Esbeekay v1.1

This is a brief guide to Esbeekay. If you have questions or bug-reports, please send them to me (Ari) at apl@vapl.demon.co.uk or on Compuserve 100042,3166.

Introduction to Esbeekay

- **Quick start to making SBKs**
- **Quick start to using SBKs**

The structure of an SBK file

- **Parameter Notes**

Editing SoundFont Banks

- **Testing patches**

Importing data

Macro Language and Library

Notes

INI settings

Many thanks to all those who have helped with debugging, testing, ideas, and parameters. Particular thanks to Andy Robinson and Jay Vaughan.

Macro Language and Library

Outline

Macro Windows

Language Description

Language Grammar

Library Functions

Macro Callbacks

Macro callbacks

Certain functions, if they exist in the macro, will be called automatically by Esbeekay at certain points. These are:

init()

This function must have no arguments, and will be called when the macro is started.

notifyPatchChange(sbk)

This function must have one argument: an sbk reference. It will be called whenever there is a change in the patches in an SBK.

notifyInstChange(sbk)

This function must have one argument: an sbk reference. It will be called whenever there is a change in the instruments in an SBK.

notifySampleChange(sbk)

This function must have one argument: an sbk reference. It will be called whenever there is a change in the samples in an SBK.

Typically, **init** is used to initialize the macro window, and **notifyPatchChange**, **notifyInstChange**, and **notifySampleChange** are used to maintain the consistency of lists of items which are displayed in the macro window.

Macro Windows

Every macro has a window defined. A macro can run with the window open or closed, and when Esbeekay starts up it will start each macro with the window in the state it was when Esbeekay was last shut down.

Each window can contain elements such as buttons and edit controls. All elements are the usual Windows dialog controls, except for the following differences:

Buttons

These can be defined to contain a .bmp graphic or text. The .bmp file is only used for the definition phase - after the button has been defined, the contents of the file will be included in the macro and the separate file is no longer needed.

Sliders

These are exactly the same as in Esbeekay dialogs.

SBKLinks

This type of control is used to link open SBK files to the macro. The SBKLink is operated by either clicking or dragging. Clicking sets the link to point to a SBK file, if only one file is open. Otherwise it will not point to any window. Dragging is used to point the link to any open SBK file, the borders of the windows will be highlighted as the user drags the link to point to a SBK window.

NoteIn

This is an edit control, but it only accepts values between 0 and 127. If focus is placed in the edit, the edit will accept MIDI notes as input.

ValueIn

This is an edit control, but it will enforce a value range for its input which must be a number.

The macro window is designed using the window designer started from the View/Macros... dialog. Each control can have properties such like fonts and colours adjusted if appropriate.

Each macro window control will have a textual identifier. This is referred to as its id. ids are used in macro library functions to refer to controls. They also have a special function for buttons, SBKLinks and radio buttons: whenever one of these is clicked, if a function exists in the macro with the same name as the control id, that function will be called. The function should not have any arguments. So, for instance if there is a button with an id of load (excluding the quotes), then if there is a function in the macro load() then that function will be called.

Outline of macros in Esbeekay

Esbeekay macros consist of a window and a macro program, stored in a single file with a .mac extension. The window can contain controls such as buttons, list boxes, sliders, text fields etc. The macro program can control the window and functions in the macro program are called in response to events in the window (such as buttons being clicked). The macro window will appear as a pop-up window which can be hidden.

Whenever Esbeekay starts, it loads the installed macros (see View/Macros...). These macros are automatically started. It is possible to define a function in the macro program which gets called when the macro starts so that the macro can initialize the macro window with suitable defaults. One physical macro (i.e. the .mac file) can be installed many times and each installed instance of the macro will be independent.

A macro program can manipulate objects in SBK files at the lowest level. They can also perform tasks such as importing data into SBK files, and copying data between SBK files.

A macro program (when executing) can be interrupted at any time by pressing CTRL-C.

Macro language description

General

The macro language in Esbeekay is a procedural language with a dynamic typing system. Local and global variables are supported. Functions can be defined by the user, and there is a library of built-in utility functions for manipulating SBKs. In syntax, the language resembles both C and Pascal. There is an extensive type system and dynamic data allocation with re-use, which is automatically maintained by the system.

Type system

It is necessary to understand the organization of an SBK file to understand the type system. The fundamental data types supported are:

1. number

   This type represents numbers, both long integers and double precision floating point numbers. The distinction between the two is maintained internally by the system, and conversion to doubles occur automatically when necessary. This means that integer accuracy is maintained as long as possible.

2. string

   Strings can be up to 32767 bytes long.

3. list

   List types contain elements, which can be indexed. Each element can be a different type. Thus a list could contain numbers, strings, and other lists.

4. sbk

   This type refers to an SBK. The SBK may be an internal one, or it may be one that is open in a window in Esbeekay.

5. patch

   This type refers to a patch within an SBK.

6. inst

   This type refers to an instrument within an SBK.

7. sample

   This type refers to a sample header within an SBK.

8. conpi

   This type refers to a connection from a patch to an instrument.

9. conis

   This type refers to a connection from an instrument to a sample.

Language constructs

Comments

Comments can be written anywhere in a macro by prefixing them with a % which makes everything on that line to the right of the % a comment.

Program

A macro program starts with an optional declaration of global variables (i.e. global within that macro instance). This is followed by function definitions. Function definitions contain a declaration of local variables, and a set of commands which are executed sequentially.

Global variables

The declaration of global variables takes the following form:

```
var      <variable>[, <variable>]* ;
```

Where <variable> is a variable name. Therefore the declaration

```
var      a, b, c;
```

declares three global variables. At the declaration stage, variables are NOT typed. Variables can take on any valid type, and can change type at run-time. The typing of a variable is therefore completely dynamic.

Functions

Functions take the following form:

```
<function-name> ( <parameter-name> [, <parameter-name> ]* )
{
  var <variable>, [<variable>]* ;
  <commands>
}
```

So for instance

```
factorial(n)
{
  if (n = 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

Is a valid definition of the factorial function. Note that the return type (if any) of the function is not declared, since a function may return different types (!). The types of parameters (or local variables were there any) are also not defined. Parameters to the function take on the values and types that are passed into the function, and local variables start out uninitialized. Local variables disappear when the function is exited - their scope is the function. Recursion is allowed and local variables behave accordingly.

Commands

Blocks
{ commands }
A block of commands
A block enclosed in {}s is syntactically equivalent to one command.
Use {}s to enclose a block of commands to execute e.g. in while loops.

Assignment
<variable> := expression
This assigns expression to <variable>, which takes on the type of the expression.

While-loops
while (expression) command
This executes the given command or block while the expression is non-zero.
The expression is tested at the beginning of the command only.

Conditionals
if (expression) command1
if (expression) command1 else command2
Executes command1 if expression is non-zero, otherwise command2 if given.

Function return
return
return expression
Returns from a function, optionally passing a return value.

List element enumeration
for <variable> in expression do command
Enumerates all elements in expression (which has to evaluate to a list). <variable> takes on the value of each element in turn and the command is executed for each iteration.

Expressions

Expressions are values. They can be formed from the following basic elements:

variables	By giving the variable name
numbers	Written in decimal or hex using the notation 0xN where N is the number.
strings	Enclosed in double quotes, e.g. abc
lists	Enclosed in {s and separated by commas, e.g. { 1, x, 3 }
function calls	By giving the name and parameters, e.g. factorial(3)

Expressions (including basic elements) can then be combined using operators. These are:

+	Applied to numbers: adds the numbers Applied to lists: concatenates lists or adds elements to lists Applied to strings: concatenates strings
-, *, /	Can only be applied to numbers
=, <, >, <=, >=, !=	Comparison operators = and != can be applied to any data type All other comparisons to numbers and strings only
!	The NOT operator
&	The AND operator

|	The OR operator
(and)	Parantheses for grouping
[ ]	Indexing of lists, e.g. a[3]

Macro language grammar

White space between identifiers is permitted. Comments are excluded from the grammar. Comments are everything to the right of % on a line.

program := varlist pdefs

pdefs := ε | definition pdefs

definition := function | callback

callback := callback <function-name> fbody

function := <function-name> (arg_list) fbody

arg_list := ε | arg_list2

arg_list2 := <variable> | <variable> , arg_list2

fbody := { varlist cmdlist }

varlist := ε | var arg_list2 ;

cmdlist := ε | cmd cmdlist

cmd := assignment |
fn_call |
while_loop |
if_cmd |
enum_list |
return_cmd |
{ cmdlist }

return_cmd := return ; | return expr ;

enum_list := for <variable> in expr do cmd

if_cmd := if (expr) cmd | if (expr) cmd else cmd

while_loop := while (expr) cmd

fn_call := <function-name> (fargs) ;

assignment := <variable> := expr ;

list_expr := { list2 }

list2 := ε | list3

list3 := expr | expr , list3

expr := <number> |
<variable> |
<string> |
list_expr |

		(expr) |
		expr [expr] |
		- expr |
		expr - expr |
		expr + expr |
		expr * expr |
		expr / expr |
		expr = expr |
		expr < expr |
		expr > expr |
		expr <= expr |
		expr >= expr |
		expr != expr |
		expr & expr |
		expr | expr |
		! expr |
		<function-name> (fargs)
fargs	:=	ε | fargs2
fargs2	:=	expr | expr , fargs2

Macro language library functions

Function parameter and return value syntax rules

I/O functions

- beep
- message
- filePrompt
- importWave
- importGUS
- importKrz
- callDLL

Object traversal functions

- getSBKPatches
- getSBKInstruments
- getSBKSamples
- getPatchToConPIs
- getConPIToInst
- getInstToConISs
- getConISToSample
- getSampleToConISs
- getInstToConPIs
- getConISToInst
- getConPIToPatch

Object creation functions

- createPatch
- createInst
- createSample
- createConPI
- createStdConPI
- createConIS
- createStdConIS

Object manipulation

- getObjectData
- setObjectData
- deleteObject
- copyFromSBK

Parameter manipulation

- setParam
- getParam
- removeParam
- getDefaultPIParams
- getDefaultISParams
- getAllParams
- setAllParams
- removeAllParams
- makeKeyRange
- makePitch
- makePitchAndRoot
- decodeKeyRange

Window control functions

- getSBKLink
- setSBKLinkManual
- windowInput
- windowOutput

Sample space functions

- delSBKData
- addSBKData
- getSBKDataLen
- notifySBKDataChanged

General functions

- valid
- filenameFromPath
- numberToString
- listLength

Parameter and return value syntax rules

The syntax for showing parameter and return value types is the following:

1. If a direct type is shown, only a value of that type is possible.
2. If a set is shown as in [a | b] then either a or b are possible types.
3. If a list is shown as in {a, b} then only a list containing a and b is possible.
4. If a list is shown as in {a*} then a list with any number of as is possible.
5. If the parameter is of type *any*, then any type is possible.
6. If the return value is *void,* the function does not return a value.

number callDLL(dllname::string, fname::string, {par::any*})

This function will call the function *fname* in the DLL specified by *dllname,* passing as parameter to the function a string which is the textual representation of all values in *par,* separated by spaces. The parameter *dllname* should not specify a directory name, just the filename of the DLL. The directory will be automatically be added by Esbeekay, and will be the directory from which the calling macro was loaded. The function returns the number returned by the DLL function.

The form of the function in the DLL should be the following:

int far pascal __export fname(LPSTR args)

(this is for MS C)

The arguments passed in the parameter *par* will be individually converted into strings and concatenated (but with separating spaces) to form the argument *args* to the DLL. The data types that can be converted into strings are:

string Will be converted into the string but will be between double quotes.

number Will be converted into a straightforward signed number, upto 32 bits in precision, or a number with a decimal point if the 32-bits precision is not enough.

sbk Will be converted into a number which will be the handle of the global memory block which contains the sample data for the SBK. The DLL can then lock the memory block and modify the sample data, you must unlock the block again before returning from the DLL.

{sample*} getSBKSamples(sbkref::sbk)

Returns a list of all samples in the SBK specified by *sbkref.*

{inst*} getSBKInstruments(sbkref::sbk)

Returns a list of all instruments in the SBK specified by *sbkref.*

void notifySBKDataChanged(sbkref::sbk)

This function will cause all displays in Esbeekay which refer to the sample space block of SBK *sbkref* to be updated. It is necessary to call this function if you have called a DLL function which modifies the sample data.

number getSBKDataLen(sbkref::sbk)

Returns the length of the data block in SBK *sbkref,* the size is measured in bytes.

number addSBKData(sbkref::sbk, len::number)

This function will add an empty area of sample space *len* bytes in length to the data block of the SBK specified by *sbkref.* The area will be initialized to zero.

The return value is the offset of the starting byte of the new area.

void delSBKData(sbkref::sbk, from::number, len::number)

Deletes a range starting with *from* which is *len* bytes long of the data block from SBK *sbkref*. Any sections of that range which are used by any RAM samples in *sbkref* will NOT be deleted.

number listLength(l::list)

Returns the length of list *l*.

{patch*} importKrz(sbkref::sbk, filename::string, patchStart::number, patchBank::number)

This function will import a .krz file specified by *filename* into the SBK specified by *sbkref.* The patches created on the import will be in bank *patchBank,* and will be numbered from *patchStart* onwards. If *patchBank* is 128, then the patches will be percussion patches.

When importing a Kurzweil patch file, the structure created will be one or more complete patch->instrument->sample links, with some instruments possibly being shared by patches which in the Kurzweil file are mapped using the same key map. The patches, instruments, and sample headers will be named with their respective index numbers as they as in the .krz file but the patch numbers will be modified to start from *patchStart.*

The function will return a list of the patches created (usually, this will just contain one patch reference) but if an error occurs, will return an invalid value.

{patch*} importGUS(sbkref::sbk, filename::string, patchStart::number, patchBank::number)

This function will import a .pat file specified by *filename* into the SBK specified by *sbkref.* The patches created on the import will be in bank *patchBank,* and will be numbered from *patchStart* onwards. If *patchBank* is 128, then the patches will be percussion patches.

When importing a GUS patch file, the structure created will be one or more complete patch->instrument->sample links, with some sample data possibly being shared by sample headers which in the GUS patch are duplicates. The patches, instruments, and sample headers will be named with their respective index numbers as they as in the .pat file but the patch numbers will be modified to start from *patchStart.*

The function will return a list of the patches created (usually, this will just contain one patch reference) but if an error occurs, will return an invalid value.

void beep()

Emits a beep.

void message(text::string)

Displays a message box with the given text.

[{string*} | string] filePrompt(typeList::string, allowMultiple::number, isOpen::number, initDir::string)

Displays a file prompt dialog. The parameters are as following:

typeList: a string which gives the file type filter and file type information. List filter descriptions and the filters themselves in this string, separated by |s. The end of the string should be indicated by ||, for example: "Wave files (*.wav)| *.wav||"

allowMultiple: if this parameter is non-zero, multiple selections are allowed.

isOpen: if this parameter is non-zero, the file dialog will be an Open... dialog. If the parameters is zero, the dialog will be a Save... dialog.

initDir: if this parameter is , then the dialog will be initialized from the current directory. Otherwise this parameter should specify the initial directory.

The return value from this function will be either a filename as a string, if no multiple selection is allowed, or if multiple selection is allowed, then a list of strings will be returned. The return value should be tested with **valid()** since if the users clicks on Cancel, the return value will not be valid.

sbk getSBKLink(ControlID::string)

Returns a reference to the SBK that the SBKLink is pointing to, if any. The value returned by this function should always be checked using **valid()**. The return value will not be valid if the SBKLink is not pointing to some SBK.

void setSBKLinkManual(ControlID::string)

Normally SBKLinks work so that they always try and maintain a link to at least some SBK, and only give up if there is more than one possible SBK to link to. This will switch off the automatic linking, and make the SBKLink behave so that the user will always have to manually link it to an SBK.

number valid(value::any)

This function will return a non-zero value to indicate the the parameter is valid.

{patch*} getSBKPatches(sbkref::sbk)

This function will return a list of the patches in the sbk specified by *sbkref.*

inst importWave(sbkref::sbk, filename::string, nameBase::string)

This function will import a .wav file specified by *filename* into the SBK specified by *sbkref*, and will name the base of the instrument and samples created using *nameBase* as the root name. When importing a mono wave, an instrument connected to a sample will be created. When importing a stereo wave, an instrument connected to two appropriately panned samples will be created. The function will return the instrument that was created.

string filenameFromPath(path::string)

This function will return the filename from a pathname specified by *path.*

patch createPatch(sbkref::sbk, name::string, nbr::number, bank::number)

This function will create a patch in the SBK specified by *sbkref,* which will have the name specified by *name,* and the patch number and bank number specified by *nbr* and *bank.* If *bank* is 128, then the patch will be a percussion patch. This function will NOT check that a patch with the same number and bank does not already exist in the SBK. The function will return the patch that was created.

inst createInst(sbkref::sbk, name::string)

This function creates an instrument with the name *name* in the SBK specified by *sbkref.* The function will return the instrument which was created.

sample createSample(sbkref::sbk, name::string, isRom::number, start::number, end::number, loopStart::number, loopEnd::number)

This function creates a sample header in the SBK specified by *sbkref.* It will not actually create any sample data. The sample header will have the name *name.* The sample will be located from *start* to *end. start* must be an even number, and *end* an odd number. The loop in the sample is located from *loopStart* to *loopEnd. loopStart* must be even and *loopEnd* must be odd. If *isRom* is non-zero then the sample locations are in AWE32 ROM, otherwise they refer to RAM.

Both RAM and ROM addresses start at zero. ROM addresses, assuming that it is 1MB in length end at 1MB. The upper boundary for RAM addresses will depend on how much sample data is associated with the SBK specified by *sbkref.* Samples can be overlaid both in ROM and RAM so that sample data may be included in the sample area of more than one sample header. To discover the addresses for ROM samples, have a look at the standard SBKs: sample dialogs in Esbeekay will give the addresses of the ROM samples which are used in those SBKs. It is unknown what will happen if you address samples in the 3MB which presumably are between the ROM and the start of the RAM - dont try it, because I wont take responsibility for anything that happens!

If *isRom* is non-zero and *start, end, loopStart and loopEnd* are all zero, then the sample header is said to be an inactive sample header and will not refer to any memory location at all.

The function will return the sample header which was created.

{number*} getDefaultPIParams()

This function will return the default parameters for connections from patches to instruments. The parameters are represented by a list of numbers.

conPI createConPI(fromPatch::patch, toInst::inst, params::{number*})

This function will create a connection from a patch to an instrument. The connection will be from *fromPatch* to *toInst* and will have the parameters given by *params.* You can get a set of default parameters by using **getDefaultPIParams()**.

The function will return the connection which was just created.

conPI createStdConPI(fromPatch::patch, toInst::inst)

This function will create a connection from a patch *fromPatch* to an instrument *toInst.* The connection will have the default parameters for a connection of that type.

The function will return the connection which was just created.

{conpi*} getPatchToConPIs(p::patch)

This function will return all the connections starting from a patch *p.* The function return value will be a list of such connections.

inst getConPIToInst(c::conpi)

This function will return the instrument which the connection *c* leads to.

{conis*} getInstToConISs(i::inst)

This function will return all the connections starting from an instrument *i.* The function return value will be a list of such connections.

sample getConISToSample(c::conis)

This function will return the sample which the connection *c* leads to.

{number*} getAllParams(obj::[patch | inst | conpi | conis])

This function will return the parameters attached to *obj*.

void setAllParams(obj::[patch | inst | conpi | conis], pars::{number*})

This function will set the parameters for the object *obj* to *pars.*

void removeAllParams(obj::[patch | inst])

This function removes all parameters from the object *obj.*

void setParam(obj::[patch | inst | conpi | conis], parNumber::[number | string], parValue::number)

This function sets the parameter indicated by *parNumber* to *parValue* for the object *obj.* *parNumber* can be a number, or it can be a string containing one of the standard parameter names (see parameter names).

number getParam(obj::[patch | inst | conpi | conis], parNumber::[number | string])

This function returns the parameter indicated by *parNumber* for the object *obj. parNumber* can be a number, or it can be a string containing one of the standard parameter names (see parameter names). If that particular parameter does not exist for that object, then an invalid value will be returned so you should always check the result of this function with **valid()** unless you know that the parameter will be there.

void removeParam(obj::[patch | inst | conpi | conis], parNumber::[number | string])

This function will remove parameter *parNum* from the object *obj. parNumber* can be a number, or it can be a string containing one of the standard parameter names (see parameter names). Eliminating a parameter gives the default behaviour for the object for that parameter.

any windowInput(controlId::string)

This function will input a value from the named macro window control *controlId.* The exact effect of this and the type of the return value depends on what type of control it is.

Checkbox

The function will return a number which will be 1 if the checkbox is checked, or zero if not.

Radiobutton

The function will return a number which will be 1 if the radiobutton is selected or zero if not.

Slider

The function will return a number giving the position of the slider.

SBKLink

The function will return the sbk that the link is pointing to, or an invalid value if none.

ValueIn

The function will return the number that the user has entered in the edit. If the number is out of range, then an invalid value will be returned (Esbeekay would have first given an error message to the user since the number was out of range).

NoteIn

The function will return the note number that the user has entered by keyboard or MIDI in the edit. If the number is out of range, then an invalid value will be returned (Esbeekay would have first given an error message to the user since the number was out of range).

TextIn

The function will return a string which will be the text the user has entered into the edit.

Listbox

If the listbox is a single-selection listbox, a number will be returned which will be -1 if there is no selection in the listbox, or a number giving the number of the selected item.
If the listbox is a multiple-selection listbox, a list of numbers will be returned which will be item numbers of all selected items in the listbox, or an empty list if none.

Combobox

The function will return a number which will be -1 if there is no selection in the combobox, or a number giving the number of the selected item.

void windowOutput(controlId::string, value::any)

This function will output *value* to the named macro window control *controlId.* The exact effect of this depends on what type of control it is.

Checkbox

The value has to be a number, and if the number is non-zero the checkbox is checked otherwise it will be unchecked.

Radiobutton

The value has to be a number, and if the number is non-zero the radiobutton is selected otherwise it will be unselected.

Slider

The value has to be a number, and is used to set the position of the slider.

ValueIn

The value has to be a number, and will be displayed in the edit.

NoteIn

The value has to be a number, and will be displayed in the edit.

TextIn

The value has to be a string, and will be displayed in the edit.

Text

The value has to be a string, and will be used to change the text being displayed in the control.

Listbox

The value has to be a list of strings. The listbox will display those strings.

Combobox

The value has to be a list of strings. The combobox will display those strings.

{number*} getDefaultISParams()

This function will return the default parameters for connections from instruments to samples. The parameters are represented by a list of numbers.

conIS createConIS(fromInst::inst, toSample::sample, params::{number*})

This function will create a connection from an instrument to a sample header. The connection will be from *fromInst* to *toSample* and will have the parameters given by *params.* You can get a set of default parameters by using **getDefaultISParams()**.

The function will return the connection which was just created.

conIS createStdConIS(fromInst::inst, toSample::sample)

This function will create a connection from an instrument *fromInst* to a sample header *toSample.*. The connection will have the default parameters for a connection of that type.

The function will return the connection which was just created.

{conis*} getSampleToConISs(obj::sample)

This function will return a list of connections from the given sample header *sample* to all instruments which use that sample header.

{conpi*} getInstToConPIs(obj::inst)

This function will return a list of connections from the given instrument *inst* to all patches which use that instrument.

inst getConISToInst(link::conis)

This function will return the instrument connected to the connection *link.*

patch getConPIToPatch(link::conpi)

This function will return the patch connected to the connection *link*.

any getObjectData(obj::[patch | inst | sample])

This function gets object data associated with an object *obj*. The type of data returned will depend on the object an is as follows:

Patches { name::string, patchNumber::number, patchBank::number}

If the patch is a percussion patch, the bank number will be 128.

Instruments name::string

Samples { name::string, isRom::number, start::number, end::number, loopStart::number, loopEnd::number}

If the sample is a ROM sample, isRom will be non-zero. If isRom is zero and start, end, loopStart and loopEnd are all zero as well then the sample header is an inactive one.

void setObjectData(obj::[patch | inst | sample], data::any)

This function sets object data associated with the object *obj.* The type of data required in the argument *data* depends on the type of *obj* as follows:

Patches	{ name::string, patchNumber::number, patchBank::number}

If the patch is a percussion patch, the bank number will be 128.

Instruments	name::string
Samples	{ name::string, isRom::number, start::number, end::number, loopStart::number, loopEnd::number}

If the sample is a ROM sample, isRom will be non-zero. If isRom is zero and start, end, loopStart and loopEnd are all zero as well then the sample header is an inactive one.

void deleteObject(obj::[patch | inst | sample | conpi | conis | sbk])

This function will delete the object *obj.* It should be stressed that one should be extremely careful deleting objects for the following reasons:

1. When an object is deleted, the object reference purports to be still valid, whereas it really isnt. So references to an object which has been deleted are not encouraged. For this reason, storing references to objects in global variables is not recommended, since the objects might disappear e.g. through user actions but the references will still be there to invalid objects.

2. When an object is deleted, all connected objects which depend on the existence of that object will also disappear, e.g. deleting a patch will also delete all connections from the patch.

3. When deleting an SBK, if the SBK refers to a file which is open as a window, the file will be closed without prompting for saving and the window will be gone.

void copyFromSBK(sbkto:sbk, sbkfrom::sbk, objs::{[patch | inst | sample | conpi | conis]*})

This function copies objects listed in *objs* from the SBK *sbkfrom* to the SBK *sbkto.* All sample data associated with the copied objects is also copied. If the objects conflict with existing objects, e.g. patch numbers conflict, then this will cause problems - Esbeekay will not test for conflicts in this function.

string numberToString(n::number, decimals::number)

This function will return a string which contains the number *n* with *decimals* decimal places.

number makeKeyRange(from::number, to::number)

This function will return a parameter value for the keyRange parameter where the range is from note *from* to note *to.*

number makePitch(freq::number)

This function will return a parameter value for the pitch parameter where the pitch is derived solely from a sample playback frequency *freq*.

number makePitchAndRoot(freq::number, root::number)

This function will return a parameter value for the pitch parameter where the pitch is derived from a sample playback frequency *freq* and a root key *root.*

{from::number, to::number} decodeKeyRange(par::number)

This function will take a parameter value *par*, assuming that *par* represents a keyRange parameter value, and return a list which contains the start and end notes of the range.

Macro language object parameter name set

These parameter names (in quotes - to make them strings) are used as parameter names for objects in the Esbeekay macro language. These represent the current known parameters.

Looping

localLoopStartOffset 16-bit word offset to beginning of loop
localLoopEndOffset 16-bit word offset to end of loop
loop Sound will loop if this is set to 1

Sound formation

Values to these parameters will be as in the AWE DIP. Probably.

LFO1Pitch Amount LFO1 changes pitch
LFO2Pitch Amount LFO2 changes pitch
EG1ToPitch Amount EG1 changes pitch
LPFFC LPF Fc
LPFFQ LPF Fq
LFO1ToLPF Amount LFO1 influences LPF
EG1ToLPF Amount EG1 influences LPF
LFO1ToVolume Amount LFO1 changes volume
LFO1Delay LFO1 delay
LFO1Freq LFO1 frequency
LFO2Delay LFO2 delay
LFO2Freq LFO2 frequency

Envelope control

There are two envelopes, EG1 and EG2. They are five-part envelopes with a sustain level. The parameter values are probably as in the AWE DIP.

EG1Delay
EG1Attack
EG1Hold
EG1Decay
EG1Sustain
EG1Release

EG2Delay
EG2Attack
EG2Hold
EG2Decay
EG2Sustain
EG2Release

Pitch control

rootKey Range 0 to 127.

pitch

The units for this parameter are cents. This parameter will define the basic pitch of the sound. If a root key is defined as well, then this parameter should be based on key 60 (i.e. it

should be 6000). Changes in sample playback frequency are made by modifying this parameter so that with 44kHz there is no modification, with 22kHz a value of 1200 is added to this parameter etc. Root keys can of course be emulated using this pitch parameter, and it is actually used for that purpose in a lot of the standard SBKs, but since the root key parameter became available it has not been used often for that purpose, only for playback frequency.

pitchOffset	Offset of sound in semitones
pitchAdjust	Pitch adjust for sound in cents, probable range -99 to 99.
pitchOffsetCent	Pitch offset in cents
quartertone	

Miscellaneous

keyRange	Note range to be played for a sample.
volume	Range 0 to 127.
chorus	Range 0 to 127.
reverb	Range 0 to 127.
pan	Range 0 to 127. 0 is left, 127 is right.

Quick Start Guide to using SBKs

There are two known methods of using SBKs for midi playback.

One is to configure the SBK as an additional user bank. To do this, start the AWE32 control panel and select the SBK in the file dialog. Double click on it and it will become user bank n where n is the user bank number shown on the left. To use the bank, insert CC0 messages in the midi file with the bank number (n) as the parameter. In Cakewalk, you can do this by calling up the event list for a track, pressing Insert, changing the event type to controller, and changing the first number in the event to zero and the second number to the bank number (n).

The second method is to store the SBK in your SoundFont directory (probably \sb16\sfbank) as synthusr.sbk. Then edit the file sbwin.ini in the Windows directory so that the line beginning with USER reads USER=C:\SB16\SFBANK\SYNTHUSR.SBK. (The directory and drive may need to be changed). Start Windows, and select the Custom User Synth in the synth combobox. The SBK will now be your primary SoundFont, and there is no need to insert CC0 events to access the sounds - they will played by default.

Quick Start Guide to making SBKs

Starting with a number of samples in either GUS, Kurzweil, or WAVE format you can produce a simple SBK file by following these steps. **Step #5 is important unless you configure the envelopes differently.**

(a) Starting with a WAVE file

1. In Esbeekay, open the WAVE file. Select the WAVE in the list box and then Edit/Copy. In the empty SBK file, select Edit/Paste. You will have a sample and instrument in the SBK.

2. Click in the patch list. Select Object/Create. Type in the patch name and select a patch number for it. Click Ok.

3. Double click on the instrument you created which will be in the instrument list.

4. Select the patch in the patch list. Select Object/Connect to view instrument.

5. Click on the connection from the instrument to the sample which will be in the graph view. Select Object/Edit. Select EG2 sustain and drag the slider to full sustain. Click Ok.

6. You now have a valid patch. You can save it, or test it by selecting the patch and Object/Play.

(b) Starting with a GUS file

This will mostly be the same as for WAVEs. Open the GUS file from Esbeekay. Select the main instrument (on the left). Select Edit/Copy. In the empty SBK file, select Edit/Paste. Make sure parameter 25 is set as in step #5 above. You now have a valid patch.

(c) Starting with a Kurzweil file

This is the same as for GUS files. To copy Kurzweil patches, select the program in the left-hand box and Edit/Copy. Follow GUS procedure for rest of steps.

Testing Patches

Important: You need to be running AWE32 drivers version 1.1 to use patch testing. These drivers are available in the AWE DIP. Esbeekay should complain if the driver is old. You can also check which drivers you have in the Esbeekay About... box. When upgrading drivers, ensure that they go in the correct directories!

You can test either the pure sound in a sample, or particular patches.

Pure sounds are tested from sample edit dialogs. Only RAM samples can be tested in this way. In the dialog, select a playback frequency and click on Play. This method of testing uses WAVE output to produce the sound. It will *not* use any parameters, envelopes etc.

To test patches and hear them as they would sound if played as midi patches, use one of two methods. The most straightforward is to double-click on the patch you want to listen to, click on the keyboard icon in the toolbar, and then click on the keys at the bottom of the graph view or use a MIDI keyboard. By selecting the play settings in the View menu you will be able to adjust the chorus, reverb and velocity of the graph keyboard. If you use a MIDI keyboard, the velocity will be taken from the MIDI keyboard.

The other way of testing patches is to select a patch(s) and use the command Object/Play. This will bring up a play test dialog for the patches. This will only work for patches. You can then play notes by giving a note range, individual notes using the keyboard and notes using a midi keyboard if one is connected. The simulated keyboard uses either the mouse or Space bar to play notes, with the arrow keys moving between notes. The octave numbers may be wrong but they are chosen to correspond with the octave numbers in the AWE32 manual GS variation tables.

The duration of notes played in the patch test dialog will vary. For a range of notes, each note will be a certain number of milliseconds long - this is adjustable in the dialog. When playing keyboard notes, the note will sound as long as the key or mouse button is held down. For midi keyboards, notes will sound as long as they are held down. Note velocity is determined in the dialog for keyboard and range notes, but for midi keyboards, velocity is as the midi keyboard sends it.

When testing patches, some delays caused writing data to AWE32 RAM will occur when the first note from an SBK is played. The AWE32 interface is not that fast. After you have played the first note, everything should proceed normally. You can have as many play testing boxes open at once as you want. The test sounds are stored in the AWE32 in addition to the normal synth bank, so if you run out of memory, set your normal synth bank to GM and that should minimize memory usage.

Notes

The sample test play rate is supposed to work for any sampling rate. However, the AWE32 Windows Wave driver doesnt allow any other playback rate except the standard multiples of 11kHz. This is sad, since some samples do not use exact multiples. I hope that the playback will work properly on systems which can play at non-standard rates, but I have not tested it.

The selection system in SBK graphs is not at all well done. Some amount of moving the mouse around might be required to select the correct object. To help with this rather haphazard technique, a description of the current object under the mouse is shown at the bottom of the main window. Alternatively, use the keyboard.

It is possible to control a few start-up colours, fonts, and settings which Esbeekay uses by modifying the esbeekay.ini file in the Windows directory. Colours are entered using RGB triples, e.g. 200,128,128 (commas must separate numbers). Fonts are entered using FontName,Size where FontName is the typeface name and Size is a number giving the point size, e.g. Helv,8. Preference settings are either Yes or No. Click here to see a list of possible keys.

The selection of items in SBK list boxes is not that easy by keyboard - i.e. if you switch to another box with TAB, then you have to first eliminate the current selection unless you want to specifically select items in both boxes. There seems to be no way to make it less awkward unfortunately.

In the default parameter dialogs it is not possible to TAB to all the controls. Use the mouse instead. I have given up trying to fix this - with custom controls and switchable contents, the scenario is just too complicated.

INI file settings

The following keys are possible (comments are in italics):

[Colours]

WindowBack	*Background colour for all main windows*
MapBack	*Background colour for SBK graph window*
MapBottom	*Background colour for bottom area of graph window*
MapSelected	*Colour of item selection border in graph*
MapCurrent	*Colour of current item border in graph*
MapCurrentAndSelected	*Colour of current and selected item border*
WaveBack	*Background colour of wave display window*
WaveLine	*Colour of waveform in wave display window*
WaveRangeBack	*Colour of bottom area in wave display window*
WaveRange	*Colour of wave range bar*
WaveLoop	*Colour of wave loop bar*
EnvelopeBack	*Background of envelope diagram*
EnvelopeLine	*Colour of envelope area in envelope diagram*
NoteBarOne	*Colour of indicator for one sample on note bar*
NoteBarMany	*Colour of indicator for many samples on note bar*
RootKey	*Colour of root key indicator in the graph view*
MacroBack	*Background colour for all macro windows*

[Fonts]

WindowFont	*Font for all main windows*
MapFontPatch	*Font used for patches in graph window*
MapFontInstrument	*Font used for instruments in graph window*
MapFontSample	*Font used for samples in graph window*

[Preferences]

SmartDraw	*Optimize drawing flicker: no flicker, but can be very slow*
AutoSampDel	*Delete unused areas of the sample chunk automatically*
ConfirmDel	*Confirm all deletes*
PrintGraph	*When printing, print graphs.*
PrintPatches	*When printing, print patch list*
PrintInstruments	*When printing, print instrument list*
PrintSamples	*When printing, print sample list*
SamplePadding	*How many bytes inserted as padding between samples*
AutoPadding	*When saving automatically insert padding between samples*
MidiInDevice	*Empty for any device, missing for SB16 midi port, or name*
HexEdit	*Old-style hex editing for parameters*
UniqueInstrumentNames	*Enforce restriction that instrument names be unique*
ShowRawParameters	*Show raw parameter values when adjusting parameter sliders*
PromptRootOnDrag	*Prompt for root key when dragging samples with root key set*

Some options here can be set in the Options menu, but most have to be set in the INI file directly.

In addition to INI file settings, there are command line options for Esbeekay as follows:

/nonew	*Stops the default empty SBK from being created*
filename	*Loads the file initially. You can have as many initial files as you like.*

Introduction to Esbeekay

Q: Why the name *Esbeekay*?

A: It is named after a small town in Austria, Esbeekay, where the some of the first pioneering work on wavetable storage systems was done.

Q: What does Esbeekay do?

A: It can be used to edit and create (often without any idea of what is being edited) SoundFont Bank files used by the AWE32. You can import Kurzweil, MOD and GUS patches into .sbk files. You can paste windows waves or wave files into an .sbk file. You can use Esbeekay to set each and every parameter in a new or an existing .sbk file. You can merge .sbk data from other .sbk's, like the ones that came with the AWE32. You can use ROM patches. You can create that monster 8MB patch file.

Q: What's the catch?

A: Unfortunately, the .sbk format is not available, i.e. it's secret. This means that many parameters in .sbk files are cryptic and their effect is unknown. Some parameters have been discovered, but others remain a mystery. So you can control a fair amount of how sounds are played, but some things are still unknown. You can however experiment with unknown parameters and if you discover what they do, they will be added to the program in an easy-to-use format. Additionally, the structure of SBK files is changing.

Q: What kind of system do I need?

A: You need a 386, Windows, and lots of memory. The program is hacked together. Therefore memory use is at times enthusiastic. You need at least as much RAM as the sizes of the samples you're manipulating. Maybe it can be swappable: I don't know. Each time you copy something into the clipboard, the samples are actually copied. So beware.

Q: Do I have to pay to use Esbeekay?

A: No. It is shareware on a strictly voluntary basis. If you like the program, and if you feel you can afford it, then by all means do send along a contribution. A suggested level would be $20. If however you do not pay anything, then that is fine too. Click here for the address for contributions.

Q: How do I find out more about the .sbk file format?

A: If you find out, please tell me too.

Q: Why is this program such a skimpy hacked-up job?

A: Because it was written by only one person in their spare time.

Q: How do I let the author know about bugs/problems?

A: Either e-mail to apl@vapl.demon.co.uk or use the direct feedback facility in the About... box.

Please send contributions to:

Ari Laakkonen
Flat 4
56 Culverden Rd
London SW12 9LS
UK

Any UK cheques, eurocheques, or international money orders will be fine.

Editing SoundFont Banks

The main window for SBKs is divided into a graph view and three list views. The graph view is used to display an object such as a patch, instrument or sample with all its connected objects. The lists display all the patches, instruments, and samples in that file. You can have any number of SBKs open at once, as well as other formats, like Kurzweil files (see Importing).

In the graph view, use the arrow keys to move the pointer in the connected hierarchy. Enter switches the current object to be the main object and shows all its connections. Space selects an item, SHIFT+Space toggles the selection state of an item. Actions in the Object and Edit menus operate on selected items.

The list views operate as list boxes. SHIFT-click extends selection, CTRL-click toggles selection, (use SHIFT+arrows and SHIFT+F8 for the keyboard). Unlike the graph view, when operating with the lists, selected items are all those selected in *all three* lists, *and* all the connections between the selected items *and* the parameter blocks connected to selected items. As in the graph view, the commands in the Object and Edit menus operate on selected objects but here some objects which will be selected (e.g. connections) will not be visible. Pressing Enter while in a list switches the graph view to display that object.

To switch between the graph view and the lists, use TAB.

As well as the main view, there are several dialog boxes for editing individual objects. These are started by selecting one or more objects and using Object/Edit... on the selection. There can be any number of edit dialogs open at once.

Graph editing

It is possible to drag samples, or their end points to note positions. This can be used to change the key range of a sample within the context of a particular instrument or patch. The instrument or patch must have a key range parameter defined for it. If a sample has a root key set for it in one of the parameter blocks leading up to the sample, you may also change the root key by dragging it (this will modify the pitch as well to keep the sample rate constant).

If a sample block is located so that it is very narrow and you are unable to select the whole block for movement, pressing SHIFT while over the sample will cause the cursor to change to a block movement cursor.

A shortcut is to click with the right mouse button on an object to edit that object. This works for both the lists at the bottom and the graph.

The keyboard below the graph shows what keys samples are mapped to. It will also show the occupied sample ranges in green, and those ranges where more than one sample are mapped (this is not an error - many sbks have overlaid samples) in red. It will show samples being dragged, and you can select all samples which map onto a particular key by clicking on the key.

Parameter editing

Parameters can be edited for connections. These are patch-instrument connections, instrument-sample connections, and also pure parameter blocks for patches and instruments.

There are two ways of editing parameters. The default parameter editing dialog has controls for all currently known parameters. It is also possible to edit the parameters by numbers, this is available by selecting Options/Hex Edit.

In the normal parameter dialog, sliders are used with the keyboard or the mouse. The keyboard arrows keys move the slider up & down, SHIFT+arrows move the slider up and down by a small amount for fine adjusting, and Space toggles the parameter on/off. Sliders can be operated using the mouse by dragging the slider knob with the left button depressed, or with the right button depressed for fine adjusting, and the parameter can be turned on/off by SHIFT+click. To enter a parameter value directly, left double click (or press Backspace) on the slider. All fields in the parameter dialog which require key/note values can also be operated using a MIDI keyboard - click in the field and press a key on the MIDI keyboard to enter a value.

Parameters can also be edited for a group of selected objects (the objects must contain parameters) using the Edit Group... command. This will only show parameters which are shared by all the objects which you had selected. It is ideal for editing logical groupings e.g. connections leading upto stereo samples which have mostly the same data.

Sample editing

It is possible to drag the small marker triangles in the wave window for RAM samples to change the start, end, loop start, and loop end positions. When dragging positions, a helper window will open which displays the current position being dragged to and the value of the sample at that point. Loop start and end phases can then be matched to avoid clicks.

There is a slider for zooming in on the zero line to make it easier to match loop points which are very near the zero line.

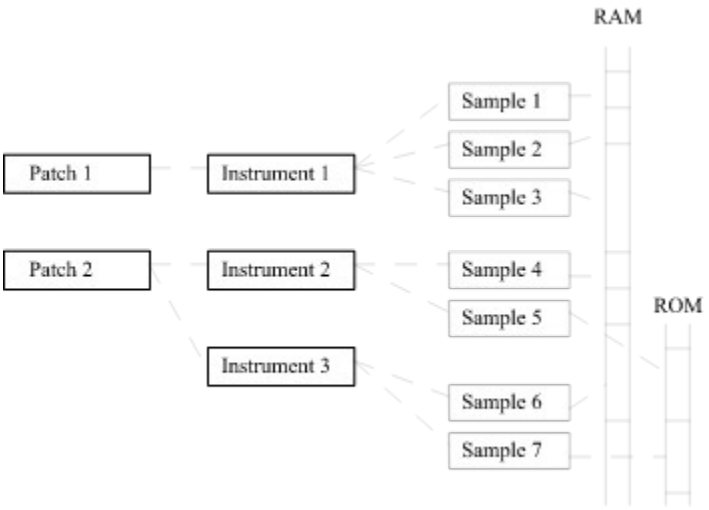
Connections

To create new connections, select one of the objects you want to connect by double-clicking so that it becomes the main object in the graph view. Then select another object in one of the list boxes, and select the Object/Connect to command. This will create a link between the two objects, and the second object will appear in the view. New connections will have the parameters set in the Default Parameters dialog.

Structure of an SBK file

An SBK contains information which allows the AWE32 synthesizer program to generate midi notes. All information presented below has been obtained by examining the file format of SBKs and experimenting with changes to them, so beware of inaccuracies.

An SBK is divided into four main layers: the sample space, sample headers (called samples), instruments, and patches. The sample space contains all of the data for RAM samples in one big chunk. Sample headers describe either RAM samples by referring to an area in the sample space, or ROM samples by giving the location in ROM of sample data. Instruments are a way of grouping sample headers together and giving a set of parameters to each unique instrument/sample connection. Instruments are often used for key maps. Patches are the top-level objects, and they group instruments together, with each unique patch/instrument connection capable of having a set of parameters. Patches have a MIDI patch and bank number attached to them, and they are played when a note is played. Patches can share instruments, and instruments can share samples. Samples can share areas of RAM - no duplication is needed at any stage.



Patches

Patches give the midi patch number and bank of a particular sound. The sound will be composed of a group of one or more instruments, and will also be characterized by the parameters connecting the patch to the instruments, see below.

Instruments

Instruments are used to group samples. This is their only function - the real functionality is provided by parameters, as discussed below.

Sample headers

Sample headers give four basic references to sample data: the start and end, and the loop start and end. From the data stored in sample headers it is not possible to deduce whether the header references ROM or RAM - so Esbeekay will deduce this from the ordering of the samples. Sample headers do **not** give any information about the root key, sample rate etc. of a sample. These are given by parameters (see below).

Sample data

Sample data is always signed 16-bit PCM with no restrictions on sampling rate. Esbeekay will automatically convert all imported data into this format. Sample data is kept in one huge chunk, and more than one sample header can use one particular area of the sample data. ROM data is also 16-bit, but it is not possible to examine ROM data - or at least it is not known how this could be done. ROM data shares the address space with RAM data in SBKs, i.e. it starts from zero. However ROM samples beginning at zero are not allowed. You can use ROM samples by giving the location in ROM of the sample data - these can be found e.g. from the sample SBKs which come with the AWE32.

From examination of standard SBKs there seems to be a small amount of padding between RAM samples. This padding is automatically inserted by Esbeekay. If it is found that samples work without padding, the padding feature can be disabled. The padding is normally 150 bytes between samples, but this can be changed in the esbeekay.ini file. The standard files seem to use 75-100 bytes.

Parameters

Connecting patches to instruments and instruments to sample headers are parameter sets. Also associated with each patch or instrument there can be a parameter set. There is no difference between patch and instrument parameters. There are probably about 50 different parameters, and with Esbeekay it is possible to edit/insert/delete each of these. The real mystery is in the meaning of the parameters. Probably most parameters are known, but there are gaps. The entire sound is controlled with parameters - i.e. volume, pitch, balance, envelope, LFOs etc. Those parameters whose function is known have an easier-to-use editing window in Esbeekay but other parameters will have to be edited as numbers.

Part of the hierarchical organization of SBKs into patches and instruments shows how parameters should be used. Those parameters relating to a patch sound globally should be attached to the patch, or to patch-instrument connections, e.g. volume Those parameters relating to specific key ranges should be attached to instrument-sample connections, e.g. key range or pitch.

Known parameters

The following is a list of the parameters known so far:

02: Sample loop start offset - references 16-bit samples.
03: Sample loop end offset - references 16-bit samples.
04: Pitch offset in cents
05: LFO1 to pitch
06: LFO2 to pitch
07: Env1 to pitch
08: Filter cutoff
09: Filter resonance
0a: LFO1 to filter
0b: Env1 to filter
0d: LFO1 to volume
0f: Chorus
10: Reverb
11: Pan
15: LFO1 delay
16: LFO1 frequency
17: LFO2 delay

18: LFO2 frequency
19: Envelope 1 delay time
1a: Envelope 1 attack time
1b: Envelope 1 hold time
1c: Envelope 1 decay time
1d: Envelope 1 sustain level
1e: Envelope 1 release time
21: Envelope 2 delay time
22: Envelope 2 attack time
23: Envelope 2 hold time
24: Envelope 2 decay time
25: Envelope 2 sustain time
26: Envelope 2 release time
29: Used as a pointer from patches to instruments
2B: Key range
30: Volume
33: Pitch offset
34: Pitch fine adjust
35: Used as a pointer from instruments to samples
36: Loop
37: Pitch
38: Quartertone scales
3a: Root key

This list is starting to finally seem quite complete, but there may be others.

Pitch is a parameter which combines sampling rate, root key, and an adjustment all in one parameter. Because of the way the parameter is formed from three components, it is not possible to extract each component again. You should therefore keep a record of the components, e.g. the sample rate. You will then be able to set the sample rate and adjust the root key. The root key can be entered as a parameter but is missing from most early SBKs.

Parameter Notes

Where do I put the parameters?

There are four places parameters could possibly go: with a patch, with an instrument, and in the connections from patch to instrument and instrument to sample. It would seem that the parameters can be treated as commands to the synth. The synth would execute or play a patch by starting at a patch and following all the instrument and sample links all the way down to samples. At each stage it would execute the parameters which are at that stage. Therefore, parameters which are stored in the patch would apply for all samples, and parameters stored in instrument-sample links are executed only for that sample.

The logical way to divide the parameters based on this would be patch-wide parameters in the patch parameter block, instrument-wide parameters in the instrument parameter block, and sample-specific parameters in the instrument-sample connections. By examining some of the standard synth banks, you will see this sort of parameter distribution.

The Amplitude Envelope

This is referred to in the AWE DIP as envelope #2. It controls the volume of the sample being played while it is being played. The delay time, attack time, hold time, decay time, sustain and release times function as expected. It would seem that it is necessary to have the sustain level at quite a high level since volume seems to fall off rapidly below about 90 (range 0 to 127). According to the documentation that would be 27.75dB less than maximum, if each unit is 0.75dB.

Pitch

Pitch is a parameter which combines sampling rate, root key, and an adjustment all in one parameter. Because of the way the parameter is formed from three components, it is not possible to extract each component again. However, if the root key parameter is used, the awkward pitch parameter can be ignored most of the time and just used for the sample rate.

The units of the pitch parameter are 1200 per octave. Therefore, to adjust sounds to play an octave lower or higher you would change the pitch by 1200.

The pitch parameter is compact but it is extremely inconvenient from the SBK-builders point of view. There is a separate root key parameter, which can be set but in early SBKs it is not used. If you use it (recommended), then the pitch slider can be used to alter the playback frequency directly.

Root key

Older SBKs will not contain root key parameters. Esbeekay will use the root key parameter if one exists. To create the root key parameter for a sample if it already does not have one, use the parameter dialog to enter one. Thereafter you may change the root key for a sample by dragging it in the graph view. When you drag the root key in the graph, the effective pitch will change although the pitch parameters will not be modified.

Looping

If this parameter is set, the sound will loop if the note is kept on for a sufficiently long period. The sample needs to have a loop range defined. The note will loop after the basic note range has been played. If you have no attack, hold, decay, or delay times in the note envelope the sustain part will be entered very fast - therefore, if the sustain value is zero you will not hear

the looping. Set parameter 25 (EG2 sustain) to full sustain to set the sustain level at the maximum for the patch.

Importing data

Data can be imported into SBK files via the clipboard, i.e. by using Paste in the Edit menu. However, only a few things can be pasted. These are data copied or cut into the clipboard from another or the same SBK, data copied from a GUS, Kurzweil, or Wave window, or simple Windows Wave data. These effectively mean that you can mix and match SBK contents along with importing data from GUS, Kurzweil and Wave sources. A varying amount of information is carried along with imported data so when importing you will need to adjust the default parameters which are automatically generated for imported data.

Data can be pasted either using the **Paste** command, or **Paste and Connect** command. Paste and Connect tries to connect the new items to the main display item in the current view - e.g. if you are displaying a patch, and you paste an instrument then a connection will automatically be made between the two.

When pasting from a non-SBK source, complete information (required for SBKs) will not accompany the data. Therefore it is recommended that a spare blank SBK be used as an intermediary editing stage, and the data can be further copied from there when ready to be transferred into the real SBK.

Import conflicts

Importing from this or another SBK

Importing from a GUS patch file

Importing from a Kurzweil patch file

Importing Windows Waves

Importing samples from a MOD file

Importing from a MOD file

MOD import will import all samples in a .mod file as patch->instrument->sample mappings. Information about volume and a guess of the sample rate is carried over but all other parameters are set to defaults.

Import conflicts

When importing data which has conflicts with current data, the user has the option to cancel the import or to delete current conflicting data and continue with the import. It is suggested however that all data to be imported be manually inspected in an SBK window to avoid mistakenly deleting important conflicting data. A common cause of conflicts is the patch number in a patch.

Importing from this or another SBK

All objects including connections are preserved as they are. There are likely to be conflicts with existing data unless care is taken.

Importing from a GUS patch file

GUS patches are composed of instruments, layers, and samples. Instruments are mapped to patches, layers are mapped to instruments, and samples are mapped to samples. Sample data is automatically converted to the correct format. Default connection information is placed in connections between instruments and samples so that root keys, key ranges, balance, and sample rates are transferred correctly. The import process also tries to estimate the volume envelope present in GUS samples, this may not always be transferred correctly since the GUS envelope is different from the AWE envelope. Other parameters are copied for connections from the default parameters for new connections.

If you select an object in a GUS file and copy it to the clipboard, all objects used to define that object are also copied.

Importing from a Kurzweil patch file

The Kurzweil format features programs, maps, layers, and samples. Programs are mapped to patches, maps to instruments, layers again to the same instruments, and samples to samples. Format conversion may not correctly occur since not enough is known about the Kurzweil format, and since the samping rate in particular is not known, the resulting pitch assigned to samples (which is based on the root key being heard at 44.1kHz) will probably be slightly wrong. Other parameters are copied for connections from the default parameters for new connections. It is easy to correct by twiddling the pitch settings in the SBK itself when the data has been imported, assuming you can hear or measure the pitch being played. Kurzweil samples provide a good source of high-quality samples.

If you select an object in a Kurzweil file and copy it to the clipboard, all objects used to define that object are also copied.

You can get a lot of Kurzweil samples from ftp.uwp.edu in /pub/music/lists/kurzweil/sounds.

Many thanks to FMJ (f93-maj@nada.kth.se) for help on the Kurzweil file format.

Importing Windows Waves

Windows Waves can be either opened as a file from **File Open...** and copied from there into the clipboard, or pasted direct from the clipboard if another application has put the Wave into the clipboard. Esbeekay cannot handle compressed Waves, but can convert from 8-bit to 16-bit Waves automatically. Unsigned/Signed conflicts will also be automatically resolved.

A Wave, when pasted into an SBK, becomes a RAM sample with an instrument connected to it. The connection between the instrument and the sample will contain parameters relating to pitch which should make the Wave sound ok, although the root key will be a guess, so that will have to be adjusted. When pasting from a Wave opened as a file, the sample and instrument names will be derived from the file name. When pasting direct from the clipboard, unique names will be generated by the system for the sample and the instrument.

Stereo Waves can also be pasted. In this case, one instrument will be created which is connected to two samples. Each connection from the instrument to the sample for the left and right channel will contain the appropriate pan settings. The Group Edit command is particularly useful for editing stereo sample parameters.