast-add a b)
```

```
(the <int>
     (+ a b)))
```

[...*which is the same as:*

```
(define-rt (int-cast-add a b)
    (declare (<float> a b))
    (the <int>
         (+ a b))) ]
```

(*include-guile-func* **...)** Macro

Includes the code of a Guile function.

```
(define (add a b)
   (+ a b))
(rt-funcall (rt (lambda ()
                  (define add (include-guile-func add))
                  (add 2 3))))
=> 5
```

(*remove-me***)** Macro

Removes the function from the realtime engine. See rt-examples.scm for an example-

(unquote ...) Macro

```
(define a 9)
(rt-funcall (rt (lambda()
                  (+ ,pi ,(+ 5 a)))))
=> 17.1415926535898
```

And:

exact? inexact? number? string? *exact->inexact inexact->exact*

printf (Using c's fprintf with stderr as the first argument. Warning, this one is not realtime safe!)

5.15 Various

* In addition to the functions and macros described above, theres a bunch of very internal functions and macros that used wrongly can hang your machine or destroy your harddisk. Most of them start with a prefix *rt-* or *rt_*.

One very useful function might be *rt_alloc* which does realtime-safe memory allocation. All memory allocated with *rt_alloc* is automatically freed when the function returns.

* Theres still a lot of smaller optimizations thats possible to do. However, gcc $(V >= 3)$ should be able to fix most of these.

* Note to myself, *define-ec-struct* in eval-c.scm needs to be documented.

6 Missing functionality

6.1 Order of execution

There is no API to set the order of execution for the <*realtime*>-instances. It can/should be done automatically by looking at which instance depends on which bus, but that hasn't been implemented (yet).

For now, you can order the execution by using the fact that instances are always put first in the queue.

7 History

08.07.2005: Various bugs fixed, added gliding shared vars and midi. Changed behaviour for read-bus and write-bus.

15.06.2005: Added *make-ladspa-gui*.

09.06.2005: Redefined *define-rt* to define rt-functions instead of guile-functions.

07.06.2005: Added setters for buses, introducing a general patching mechanism. Very nice I think.

06.06.2005: Added the hidden in-bus and out-bus keyword arguments for definstrument. (the *in* and *out* macros uses in-bus and out-bus)

05.06.2005: Added -fast-math to the compiler-options (again). Don't think there is any good reason not to... (?)

05.06.2005: Made input and output busses (∗*in-bus*∗ and ∗*out-bus*), and let *in* and *out* call *read-bus* and *write-bus*.

02.06.2005: Added support for the intel C compiler. To use icc, set ∗*eval-c-compiler*∗ to `"icc"`. (I did not notice any difference in the performance compaired to gcc 4.0 (or even gcc 3.2), but it is supposed to be faster in some situations)

02.06.2005: Added shared variables and ladspa. Fixed letrec*

20.05.2005: Added *vct*, *make-vct*, and the bus-system.

8 Notes

[3] Paul Graham, “ANSI Common Lisp”, 1996, p. 313: “Not an operator, but resembles one (...)”. (About *declare*)