The BEAST Manual

The Beast Project <beast.testbit.org>

March 2019

Abstract

Manual about usage, installation, development and implementation of the Beast digital synthesizer and music creation system.

Contents

List of Tables

1 The BEAST Manual

This is the Beast manual, Beast is a music synthesis and composition program.

The manual is structured into sections covering tutorial material, Howto descriptions, Man pages, file formats and conventions, background discussions concerning development considerations and an API reference.

It is written in Markdown and contributions are welcome, e.g. as pull requests or via the Beast issue tracker.

2 Tutorials

2.1 First Song Editing

Start Beast, it will open up with an empty project by default.

Select `Project` → `New Song`, in order to add notes, tracks and instruments for some music composition.

Select the `Add Track` icon button, so the song has at least one track which corresponds to a playable instrument.

Select the tracks `Synth` field and select an instrument to be played back for this track.

Create a new part within the track by clicking into the tracks timeline. A part contains the notes that the instrument will play when playback is actiavted for this song.

3 How-to Articles

This space is for Howto articels about how to do things.

3.1 Using a MIDI device

1. From the command line, list the known PCM / MIDI devices: `beast --bse-driver-list`
2. Start Beast with a selected MIDI device, e.g.: `beast -m alsa=hw:0,0`
3. Select and play `Demo` → `MIDI Test` to test notes from devices like MIDI keyboards.

4 Manual Pages

4.1 BEAST(1)

NAME

beast - Music composition and modular synthesis application

SYNOPSIS

beast [*OPTIONS*] [*FILES...*]

DESCRIPTION

Beast is the **BE**tter **A**udio **S**ys**T**em. It is a music composition and modular synthesis application released as free software under the GNU LGPL.

Beast comes with various synthesis modules which can be arranged in networks for modular synthesis. It is capable of monophonic and polyphonic voice processing, provides MIDI sequencer functionality and supports external sequencer sources. A huge set of extra modules is available through the support for LADSPA (Linux Audio Developer's Simple Plugin API) plugins.

OPTIONS

Beast follows the usual GNU command line syntax, with long options starting with two dashes ('-').

GENERAL OPTIONS

--skinrc *FILENAME*
: Read skin resources from *FILENAME*.

--print-dir *RESOURCE*
: Print the directory for a specific resource (e.g. 'plugins' or 'images'). Giving just **--print-dir** without an extra argument causes Beast to print the list of available resources.

--merge
: Cause the following files to be merged into the previous or first project.

--devel
: Enrich the GUI with hints useful for (script) developers

-h, --help
: Show a brief help message.

-v, --version
: Print out Beast with component versions and file paths.

-n *NICELEVEL*
: Execute with priority *NICELEVEL*, this option only takes effect for the root suid wrapper 'beast'.

-N Disable renicing to execute with existing priority.

--display *DISPLAY*
: X server display for the GUI; see **X(7)**.

--bse-latency *USECONDS*
: Set the allowed synthesis latency for Bse in milliseconds.

--bse-mixing-freq *FREQUENCY*
: Set the desired synthesis mixing frequency in Hz.

--bse-control-freq *FREQUENCY*
: Set the desired control frequency in Hz, this should be much smaller than the synthesis mixing frequency to reduce CPU load. The default value of approximately 1000 Hz is usually a good choice.

--bse-pcm-driver *DRIVER-CONF*

-p *DRIVER-CONF*
This options results in an attempt to open the PCM driver *DRIVER-CONF* when playback is started. Multiple options may be supplied to try a variety of drivers and unless *DRIVER-CONF* is specified as 'auto', only the drivers listed by options are used. Each *DRIVER-CONF* consists of a driver name and an optional comma seperated list of arguments attached to the driver with an equal sign, e.g.: **-p oss=/dev/dsp2,rw -p auto**

--bse-midi-driver *DRIVER-CONF*

-m *DRIVER-CONF*
This option is similar to the **--bse-pcm-driver** option, but applies to MIDI drivers and devices. It also may be specified multiple times and features an 'auto' driver.

--bse-driver-list
Produce a list of all available PCM and MIDI drivers and available devices.

-- Stop argument processing, interpret all remaining arguments as file names.

DEVELOPMENT OPTIONS

--debug *KEYS*
Enable certain verbosity stages.

--debug-list
List possible debug keys.

-: [*FLAGS*]
This option enables or disables various debugging specific flags for Beast core developers. Use of **-:** is not recommended, because the supported flags may change between versions and cause possibly harmful misbehaviour.

GTK+ OPTIONS

--gtk-debug *FLAGS*
Gtk+ debugging flags to enable.

--gtk-no-debug *FLAGS*
Gtk+ debugging flags to disable.

--gtk-module *MODULE*
Load additional Gtk+ modules.

--gdk-debug *FLAGS*
Gdk debugging flags to enable.

--gdk-no-debug *FLAGS*
Gdk debugging flags to disable.

--g-fatal-warnings
Make warnings fatal (abort).

--sync
Do all X calls synchronously.

SEE ALSO

bse(5), **sfidl(1)**, **Beast/Bse Website**

4.2 BSEWAVETOOL(1)

NAME

bsewavetool - A tool for editing the native multisample format of Beast and Bse

SYNOPSIS

bsewavetool [*tool-options*] command <*file.bsewave*> [*command-arguments*]

DESCRIPTION

Bsewavetool is a command line tool for editing the native multisample format for Beast and Bse, the Bsewave format. Some common operations are creating new Bsewave files, adding chunks to an existing file, encoding the sample data, adding meta information or exporting chunks.

Common uses for Bsewave files are:

- Mapping an individual sample to each midi note (key on the keyboard) - this is mainly useful for drumkits.
- Approximating the sound of an instrument (such as a piano) by sampling some notes, and mapping these to the corresponding frequencies in a Bsewave file - when such a file is loaded by Bse and a note is played, Bse will play the 'nearest' note, and - if necessary - pitch it.

OPTIONS

A number of options can be used with **bsewavetool** in combination with the commands:

-o <***output.bsewave***>
Name of the destination file (default: <file.bsewave>).

--silent
Suppress extra processing information.

--skip-errors
Skip errors (may overwrite Bsewave files after load errors occoured for part of its contents).

-h, --help
Show elaborated help message with command documentation.

-v, --version
Print version information.

COMMANDS

STORE

store

Store the input Bsewave as output Bsewave. If both file names are the same, the Bsewave file is simply rewritten. Allthough no explicit modifications are performed on the Bsewave, externally referenced sample files will be inlined, chunks may be reordered, and other changes related to the Bsewave storage process may occour.

CREATE

create <*n_channels*> [*options*]

Create an empty Bsewave file, <*n_channels*> = 1 (mono) and <*n_channels*> = 2 (stereo) are currently supported.

Options:

-N <***wave-name***>
Name of the wave stored inside of <output.bsewave>.

-f Force creation even if the file exists already.

OGGENC

oggenc [*options*]

Compress all chunks with the Vorbis audio codec and store the wave data as Ogg/Vorbis streams inside the Bsewave file.

Options:

-q <***n***>
Use quality level <*n*>, refer to oggenc(1) for details.

ADD CHUNK

add-chunk [*options*] {**-m**=*midi-note*|**-f**=*osc-freq*} <*sample-file*> ...

Add a new chunk containing <*sample-file*> to the wave file. For each chunk, a unique oscillator frequency must be given to determine what note the chunk is to be played back for. Multi oscillator frequency + sample-file option combinations may be given on the command line to add multiple wave chunks. The -f and -m options can be omitted for a sample file, if the oscillator frequency can be determined through auto extract options.

Options:

-f <***osc-freq***>
Oscillator frequency for the next chunk.
-m <***midi-note***>
Alternative way to specify oscillator frequency.
--auto-extract-midi-note <***nth***>
Automatically retrieve the midi note by extracting the <*nth*> number from the base name of <*sample-file*>.
--auto-extract-osc-freq <***nth***>
Automatically retrieve the oscillator frequency by extracting the <*nth*> number from the base name of <*sample-file*>.

ADD RAW CHUNK

add-raw-chunk [*options*] {**-m**=*midi-note*|**-f**=*osc-freq*} <*sample-file*> ...

Add a new chunk just like with 'add-chunk', but load raw sample data. Additional raw sample format options are supported.

Options:

-R <***mix-freq***>
Mixing frequency for the next chunk [44100].
-F <***format***>
Raw sample format, supported formats are: alaw, ulaw, float, signed-8, signed-12, signed-16, unsigned-8, unsigned-12, unsigned-16 [signed-16].
-B <***byte-order***>
Raw sample byte order, supported types: little-endian, big-endian [little-endian].

DEL CHUNK

del-chunk {**-m**=*midi-note*|**-f**=*osc-freq*|**--chunk-key**=*key*|**--all-chunks**}

Removes one or more chunks from the Bsewave file.

Options:

-f <***osc-freq***>
Oscillator frequency to select a wave chunk.
-m <***midi-note***>
Alternative way to specify oscillator frequency.
--chunk-key <***key***>
Select wave chunk using chunk key from list-chunks.
--all-chunks
Delete all chunks.

XINFO

xinfo {**-m**=*midi-note*|**-f**=*osc-freq*|**--chunk-key**=*key*|**--all-chunks**|**--wave**} **key**=[*value*] ...

Add, change or remove an XInfo string of a Bsewave file. Omission of [*value*] deletes the XInfo associated with the key. Key and value pairs may be specified multiple times, optionally preceeded by location options to control what portion of a Bsewave file (the wave, individual wave chunks or all wave chunks) should be affected.

Options:

-f <***osc-freq***>
Oscillator frequency to select a wave chunk.
-m <***midi-note***>
Alternative way to specify oscillator frequency.
--chunk-key <***key***>
Select wave chunk using chunk key from list-chunks.
--all-chunks
Apply XInfo modification to all chunks.
--wave
Apply XInfo modifications to the wave itself.

INFO

info {**-m**=*midi-note*|**-f**=*osc-freq*|**--chunk-key**=*key*|**--all-chunks**|**--wave**} [*options*]

Print information about the chunks of a Bsewave file.

Options:

-f <***osc-freq***>
Oscillator frequency to select a wave chunk.
-m <***midi-note***>
Alternative way to specify oscillator frequency.
--all-chunks
Show information for all chunks (default).
--chunk-key <***key***>
Select wave chunk using chunk key from list-chunks.
--wave
Show information for the wave.
--pretty=***medium***
Use human readable format (default).
--pretty=***full***
Use human readable format with all details.
--script <***field1***>,<***field2***>,<***field3***>,...,<***fieldN***>
Use script readable line based space separated output.

Valid wave or chunk fields:

channels
Number of channels.
label
User interface label.
blurb
Associated comment.

Valid wave fields:

authors
Authors who participated in creating the wave file.
license
License specifying redistribution and other legal terms.
play-type
Set of required play back facilities for a wave.

Valid chunk fields:

osc-freq
Frequency of the chunk.
mix-freq
Sampling rate of the chunk.
midi-note
Midi note of a chunk.
length
Length of the chunk in sample frames.
volume
Volume at which the chunk is to be played.
format
Storage format used to save the chunk data.
loop-type
Whether the chunk is to be looped.
loop-start
Offset in sample frames for the start of the loop.
loop-end
Offset in sample frames for the end of the loop.
loop-count
Maximum limit for how often the loop should be repeated.

Chunk fields that can be computed for the signal:

+**avg-energy-raw**
Average signal energy (dB) of the raw data of the chunk.
+**avg-energy**
Average signal energy (dB) using volume xinfo.

The script output consists of one line per chunk. The individual fields of a line are separated by a single space. Special characters are escaped, such as spaces, tabs, newlines and backslashes. So each line of script parsable output can be parsed using the **read(P)** shell command. Optional fields will printed as a single (escaped) space.

The human readable output formats (*--pretty*) may vary in future versions and are not recommended as script input.

CLIP

clip {**-m**=*midi-note*|**-f**=*osc-freq*|**--chunk-key**=*key*|**--all-chunks**} [*options*]

Clip head and or tail of a wave chunk and produce fade-in ramps at the beginning. Wave chunks which are clipped to an essential 0-length will automatically be deleted.

Options:

-f <***osc-freq***>
Oscillator frequency to select a wave chunk.
-m <***midi-note***>
Alternative way to specify oscillator frequency.
--chunk-key <***key***>
Select wave chunk using chunk key from list-chunks.
--all-chunks
Try to clip all chunks.
-s=<***threshold***>
Set the minimum signal threshold (0..32767) [16].
-h=<***head-samples***>
Number of silence samples to verify at head [0].
-t=<***tail-samples***>
Number of silence samples to verify at tail [0].
-f=<***fade-samples***>
Number of samples to fade-in before signal starts [16].
-p=<***pad-samples***>
Number of padding samples after signal ends [16].
-r=<***tail-silence***>
Number of silence samples required at tail to allow tail clipping [0].

NORMALIZE

normalize {**-m**=*midi-note*|**-f**=*osc-freq*|**--chunk-key**=*key*|**--all-chunks**} [*options*]

Normalize wave chunk. This is used to extend (or compress) the signal range to optimally fit the available unclipped dynamic range.

Options:

-f <***osc-freq***>
Oscillator frequency to select a wave chunk.
-m <***midi-note***>
Alternative way to specify oscillator frequency.
--chunk-key <***key***>
Select wave chunk using chunk key from list-chunks.
--all-chunks
Try to normalize all chunks.

LOOP

loop {**-m**=*midi-note*|**-f**=*osc-freq*|**--all-chunks**} [*options*]

Find suitable loop points.

Options:

-f <***osc-freq***>
Oscillator frequency to select a wave chunk
-m <***midi-note***>
Alternative way to specify oscillator frequency
--chunk-key <***key***>
Select wave chunk using chunk key from list-chunks
--all-chunks
Try to loop all chunks

HIGHPASS

highpass [*options*]

Apply highpass filter to wave data.

Options:

--cutoff-freq <***f***>
Filter cutoff frequency in Hz
--order <***o***>
Filter order [64]

LOWPASS

lowpass [*options*]

Apply lowpass filter to wave data.

Options:

--cutoff-freq <***f***>
Filter cutoff frequency in Hz
--order <**o**>
Filter order [64]

UPSAMPLE2

upsample2 [*options*]

Resample wave data to twice the sampling frequency.

Options:

--precision <***bits***>
Set resampler precision bits [24]. Supported precisions: 1, 8, 12, 16, 20, 24 (1 is a special value for linear interpolation)

DOWNSAMPLE2

downsample2 [*options*]

Resample wave data to half the sampling frequency.

Options:

--precision <***bits***>
Set resampler precision bits [24]. Supported precisions: 1, 8, 12, 16, 20, 24 (1 is a special value for linear interpolation).

EXPORT

export {**-m**=*midi-note*|**-f**=*osc-freq*|**--chunk-key**=*key*|**--all-chunks**|**-x**=*filename*} [*options*]

Export chunks from Bsewave as WAV file.

Options:

-x <***filename***>
: Set export filename (supports %N %F and %C, see below).

-f <***osc-freq***>
: Oscillator frequency to select a wave chunk.

-m <***midi-note***>
: Alternative way to specify oscillator frequency.

--chunk-key <***key***>
: Select wave chunk using chunk key from list-chunks.

--all-chunks
: Try to export all chunks.

The export filename can contain the following extra information:

%F The frequency of the chunk.
%N The midi note of the chunk.
%C Cent detuning of the midi note.

LIST CHUNKS

list-chunks [*options*]

Prints a list of chunk keys of the chunks contained in the Bsewave file. A chunk key for a given chunk identifies the chunk uniquely and stays valid if other chunks are inserted and deleted.

This bash script shows the length of all chunks (like info --all-chunks):

```
for key in `bsewavetool list-chunks foo.bsewave` ; do
  bsewavetool info foo.bsewave --chunk-key $key --script length ;
done
```

SEE ALSO

beast(1), **Beast/Bse Website**, **Samples and Wave Files in Beast**

4.3 BSE(5)

NAME

BSE - Better Sound Engine File Format

SYNOPSIS

filename.**bse**

DESCRIPTION

The **Bse** file format is used by the **Bse** library and dependent programs to save projects, songs, instruments and sample collections.

FORMAT

Bse files start out with a special magic string `"; BseProject\n"` and then contain nested expressions in scheme syntax using the ASCII charset. Binary data may be appended to a `*.bse` file if it is separated

from the preceeding ASCII text by one or more literal NUL characters (`'\0'`). This mechanism is used to store arbitrary binary data like *WAV* or *Ogg/Vorbis* files in Bse projects, while keeping the actual content user editable - text editors that preserve binary sections have to be used, such as **vi(1)** or **emacs(1)**.

COMPATIBILITY

The exact format and sets of objects and properties used in a Bse file depend highly on the library version that was used to save the file. Compatibility functions are supplied by the library itself, so old Bse files can be converted when the file is loaded. To enable this mechanism, all Bse files contain a `"bse-version"` directive which indicates the Bse file format version of the supplied content.

SEE ALSO

beast(1), **bsewavetool(1)**, **BSE Reference**

4.4 SFIDL(1)

NAME

sfidl - SFI IDL Compiler (Beast internal)

SYNOPSIS

sfidl [*OPTIONS*] *input.idl*

DESCRIPTION

Sfidl generates glue code for Bse objects and plugins from interface definition language files.

OPTIONS

--help [*BINDING*]
: Print general usage information. Or, if *BINDING* was specified, print usage information for this language binding.

--version
: Print program version.

-I *DIRECTORY*
: Add *DIRECTORY* to include path.

--print-include-path
: Print include path.

--nostdinc
: Prevents standard include path from being used.

LANGUAGE BINDINGS:

--client-c
: Generate C client language binding.

--client-c
: Generate C core language binding.

--host-c
: Generate C host language binding.

--client-cxx
: Generate C++ client language binding.

--core-cxx
: Generate C++ core language binding.

--plugin
Generate C++ plugin language binding.

--list-types
Print all types defined in the idlfile. This option is used only for BSE internally to ease transition from C to C++ types.

LANGUAGE BINDING OPTIONS:

--header
Generate header file, this is the default.

--source
Generate source file.

--prefix *prefix*
C host/client language binding option, sets the prefix for C functions. The prefix ensures that no symbol clashes will occur between different programs/libraries which are using a binding, so it is important to set it to something unique to your application/library.

--init *name*
Set the name of the init function for C host/core bindings.

--namespace *namespace*
C++ client language binding, sets the namespace to use for the code. The namespace ensures that no symbol clashes will occur between different programs/libraries which are using a binding, so it is important to set it to something unique to your application/library.

--lower
Select lower case identifiers in the C++ client language binding (create_midi_synth), this is the default.

--mixed
Select mixed case identifiers in the C++ client language binding (createMidiSynth).

SEE ALSO

BSE Reference, **Sfidl Manual**

5 File Formats and Conventions

5.1 The overall structure of BSE files

A `*.bse` file is an Ascii file with occasional binary appendix and contains various data proccessed by the BSE library. The readable (Ascii) part of a `.bse` file is held in lisp syntax and mostly build up out of (keyword value) pairs.

5.1.1 Identification Tag

[NOTE: currently (1999-12-21) the identification tag support is disabled, and subject for general overhaul. A new implementation is likely to come soon and will probably (roughly) follow this specification.]

Each file has a special identification tag in the first line:

```
(BSE-Data V1 0000000004);               -*- scheme -*-
 ^^^^^^^^ ^^ ^^^^^^^^^^
 |        |  |
 |        |  ten-digit octal flag
 |        |
 |        version tag
 |
 BSE data-identifier
```

and is matched against a regexp pattern (in grep -E syntax):

```
^\(BSE-Data\ V1\ [0-7]{10}\);.*$
```

Note that the `';'` at the end of the right parenthesis is actually part of the expression and needs to directly follow the right parenthesis, whereas the space inbetween the `';'` and the line end `'\n'` can contain any number of, or even no characters at all. The ten-digit octal value is a short hand flag to identify the various kinds of data held in the current file. Its value corresponds to the enum definiton BseIoDataFlags in <bse/bseenums.h>.

5.1.2 Important keywords

Main statement keywords - prefixed by a left parenthesis, to start a certain statement - are:

BseSNet
: A source network specification, containing all the sources (filters) contained in the network, their properties and associated link structures.

BseSong
: Holding a song definition including all of a song's data, e.g. instrument definitions, patterns, effects, etc.

BseSample
: A sample definition, usually including references to certain binary blocks that have their binary data appended to the file.

Substatement keywords depend on the object (main statement) they apply to, and should be mostly self explanatory ;)

5.1.3 Binary Appendices

One object invariant substatement keyword is "`BseBinStorageV0`", it's syntax is as follows:

```
( BseBinStorageV0 <UOFFSET> <ENDIANESS>:<N_BITS> <ULENGTH> )
```

<UOFFSET>
: offset within the binary appendix for this block's data

<ENDIANESS>
: either 'L' or 'B' for respectively little or big endianess

<N_BITS>
: number of bits per (endianess-conformant) value stored in this block

<ULENGTH>
: number of bytes contained in this block

Binary appendices are stored after the ascii part of a .bse file. A NUL (`'\0'`) byte character unambiguously marks the end of the ascii portion. Directly following this NUL byte the binary appendix starts out, containing binary data as specified through the various BseBinStorageV0 statements within the ascii portion. Files without binary appendix may not contain the terminating NUL byte.

6 Background & Discussions

Space for notes about the synthesis engine, undo/redo stacks and other technical bits.

6.1 Threading Overview

On a high level, Beast constitutes the front-end to the libbse sound engine.

The BSE sound engine spawns a separate (main) thread separate from the UI for all sound related management and a number of digital signal processing (DSP) sub threads to facilitate parallelized synthesis.

Invocations of the libbse API are automatically relayed to the BSE thread via an IPC mechanism.

The main BSE thread is where the API facing BSE objects live. For actual sound synthesis, these objects spawn synthesis engine modules that process audio buffers from within the DSP threads. The number of parallel DSP threads usually corresponds to the number of cores available to libbse.

Additionally, libbse spawns a separate sequencer thread that generates `note-on`, `note-off` and related commands from the notes, parts and tracks contained in a project. The synchronization required by the sequencer thread with the main thread and the DSP threads is fairly complicated, which is why this is planned to be merged into the DSP threads at some point.

6.2 AIDA - Abstract Interface Definition Adapter

Aida is used to communicate with a regular API between threads and to cross programming language barriers. At the heart of Aida is the IDL file which defines Enums, Records, Sequences and interfaces. Interface methods can be called from one thread and executed on class instances living in other threads. Additionally, callback execution can be scheduled across thread boundaries by attaching them to Event emissions of an instance living in another thread.

Within Beast, Aida is used to bridge between the UI logic (the ”Beast-GUI” or ”EBeast-module” threads) and the sound synthesis control logic (the ”BseMain” thread).

For an IDL interface `Foo`, Aida generates a ”client-side” C++ class `FooHandle` and a ”server-side” C++ class `FooIface`. The `Iface` classes contain pure virtual method definitions that need to be implemented by the server-side logic. The `Handle` classes contain corresponding method wrappers that call the `Iface` methods across thread boundaries. The `Handle` methods are implemented non-virtual to reduce the risk of ABI breakage for use cases where only the client-side API is exposed.

6.2.1 IDL - The interface definition files

6.2.1.1 Namespace

At the outermost scope, an IDL file contains a namespace definition. Interfaces, enums, records and sequences all need to be defined with namespaces in IDL files.

6.2.1.2 Primitive Types

The primitive types supported in IDL files are enums, bool, integer, float and string types. The supported compound types are sequences - a typed variable length array, records - a set of named and typed variables with meta data, Any - a variant type, and inheritable interfaces.

6.2.1.3 Enum

Enum values are defined with integer values and allow `VALUE = Enum (N, strings...)` syntax to add meta data to enum values.

6.2.1.4 Interface

An interface can contain methods and property definitions and inherit from other interfaces. Properties defined on an interface are largely sugar for generating two idiomatic getter and setter access methods, but the set of properties and associated meta data of an interface can be accessed programmatically through the `__access__()` method. The `Bse::Object` implementation makes use of this facility to implement the dynamic property API `set_prop()`, `get_prop()`, `find_prop()`, `list_props()`. The syntax for property definitions with meta data is `TYPE property_name = Auxillary (strings...);`, the `Auxillary()` constructor can take various forms, but ultimately just constructs a list of UTF-8 `key=value` strings that can be retrieved at runtime.

6.2.1.5 Any

An Any is a variant type that can contain any of the other primitive types. It is most useful to represent dynamically typed values in C++ which are common in languages like Python or Javascript. If an Any is set to store an IDL Enum value, it also records the Enum type name, to allow later re-identification. In general, Any should only be needed for languages that do not have dynamic variable types (like C++), and should not normally need to be exposed in dynamic languages (like Python, Javascript). E.g. converting an enum property value stored in an Any to a native type in a dynamic language, a pair could be returned, like `(1, "Bse::FooEnum")`.

7 API Reference

7.1 Namespace Bse

Members declared within the `Bse` namespace.

7.1.1 Bse::init_async()

```
// namespace Bse
void
init_async (int *argc,
            char **argv,
            const char *app_name,
            const StringVector &args);
```

Initialize and start BSE. Initialize the BSE library and start the main BSE thread. Arguments specific to BSE are removed from `argc` / `argv`.

8 Historic Artefacts

Historic notes and writeups that are likely outdated or obsolete.

8.1 Ancient type system description

Description of the BSE types and its type system around 1999, before the migration to GObject.

8.1.1 BSE Structures

8.1.1.1 BseChunk

A BseChunk contains header informations for sample value blocks passed around during sound processing, and a pointer to the associated sample data itself. The pure sample data blocks are referred to as ”Hunk”s. A hunk’s size is determined by the size of each sample value: `sizeof (BseSampleValue)`, the number of tracks contained in a hunk (for instance a stereo hunk contains 2 tracks) and a global variable that designates the block (hunk’s) size for the current sound processing run: `BSE_TRACK_LENGTH` (`bse_globals->track_length`). The sample values themselves are layed out interleaved within the hunk, e.g. a 2 track stereo hunk with a `hunk_length` of 16 would contain:

```
L - left track (1) value
R - right track (2) value
LRLRLRLRLRLRLRLRLRLRLRLRLRLR
```

Each chunk is associated with a certain output channel of a specific source, and applies to a distinct time frame. The actuall time frame length is determined through `BSE_TRACK_LENGTH` and the current mixing frequency `BSE_MIX_FREQ`.

8.1.1.2 BseParam

A parameter that can be get/set on an object. Each parameter has a parameter specification associated with it.

8.1.1.3 BseParamSpec*

A parameter specification similar to an object’s class or to procedure classes, but for a given parameter type multiple parameter specifications may exist (e.g. to implement integer parameters with different ranges). Thus the type system actually implements parameters as ”unclassed” types.

8.1.1.4 BseObject

BseObject implements basic structure operations and provides the necessary cruft for objects to have classes.

8.1.1.5 BseObjectClass

A class associated with each objects instance. A class is referenced by its object instances. introduces per-object parameters.

8.1.1.6 BseProcedure

A BseProcedure is a a special classed-only (i.e. not-object) type that holds parameter specifications in it’s calss structure and can be called. Work is in progress to map certain procedures to certain

object types, so they can be thought of as special object-methods to do perform certain object specific modifications.

8.1.1.7 BsePlugin

A plugin handle that holds information about existing plugins, such as where the actuall plugin's implementation is stored on disk and what types of objects are implemented by the plugin.

Plugin can provide:

- object and class implementations derived from BseEffect or BseSource
- procedure classes
- enum and flags definitions (currently lacks implementation)
- sample/song file loading and saving routines (currently lacks implementation)

8.1.1.8 BseType

type ids are provided for objects & classes, object interfaces, parameters, procedures, enum & flags definitions.

8.1.2 The BSE type system

BSE features a single inheritance object hierarchy with multiple interfaces. It therefore contains a type system that registers several object and non-object types. The type system features unclassed types (parameters), classed types (enumerations and procedures) and object types (also classed). The type system implemented by BSE has several analogies to the Gtk+ type system and several type system related code portions have been traded between the two projects. However, the BSE type system got furtherly enhanced (read: complicated ;) in some regards by featuring dynamic types to allow certain implementations to reside in dynamic plugins.

Usually, all types get registered with their name, description and inheritance information upon startup and for static types the implementation related type information is also stored right away. This information cannot be stored right away for dynamic types since the implementation related type information contains function pointers that will not remaind valid across the life time of plugins (a plugin being loaded, unloaded and the reloaded again will most certainly end up in a different memory location than before). When class or object instances of a dynamic type need to be created, the corresponding plugin will be loaded into memory and the required implementation related type information will be retrieved.

Upon destruction of the instance, the type info is released again and the plugin will be unloaded. So for dynamic types, the only things that stays persistent across plugin life times are the type's name, description, inheritance information and type id, as well as additional information about what plugin this type is implemented by.

Parameter types are unclassed and do not feature inheritance, they are merely variable type ids to distinguish between e.g. strings and integers. Enumeration types are classed, with flags being a certain kind of enumerations in that their values only feature certain bits. For enumertaions, a flat hierarchy is supported, where every enumeration or flags type derives from `BSE_TYPE_ENUM` or `BSE_TYPE_FLAGS` respectively.

Object types are classed as well, and may exist in multiple instances per type id. They feature single inheritance and multiple interfaces, where an interface type denotes a certain alternate class-based vtable that is required to adress similar object methods from different heirarchy branches with the same API. Type inheritance can be examined by testing their `is_a` relationships, e.g. for a type `parent`

and a type `child` deriving from `parent`, the following holds true: `child is_a parent`, while the reverse: `parent is_a child` would fail, because `child` inherits all of `parent`'s facilities, but not the other way around.

Whether a certain object type implements a specific interface, can be tested through the `conforms_to` relationship, e.g. for an interface `I` requirering a method implementation `foo` and a base type `B`, not implementing `foo` with it's two child types `C` and `D`, both carrying different imlpementations of `foo`, the following applies:

- `B conforms_to I`: `FALSE`
- `C conforms_to I`: `TRUE`
- `D conforms_to I`: `TRUE`

We will now outline the several steps that are taken by the type system to create/destroy classes and objects. This is actually the guide line for the type system's implementation, so feel free to skip this over ;) In the below, unreferencing of certain objects may automatically cause their destruction, info structure retrival and releasing may (for dynamic types) accompany plugin loading/unloading.

8.1.2.1 initialization

- all static type ids + type infos get registered
- all dynamic type ids get registered (implies: all interface type ids are registered)
- static type infos are persistant

8.1.2.2 class creation

- type info is enforced
- parent class is referenced
- class gets allocated and initialized

8.1.2.3 object creation

- class is referenced
- object is allocated and initialized

8.1.2.4 object destruction

- object gets deallocated (destruction has already been handled by the object system)
- class gets unreferenced

8.1.2.5 class destruction

- class gets destructed and deallocated
- type info is released
- parent class is unreferenced

8.1.2.6 interface registration

- interface id gets embeded in the host type's interface entry list
- the interface is registered for all children as well

8.1.2.7 interface class creation

- class is given
- class is referenced
- interface info is enforced
- interface is referenced

- interface entry info is enforced
- interface class is allocated and initialized (and propagated to children)

8.1.2.8 interface class destruction

- interface class gets destructed and deallocated (and propagated to children)
- interface entry info is released
- interface gets unreferenced
- interface info is released
- class gets unreferenced

8.1.2.9 BSE Sources

Each source can have an (conceptually) infinite amount of input channels and serve a source network (BseSNet) with multiple output channels. Per source, a list of input channels is kept (source pointer and output channel id, along with an associated history hint). Sources also keep back links to other sources using its output channels, this is required to maintain automated history updates and purging.

8.1.2.10 BSE Source Networks

A BSE Source Network is an accumulation of BseSources that are interconnected to controll sample data flow. The net as a whole is maintained by a BseSNet object which keeps references of all sources contained in the net and provides means to store/retrieve undo/redo information for the whole net and implements the corresponding requirements to save and load the complete net at a certain configuration.

Implementation details:

- a BseSNet keeps a list and references of all sources contained in the net.
- each source can have a (conceptually) infinite amount of input channels and serve the net with multiple output channels.
- per source, a list of input channels is kept (source pointer plus output channel id, along with an associated history hint).
- sources also keep back links to sources using its output channels, this is required to maintain automated history updates.

8.2 BSE category description

Description of the BSE categories around 2003, before the removal of procedures.

8.2.1 BSE Categories

The purpose of BSE Categories is to provide a mapping of functionality provided by BSE and its plugins to GUI frontends. Categories are used to provide a hirarchical structure, to group certain services and to aid the user in finding required functionality.

8.2.1.1 Object Categories

BSE objects and procedures are ”categorized” to easily integrate them into menu or similar operation structures in GUIs.

Objects that are derived from the BseSource class with the main intend to be used as synthesis modules in a synthesis network are categorized under the `/Modules/` toplevel category, ordered by operational categories.

8.2.1.2 Core Procedures

Procedures provided by the BSE core are categorized under a subcategory of the `/Methods/` toplevel category. The object type a procedure operates on is appended to the toplevel category to form a category name starting with `/Methods/<ObjectType>/`. Procedures implementing object methods need to take the object as first argument and follow a special naming convention (for the procedure name). To construct the procedure type name, the object type and method named are concatenated to form `<ObjectType>+<MethodName>`. This ensure proper namespacing of method names to avoid clashes with equally named methods of distinct object types. Some special internal procedures may also be entered under the `/Proc/` toplevel category.

8.2.1.3 Utility Procedures

Procedures implementing utility functionality, usually implemented as scripts, are entered into one of the below listed categories. Depending on the category, some procedure arguments are automatically provided by the GUI, the name and type of automatic arguments are listed with the category in the following table:

Table 1: Set of toplevel categories used by libbse.

Category	Automatic arguments as (name:type) pairs
/Project/	project:BseProject
/SNet/	snet:BseSNet
/Song/	song:BseSong
/Part/	part:BsePart
/CSynth/	csynth:BseCSynth
/WaveRepo/	wave_repo:BseWaveRepo
/Wave/	wave:BseWave

The set of automatic arguments provided upon invocation is not restricted by te above lists, so caution should be taken when naming multiple arguments to a procedure `auto_*`.

8.2.1.4 Aliasing/Mapping

A certain BSE type (procedure or object) may be entered under several categories simultaneously, thus allowing multiple categories to refer (alias) to the same type (functionality).

8.2.1.5 Examples

Table 2: Category examples from the Beast source code.

Category	TypeName	Comment
/Methods/BseProject/File/Store	BseProject + store-bse	BseProject procedure
/Modules/Spatial/Balance	BseBalance	synthesis module
/SNet Utility/Utils/Grid Align	modules2grid	scheme script

8.3 BseHeart - Synthesis Loop, Dec 1999

BSE's heart is a unique globally existing object of type BseHeart which takes care of glueing the PCM devices and the internal network structure together and handle syncronization issues.

BseHeart brings the internal network to work by incrementing a global BSE index variable, cycling all active BseSources to that new index, collecting their output chunks and writing them into the output PCM devices. As of now there are still some unsolved syncronization problems, such as having multiple PCM devices that run at different speeds. While this could be solved for multiples/fractions of their mixing frequencies, such as one device (card) running at 44100Hz and another one at 22050Hz, problems still remain for slight alterations in their oscilaltors if a device e.g. runs at 44099Hz (such as the es1371 from newer PCI SB cards).

For the time being, we do not handle different output/input frequencies at all and use our own syncronization within BSE, bound to GLib's main loop mechanism by BseHeart attaching a GSource.

To support user defined latencies, PCM Devices implement their own output buffer queue which is used for temporary storage of output buffers if the specified latency forces a delay greater than what can be achived through the device driver's queue. The devices also maintain an input queue for recorded data to avoid overruns in the PCM drivers, ideally these queues will immediatedly be emptied again because output is also required.

FIXME: A check may be necessary to catch indefinitely growing input queues due to input->output frequency differences. We will also need to catch input underruns here, though handling that is somwhat easier in that we can simply slide in a zero pad chunk there.

The fundamental BSE cycling (block-wise processing of all currently active sources) is bound to the PCM devices output requirements, according to the current latency setting. Integrated into GLib's main loop, we perform the following steps:

prepare():
: We walk all input devices until one is found that reports requirement to process incoming data. We walk all output devices until one is found that reports requirement to process outgoing data or needs refilling of its output queue, according to the latency settiongs. If none is found, the devices report an estimated timeout value for how long it takes until they need to process further data. *[Implementation note: as an optimization, we walk all PCM devices in one run and leave the distinction between input and output up to them]* The minimum of the reported timeout values is used for GLib's main loop to determine when to check for dispatching again.

check():
: We simply do the same thing as in prepare() to figure whether some PCM devices need processing and if necessary request dispatching.

dispatch():
: We walk the output devices in order to flush remaining queued chunks, and collect reports on whether they need new input blocks according to the latency setting. We walk the input devices so they can process incoming data (queue it up in chunks). *[Implementation note: as an optimization, we currently do these two walks in one step, leaving the details to the PCM devices]* Now here's the tweak: if *any* output device reports the need for further input data, we perform a complete BSE cycle. If no output devices require further input data, we walk the input devices and check whether they have more than one chunk in their queue (and if necessary, fix that) this is required to catch overruns and constrain output latency.

cycle():
: For input devices that have empty queues, we report underruns and pad their queues with zero chunks, i.e. we ensure all input PCM devices have at least one usable chunk in their input queue.

We increment the BseIndex of all currently active BseSources, calc their output chunks and queue them up for all output sources connected to them. We walk the input devices and remove exactly one chunk from their input queues.

A Appendix

A.1 One-dimensional Cubic Interpolation

With four sample values V_0, V_1, V_2 and V_3, cubic interpolation approximates the curve segment connecting V_1 and V_2, by using the beginning and ending slope, the curvature and the rate of curvature change to construct a cubic polynomial.

The cubic polynomial starts out as:

(1) $f(x) = w_3x^3 + w_2x^2 + w_1x + w_0$

Where $0 <= x <= 1$, specifying the sample value of the curve segment between V_1 and V_2 to obtain.

To calculate the coefficients $w_0, ..., w_3$, we set out the following conditions:

(2) $f(0) = V_1$
(3) $f(1) = V_2$
(4) $f'(0) = V_1'$
(5) $f'(1) = V_2'$

We obtain V_1' and V_2' from the respecting slope triangles:

(6) $V_1' = \frac{V_2 - V_0}{2}$
(7) $V_2' = \frac{V_3 - V_1}{2}$

With (6) → (4) and (7) → (5) we get:

(8) $f'(0) = \frac{V_2 - V_0}{2}$
(9) $f'(1) = \frac{V_3 - V_1}{2}$

The derivation of $f(x)$ is:

(10) $f'(x) = 3w_3x^2 + 2w_2x + w_1$

From $x = 0 \rightarrow$ (1), i.e. (2), we obtain w_0 and from $x = 0 \rightarrow$ (10), i.e. (8), we obtain w_1. With w_0 and w_1 we can solve the linear equation system formed by (3) → (1) and (5) → (10) to obtain w_2 and w_3.

(11) (3) → (1): $w_3 + w_2 + \frac{V_2 - V_0}{2} + V_1 = V_2$
(12) (5) → (10): $3w_3 + 2w_2 + \frac{V_2 - V_0}{2} = \frac{V_3 - V_1}{2}$

With the resulting coefficients:

$$
\begin{aligned}
w_0 &= V_1 && (\textit{initial value}) \\
w_1 &= \frac{V_2 - V_0}{2} && (\textit{initial slope}) \\
w_2 &= \frac{-V_3 + 4V_2 - 5V_1 + 2V_0}{2} && (\textit{initial curvature}) \\
w_3 &= \frac{V_3 - 3V_2 + 3V_1 - V_0}{2} && (\textit{rate change of curvature})
\end{aligned}
$$

Reformulating (1) to involve just multiplications and additions (eliminating power), we get:

(13) $f(x) = ((w_3x + w_2)x + w_1)x + w_0$

Based on $V_0, ..., V_3$, $w_0, ..., w_3$ and (13), we can now approximate all values of the curve segment between V_1 and V_2.

However, for practical resampling applications where only a specific precision is required, the number of points we need out of the curve segment can be reduced to a finite amount. Lets assume we require n equally spread values of the curve segment, then we can precalculate n sets of $W_{0,...,3}[i]$, $i = [0, ..., n]$, coefficients to speed up the resampling calculation, trading memory for computational performance. With $w_{0,...,3}$ in (1):

$$\begin{aligned} f(x) = & \frac{V_3 - 3V_2 + 3V_1 - V_0}{2}x^3 + \\ & \frac{-V_3 + 4V_2 - 5V_1 + 2V_0}{2}x^2 + \\ & \frac{V_2 - V_0}{2}x + \\ & V1 \end{aligned}$$

sorted for $V_0, ..., V_4$, we have:

(14)

$$\begin{aligned} f(x) = & V_3\,(0.5x^3 - 0.5x^2) + \\ & V_2\,(-1.5x^3 + 2x^2 + 0.5x) + \\ & V_1\,(1.5x^3 - 2.5x^2 + 1) + \\ & V_0\,(-0.5x^3 + x^2 - 0.5x) \end{aligned}$$

With (14) we can solve $f(x)$ for all $x = \frac{i}{n}$, where $i = [0, 1, 2, ..., n]$ by substituting $g(i) = f(\frac{i}{n})$ with

(15) $g(i) = V_3W_3[i] + V_2W_2[i] + V_1W_1[i] + V_0W_0[i]$

and using n precalculated coefficients $W_{0,...,3}$ according to:

$$\begin{aligned} m &= \frac{i}{n} \\ W_3[i] &= 0.5m^3 - 0.5m^2 \\ W_2[i] &= -1.5m^3 + 2m^2 + 0.5m \\ W_1[i] &= 1.5m^3 - 2.5m^2 + 1 \\ W_0[i] &= -0.5m^3 + m^2 - 0.5m \end{aligned}$$

We now need to setup $W_{0,...,3}[0, ..., n]$ only once, and are then able to obtain up to n approximation values of the curve segment between V_1 and V_2 with four multiplications and three additions using (15), given $V_0, ..., V_3$.

A.2 Modifier Keys

There seems to be a lot of inconsistency in the behaviour of modifiers (shift and/or control) with regards to GUI operations like selections and drag and drop behaviour.

According to the Gtk+ implementation, modifiers relate to DND operations according to the following list:

Table 3: GDK drag-and-drop modifier keys

Modifier		Operation	Note / X-Cursor
none	→	copy	(else move (else link))
SHIFT	→	move	GDK_FLEUR
CTRL	→	copy	GDK_PLUS, GDK_CROSS
SHIFT+CTRL	→	link	GDK_UL_ANGLE

Regarding selections, the following email provides a short summary:

```
From: Tim Janik <timj@gtk.org>
To: Hacking Gnomes <Gnome-Hackers@gnome.org>
Subject: modifiers for the second selection
Message-ID: <Pine.LNX.4.21.0207111747190.12292-100000@rabbit.birnet.private>
Date: Thu, 11 Jul 2002 18:10:52 +0200 (CEST)

hi all,

in the course of reimplementing drag-selection for a widget,
i did a small survey of modifier behaviour in other (gnome/
gtk) programs and had to figure that there's no current
standard behaviour to adhere to:

for all applications, the first selection works as
expected, i.e. press-drag-release selects the region
(box) the mouse was draged over. also, starting a new
selection without pressing any modifiers simply replaces
the first one. differences occour when holding a modifier
(shift or ctrl) when starting the second selection.

Gimp:
Shift upon button press:        the new seleciton is added to the existing one
Ctrl upon button press:         the new selection is subtracted from the
                                existing one
Shift during drag:              the selection area (box or circle) has fixed
                                aspect ratio
Ctrl during drag:               the position of the initial button press
                                serves as center of the selected box/circle,
                                rather than the upper left corner

Gnumeric:
Shift upon button press:        the first selection is resized
Ctrl upon button press:         the new seleciton is added to the existing one

Abiword (selecting text regions):
Shift upon button press:        the first selection is resized
Ctrl upon button press:         triggers a compound (word) selection that
                                replaces the first selection

Mozilla (selecting text regions):
```

```
Shift upon button press:        the first selection is resized

Nautilus:
Shift or Ctrl upon buttn press: the new selection is added to or subtracted
                                from the first selection, depending on whether
                                the newly selected region was selected before.
                                i.e. implementing XOR integration of the newly
                                selected area into the first.

i'm not pointing this out to start a flame war over what selection style
is good or bad and i do realize that different applications have
different needs (i.e. abiword does need compound selection, and
the aspect-ratio/centering style for gimp wouldn't make too much
sense for text), but i think for the benfit of the (new) users,
there should me more consistency regarding modifier association
with adding/subtracting/resizing/xoring to/from existing selections.

---
ciaoTJ
```