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

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

Under Developement

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 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 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 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.

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.

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

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

addSBKData(<SBKRef>,<nBytes>) -> nOffset

Description:

This function will add an empty area of sample space to the data block of an SBK. The added area is initialised to zero.

Arguments:

<SBKRef> is an SBK reference. It is to this referenced SBK that the empty area is added. A reference to an SBK file is obtained with the function windowInput()

<nBytes> is the length of the added area in bytes.

Returns:

nOffset is the offset of the beginning of the new area from the beginning of the original sample block.

beep()

Description:

This function sounds the speaker.

Arguments:

None.

Returns:

None.

```
callDLL(<cDllName>,
        <cFunctionName>,
        <lParameters>) -> nReturnValue
```

Description:

This function will call the function **<cFunctionName>** in the 16-bit DLL specified by **<cDllName>**, passing as a parameter to the function a string which is the textual representation of all values in **<lParameters>**, separated by spaces. The parameter **<cDllName>** should not specify a directory name, just the filename of the DLL. DLL's that you want to reference must be located in the same directory as the macro which calls that DLL. The function returns **nReturnValue** ,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 elements in the parameter **<lParameters>** 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.

When running under a 32-bit operating system (Windows NT), you cannot call 16-bit DLLs.

Arguments:

<cDllName> is the name of the DLL in which the function you want to call resides.

<cFunctionName> is the name of the function you want to call inside the DLL. <cDllName>.

<lParameters> is a list of parameters which will be passed to the function **<cFunctionName>** inside the DLL. **<cDllName>**.

Returns:

nReturnValue is a numeric value which will be returned from the dll function.

Example:

```
TestFunction()
{
var param1;

param1:= {"john",5};

callDLL("my.dll","MyFunction",param1);
}
```

```
callDLL32(<cDllName>,
          <cFunctionName>,
          <lParameters>) -> nReturnValue
```

Description:

This function will call the function **<cFunctionName>** in the 32-bit DLL specified by **<cDllName>**, passing as a parameter to the function a string which is the textual representation of all values in **<lParameters>**, separated by spaces. The parameter **<cDllName>** should not specify a directory name, just the filename of the DLL. DLL's that you want to reference must be located in the same directory as the macro which calls that DLL. The function returns **nReturnValue** ,the number returned by the DLL function.

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

extern "C" long __declspec(dllexport) fname(char *args)

(this is for MS Visual C++ v2.0)

The elements in the parameter **<lParameters>** 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.

Arguments:

<cDllName> is the name of the DLL in which the function you want to call resides.

<cFunctionName> is the name of the function you want to call inside the DLL. **<cDllName>**.

<lParameters> is a list of parameters which will be passed to the function **<cFunctionName>** inside the DLL. **<cDllName>**.

Returns:

nReturnValue is a numeric value which will be returned from the dll function.

Example:

```
TestFunction()
{
var param1;

param1:= {"john",5};

callDLL("my.dll","MyFunction",param1);
}
```

callMacro(<cMacroName>,<cFunctionName>,<?Parameter>) -> ?AnyValue

Description:

This function allows you to call another function **<cFunctionName>** contained within another macro **<cMacroName>**.
If **<cMacroName>** is not prefixed with a valid path specification, it is expected that **<cMacroName>** should reside in the same directory as the macro which initiates this function call.

Where there is a window associated with the macro in which **<cFunctionName>** resides, the window will NOT be opened when **<cFunctionName>** is called by means of callMacro().

See also openMacroWindow()

A return value from the called function **<cFunctionName>** is compulsory. The return value may however be of any data type.

Probably the greatest advantage of this function in it's current form, is the fact that it allows one to create function library's. Functions that you find are contained in many of your macro's may now be contained in a separate macro. Once these functions are working correctly, they may simply be called from other macro's, thus reducing macro developement time significantly.

Arguments:

<cMacroName> is the name of the macro in which **<cFunctionName>** resides. It is a string type value. **<cMacroName>** may have a prefixed path name if you desire, in which case, the macro in which the called function **<cFunctionName>** resides, may be located anywhere you like. If **<cMacroName>** is not prefixed with a path specification, it is expected that **<cMacroName>** should reside in the same directory as the macro initiating this function call.

<cFunctionName> is the name of the function contained within the macro **<cMacroName>** that you wish to call. This is a string type parameter. Do not include parentheses as part of the function name. This is the correct form for the function beep() "beep"

<?Parameters> is the parameter or list of parameters you want to pass to the function **<cFunctionName>**. This parameter may be of any data type. Should you wish to pass more than one parameter to **<cFunctionName>** you may do so by means of a List data type.
eg:
{"apples",5}

Returns:

?ReturnValue is the value which is returned from the function **<cFunctionName>**. While this return value may be of any data type, it is compulsory that a return value be returned from **<cFunctionName>**.

Example:

In mymac1.mac

```
ok:= callMacro("C:\esbk\global.mac","beeplots",5);
```

In global.mac

```
beeplots(x)
{
var counter;
counter:= 0;

while (counter <= x)
```

```
 {
   beep();
   counter:= counter +1;
 }

return 0;
}
```

closeMacroWindow(<cMacroFileName>)

Description:

This function closes the open macro window of any macro that you specify. The macro that you specify must however be a floating macro and must have been installed.
openMacroWindow() may be used to open a macro window.

Arguments:

<cMacroFileName) is the name of the macro window that you wish to close. It is a string type parameter.
<cMacroFileName> should not contain the pathname of the macro. The macro filename extension is optional. It is expected, therefore that <cMacroFileName> should reside in the same directory as the macro in which this function, closeMacroWindow(), is contained.

Returns:

None.

```
copyFromSBK( <SBKRefTarget>,
             <SBKRefSource>,
             <lpatchobject |
               linstobject |
               lsampleobject |
               lconpiobject |
               lconisobject> )
```

Description:

This function copies a list of objects from the SBK , **<SBKRefSource>** to the SBK **<SBKRefTarget>**. 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.

Arguments:

<SBKRefTarget> is an SBKRef designating the SBK from which objects are to be copied.

<SBKRefSource> is an SBKRef designating the SBK to which objects are to copied. SBK reference objects can be obtained with windowInput()

The object parameter is a list of objects to copied. The object parameter will be one of the following depending upon the type of object you want to copy.

<lpatchobject>	is a Patch Object.
<linstobject>	is an Instrument Object.
<lsampleobject>	is a Sample Object.
<lconpiobject>	is a Patch Instrument Connection Object.
<lconisobject>	is an Instrument Sample Connection Object.

Returns:

None.

createConISO(<instobject>,
<sampleobject>,
<lParameters>) -> conisobject

Description:

This function will create a connection from an instrument object **<instobject>** to a sample header object **<sampleobject>**.
It will have the parameters specified by **<lParameters>**.

Arguments:

<instobject> is an instrument object. The connection will be created from this instrument.
<sampleobject> is the sample header object to which the connection from the instrument object will lead.
Seeing as an instrument to sample connection is an object, conisobject ,it has parameters. **<lParameters>** is a list of parameters which will be taken on by the new connection that is created.
To get a list of default parameters for this type of object, you may use the getDefaultISParams() function.

Returns:

conisobject is an instrument to sample connection object.

createStdConIS(<instobject>,<sampleobject>) -> conisobject

Description:

This function will create a connection from an instrument object **<instobject>** to a sample header object **<sampleobject>** The connection will have the default parameters for a connection of this type.

Default parameters for this type of connection can be set from the Esbeekay menu. <OPTIONS><(instrument-sample defaults).

The function will return the connection which was just created.

Arguments:

<instobject> is an instrument object. The connection will be created from this instrument.

<sampleobject> is a sample header object. It is to this object that the connection will be created.

Returns:

conisobject is an instrument to sample connection object. This is the object created by this function.

createConPI(<patchobject>,
<instobject>,
<lParameters>) -> conpiobject

Description:

This function will create a connection from an patch object **<patchobject>** to a instrument object **<instobject>**. It will have the parameters specified by **<lParameters>**.

Arguments:

<patchobject> is an patch object. The connection will be created from this patch.
<instobject> is the instrument object to which the connection from the patch object will lead.
Seeing as an patch to instrument connection is an object, conisobject ,it has parameters. **<lParameters>** is a list of parameters which will be taken on by the new connection that is created.
To get a list of default parameters for this type of object, you may use the getDefaultPIParams() function.

Returns:

conpiobject is an patch to instrument connection object.

createInst(<SBKRef>,<cInstrumentName>) -> instobject

Description:

This function creates a new instrument in the SBK file pointed to by **<SBKRef>**. The new instrument will have the name, **<cInstrumentName)**.

Arguments:

<SBKRef> points to the SBK File in which the new instrument will be created.
It is an SBKRef object type.

<cInstrumentName> is a string type parameter. It is the name which will be given to the new instrument.

Returns:

instobject is the object which is created by this function. It is an instrument object type.

```
createPatch(<SBKRef>,
            <cPatchName>,
            <nPatchNumber>,
            <nBankNumber>) -> patchobject
```

Description:

This function will create a patch in the SBK file pointed to by **<SBKRef>**. The new patch will have the name **<cPatchName>**. It's patch number will be **<nPatchNumber>** and it's bank number will be **<nBankNumber>**.

If **<nBankNumber>** = 128 then the patch will be a percussion bank. This function does not check for the existence of a patch having an identical patch number and bank number within the same SBK.

Arguments:

<SBKRef> is an SBKRef object. It points to the SBK in which you want the new patch to be created.

<cPatchName> is a string parameter and is the name that the new patch will, have when it is created.

<nPatchNumber> is a numeric parameter and is the number the patch will have when created.

<nBankNumber> is a numeric parameter and is the bank number the new patch will have when created.

Returns:

patchobject is a patch object. It is the patch object that is created by the function.

createSample(<SBKRef>,
<cSampleName>,
<nIsRamRom>,
<nStartPos>,
<nEndPos>,
<nLoopStart>,
<nLoopEnd>) -> sampleobject

Description:

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 **<cSampleName>**. The sample will be located from **<nStartPos>** (which must be an even number) to **<nEndPos>** (which must be an odd number). The loop in the sample is located from **<nLoopStart)** (which must be an even number) to **<nLoopEnd)** (which must be an odd number). If **<nIsRamRom>** 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 SBK's: sample dialogs in Esbeekay will give the addresses of the ROM samples which are used in those SBK's. 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 - don't try it, because I won't take responsibility for anything that happens!

If **<nIsRamRom>** is non-zero and **<nStartPos>**, **<nEndPos>**, **<nLoopStart>** and **<nLoopEnd>** 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 returns the sample header object which was created.

Arguments:

<SBKRef> is an SBKRef type. It specifies in which SBK the new sample object will be created.

<cSampleName> is a string type parameter. This is the name the new sample object will have when created.

<nIsRamRom> is a numeric parameter. If this parameter = 0, then the sample locations are in ram. Otherwise, if the parameter is not = 0 then the sample locations are in AWE32 Rom.

<nStartPos> is a numeric parameter which must be an even number. The sample will be located from this position.

<nEndPos> is a numeric parameter which must be an odd number. The sample will end at this position.

<nLoopStart> is a numeric parameter which must be an even number. The loop in the sample will start at this position.

<nLoopEnd> is a numeric parameter which must be an odd number. The loop kin the sample will end at this position.

Returns:

sampleobject is a sample object. It is the object created by this function.

createStdConPI(<patchobject>,<instobject>) -> conpiobject

Description:

This function will create a connection from an patch object **<patchobject>** to an instrument object **<instobject>** The connection will have the default parameters for a connection of this type.

Default parameters for this type of connection can be set from the Esbeekay menu. <OPTIONS><(patch-instrument-defaults).

The function will return the connection which was just created.

Arguments:

<patchobject> is a patch object. The connection will be created from this patch.

<instobject> is an instrument object. It is to this object that the connection will be created.

Returns:

conpiobject is a patch to instrument connection object. This is the object created by this function.

decodeKeyRange(<nParameterValue>) -> lRangeValues

Description:

This function returns the upper and lower range values for the keyrange parameter **<nParameterValue>**.

The keyrange value is stored in an SBK as a single value.
Esbeekay works with this parameter as two separate values which is far more convenient from a user standpoint. It is therefore necessary to be able to convert between the two formats.

Arguments:

<nParameterValue> is a numeric value. This is the actual parameter value that would be stored in the SBK file.

Returns:

lRangeValues is a list which has two elements. The first element is the lower range value for the keyrange parameter, while the second element is the lower range value for the keyrange parameter.

The following functions are closely related to this function:

makeKeyRange()

getParam()

setParam()

decodeVelocityRange(<nParameterValue>) -> lRangeValues

Description:

This function returns the upper and lower range values for the velocity range parameter **<nParameterValue>**.

The velocity range value is stored in an SBK as a single value.
Esbeekay works with this parameter as two separate values which is far more convenient from a user standpoint. It is therefore necessary to be able to convert between the two formats.

Arguments:

<nParameterValue> is a numeric value. This is the actual parameter value that would be stored in the SBK file.

Returns:

lRangeValues is a list which has two elements. The first element is the lower range value for the velocity range parameter, while the second element is the lower range value for the velocity range parameter.

The following functions are closely related to this function:

makeVelocityRange()

getParam()

setParam()

deleteObject(<patchobject |
instobject |
sampleobject |
conpiobject |
conisobject |
SBKRef>)

Description:

This function will delete an object. 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 isn't. 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.

Arguments:

Objects that may be deleted by this function may be any one of the following.

<patchobject> is a patch object
<instobject> is an instrument object
<sampleobject> is a sample object
<conpiobject> is a connection from patch to instrument object
<conisobject> is a connection from instrument to sample object
<SBKRef> is an SBKRef object

Returns:

None.

delSBKData(<SBKRef>,<nFrom>,<nLength>)

Description:

Deletes a range starting with **<nFrom>** which is **<nLength>** bytes long of the data block from **<SBKRef>**. Any sections of that range which are used by any RAM samples in **<SBKRef>** will NOT be deleted.

Arguments:

<SBKRef> is an SBKRef type. It represents the SBK from which you want to delete a data block.

<nFrom> is a numeric value which represents the starting position from which to delete.

<nLength> is the number of bytes to delete string from the position <nFrom>

Returns:

None.

displayHelp(nContextIdentifier)

Description:

This function displays a help file.
The help file must have the same name as the macro that invokes the function. The help file must have the .hlp extension.
The help file must reside in the same directory as the macro which invokes this function.

Arguments:

<nContextIdentifier> is a numeric value. If this value is 0, the help file will be opened at it's opening topic page. If **<nContextIdentifier>** is a valid context identifier within the help file, the help file will be opened at it's relevant topic page.

Returns:

None.

filenameFromPath(cPath>-> cFileName

Description:

Extracts the filename **cFileName** from a path specification **<cPath>**

Arguments:

<cPath> is a string type parameter. It is a complete path specification including a filename.
eg: "c:\wavfiles\snare.wav".

Returns:

cFileName is the extracted filename. It is returned as a string type.

Example:

```
var pathspec, filename;
pathspec:= "c:\wavfiles\snare.wav";
filename:= filenameFromPath(pathspec);
message(filename);      % Displays snare.wav
```

```
filePrompt(<lcFileMasks>,
           <lcMaskDescriptions>,
           <nAllowMultipleSelection>,
           <nIsSaveFileDialog>,
           <cInitialDirectory>)-> lFilenames
```

Description:

Displays a file prompt dialog. The dialog may be either for saving files or loading files. The initial directory can be specified as can the allowable file masks complete with their corresponding descriptions.
Both **<lFileMasks>** and **<lMaskDescriptions>** must have the same number of elements. If the length of both these lists is 0, then the default "All Files" mask will be used.

Arguments:

<lFileMasks> is a list of strings. Each string being a file mask.

<lMaskDescriptions> is a list of strings. Each string being the corresponding description for the masks specified in **<lFileMasks>**

<nAllowMultiple> is a numeric parameter. If this parameter <> 0 (True), then multiple file selections are enabled within the dialog.

<nOpen) is a numeric parameter. If this parameter is <> 0 (True), the dialog will be an Open Dialog.

<cDirectory> is a string parameter specifying the default directory for the dialog. If the parameter = "", then the default directory will be the current directory.

Returns:

lFilenames is a list of strings. Each string being a filename selected in the dialog. If no files are selected in the dialog, this function returns an empty string "".
Empty strings can be tested for with length()

Example:

```
FileOpen()

{
var mask,descrip,open,multiple,open,dir,selected,eachfile;

mask:= "*.SBK","*.WAV"              % File Masks
descrip:= Sbk files","Wav Files";           % Mask Descriptions
open:= 1;                                   % Specify File Open Dialog
multiple:= 1;                               % Allow Multiple Selections
dir:= "";                                   % Current Directory Is Default

selected:= filePrompt(mask,descrip,open,multiple,dir);       % Show Dialog

if (Length(selected) <> 0)          % Some Files Were Selected

   {
    for eachfile in selected                % For Each Selected File
        message(eachfile);                          % Display File Name
   }
   else                                     % No Files Selected
   {
    beep();                                 % Beep Speaker
    message("No Files Selected");           % Display 'No Files Selected'
```

```
    }
  }
}
```

getConISToInst(<conisobject>)->instobject

Description:

This function returns the instrument object **instobject** connected to the instrument sample connection **<conisobject>**

Arguments:

<conisobject> is an instrument sample connection object.

Returns:

instobject is the instrument object to which the instrument sample connection **<conisobject>** is connected.

Where **<conisobject>** has no connected instruments, **instobject** will be an invalid object and can be tested as such with the objectIsValid() function.

getConPIToPatch(<conpiobject>)->patchobject

Description:

This function returns the patch object **patchobject** connected to the patch instrument connection **<conpiobject>**

Arguments:

<conpiobject> is a patch instrument connection object.

Returns:

patchobject is the patch object to which the patch instrument connection **<conpiobject>** is connected.

Where **<conpiobject>** has no connected patches, **patchobject** will be an invalid object and can be tested as such with objectIsValid()

getInstToConPIs(<instobject>)->lconpiobjects

Description:

This function returns a list of all connections from a given instrument object **<instobject>** to it's connected patches.

Arguments:

<instobject> is an instrument object.

Returns:

lconpiobjects is a list of conpi objects. Where **<instobject>** has no connections to any patches, the returned list **lconpiobjects**, will be empty. Use the length() function to test for empty lists.

```
getAllParams(<patchobject |
             instobject |
             conpiobject |
             conisobject>) -> IParameterValues
```

Description:

This function will return a list of all parameter values for the object supplied as an argument.

Arguments:

<patchobject> or ..
<instobject> or ..
<conpiobject> or ..
<conisobject>

This argument should be an object being one of the following object types.

Patch Object
Instrument Object
ConPi Object]
ConIs Object

Returns:

IParameterValues is a list of values for the parameters for the object specified.

```
getAttachedObject() -> patchobject |
                       conpiobject |
                       instobject  |
                       conisobject |
                       sampleobject
```

Description:

This function will work only in macro windows which are attached to dialogs or SBK windows. The function will return the object which is being edited in the attached window.
For SBK windows this will be an <SBKRef> for the SBK. For sample dialogs it will be the sample object, etc.

Arguments:

None.

Returns:

The function will return one of the following objects, depending on the dialog type that the macro is attached to ..

Returned Object	Macro Window Type
patchobject	Patch Edit
conpiobject	P-I Edit
instobject	Instrument Edit
conisobject	I-S Edit
sampleobject	Sample Edit
SBKRef	SBK Window

getConISToSample(<conisobject>)->sampleobject

Description:

This function returns the sample object **sampleobject** connected to the instrument sample connection **<conisobject>**

Arguments:

<conisobject> is an instrument sample connection object.

Returns:

sampleobject is the sample object to which the instrument sample connection **<conisobject>** is connected.

Where **<conisobject>** has no connected samples, **sampleobject** will be an invalid object and can be tested as such with the objectIsValid() function.

getConPITOInst(<conpiobject>)->instobject

Description:

This function returns the instrument object **instobject** connected to the patch instrument connection **<conpiobject>**

Arguments:

<conpiobject> is a patch instrument connection object.

Returns:

instobject is the instrument object to which the patch instrument connection **<conpiobject>** is connected.

Where **<conpiobject>** has no connected instruments, **instobject** will be an invalid object and can be tested as such with the objectIsValid() function.

getDefaultPIParams() -> IParameterValues

Description:

This function will return a list of numeric values corresponding to the default parameter values for patch instrument connections.

Default parameters for this type of connection can be set from the Esbeekay menu. <OPTIONS><(patch-instrument-defaults).

Arguments:

None.

Returns:

IParameterValues is a list of numeric values which are the values corresponding to the default parameter values for a patch instrument connection.

getDefaultISParams() -> IParameterValues

Description:
This function will return a list of numeric values corresponding to the default parameter values for instrument sample connections.

Default parameters for this type of connection can be set from the Esbeekay menu. <OPTIONS><(instrument-sample defaults).

Arguments:

None.

Returns:

IParameterValues is a list of numeric values which are the values corresponding to the default parameter values for an instrument sample connection.

getInstToConISs(<instobject>)->lconisobjects

Description:

This function returns a list of all connections from a given instrument object **<instobject>** to it's connected samples.

Arguments:

<instobject> is an instrument object.

Returns:

lconisobjects is a list of conis objects. Where **<instobject>** has no connections to any samples, the returned list, **lconisobjects**, will be empty. Use the length() function to test for empty lists.

```
getObjectData(<patchobject |
              instobject |
              sampleobject>) -> lProperties
```

Note !! This function has been superceded by the function getObjectProperties(). In order to maintain future compatibility, it is recommended that you no longer use getObjectData() and that you change your existing macro's to use getObjectProperties() at your earliest convenience. While getObjectData() is supported in this version, it may be removed from future versions.

Description:

This functions returns the properties of a specified object. The returned properties are returned as list elements.

Arguments:

The argument to this function may be one of the following ..

<patchobject>	is a patch object.
<instobject>	is an instrument object.
<sampleobject>	is a sample object.

Returns:

lProperties is a list of properties for the specified object.

getObjectSBK(<patchobject |
instobject |
sampleobject>)-> SBKRef

Description:

This function returns the SBK that a particular object is attached to.

Arguments:

This function can take any one of the following three arguments.

<patchobject> A patch object
<instobject> An instrument object
<sampleobject>A sample object

Returns:

<SBKRef> is an SBKRef Object and represents the SBK to which the specified object belongs.

```
getParam(<SBKRef>,
         <patchobject |
          instobject |
          conpiobject |
          conisobject>,
         <cParameterName |
          nParameterNumber>)-> nParameter
```

Description:

This function returns the value of the parameter as indicated by **<cParameterName>** within the SBK referred to by **<SBKRef>**.

Arguments:

<SBKRef> is an SBKRef Object referring to an SBK file and may be obtained by using the function windowInput().

The object for which you require the specified parameter **<cParameterName>**
may be any one of the following..

<patchobject> is a Patch Object
<instobject> is an Instrument Object
<conpiobject> is a Patch Instrument Connection Object
<conisobject> is an Instrument Sample Connect Object

<cParameterName> is the name of the parameter to query and is a string type.
List Of Valid Parameter Names

Alternatively you may use **<nParameterNumber>** and specify the parameter number you wish to query instead.
List Of Valid Parameter Numbers

Returns:

This function returns the parameter value for the **<ParameterName>** for the given object within **<SBKRef>**

If **<cParameterName>** does not exist for the given object, then an invalid value will be returned. You should, therefore, always test the result of this function with the objectIsValid() function unless you know for certain that the parameter will exist.

getPatchToConPIs(<patchobject>)->lconpiobjects

Description:

This function returns a list of all connections from a patch object **<patchobject>** to it's connected instruments.

Arguments:

<patchobject> is a patch object.

Returns:

lconpiobjects is a list of conpi objects. Where **<patchobject>** has no connections to any instruments, the returned list **<lconpiobjects>**, will be empty. You may use the length() function to test for empty lists.

```
getObjectProperties(<patchobject |
                    instobject |
                    sampleobject>) -> lProperties
```

Note !! this function supercedes getObjectData().

Description:

This functions returns the properties of a specified object. The returned properties are returned as list elements.

Arguments:

The argument to this function may be one of the following ..

<patchobject>	is a patch object.
<instobject>	is an instrument object.
<sampleobject>	is a sample object.

Returns:

lProperties is a list of properties for the specified object.

Example:

```
showObjectProperties()
{
var mylist, myobject;

myobject:= getAttachedObject();
mylist:= getObjectProperties(myobject);

message(mylist[0]);
message("Patch Number Is " + numberToString(mylist[1],0));
message("Bank Number Is " + numberToString(mylist[2],0));
}
```

getSampleToConISs(<sampleobject>)->Iconisobjects

Description:

This function returns a list of all connections from a given sample header object **<sampleobject>** to it's connected instruments.

Arguments:

<sampleobject> is a sample header object.

Returns:

Iconisobjects is a list of conis objects. Where **<sampleobject>** has no connections to any instruments, the returned list **Iconisobjects**, will be empty. You may use the length() function to test for empty lists.

getSBKDataLen(<SBKRef>) -> nBlockLength

Description:

Returns the length of the data block in SBK **<SBKRef>**, the size is measured in bytes.

Arguments:

<SBKRef> is an SBKRef pointer to the SBK you wish to query.

Returns:

nBlockLength is a numeric value and represents the length of the data block in the SBK pointed to by **<SBKRef>** in bytes.

getSBKInstruments(<SBKRef>) -> lInstrumentObjects

Description:

This function returns a list of instrument objects contained in the SBK pointed to by **<SBKRef>**.

Arguments:

<SBKRef> is an SBKRef type. The SBK to which it points is the SBK from which the list of instrument objects will be extracted.

Returns:

lInstrumentObjects is a list of instrument objects contained in the SBK to which **<SBKRef>** points.

Where there are no instruments contained in the SBK **<SBKRef>**, an empty list {} will be returned by this function. Empty lists may be tested for by using the length() function.

getSBKLink(<controlid>)-> SBKRef

NOTE!!

This function has been superceded by the function windowInput(). In order to maintain future compatibility, it is recommended that you no longer use getSBKLink() and that you change your existing macro's to use windowInput() at your earliest convenience. While getSBKLink() is supported in this version, it may be removed from future versions.

Description:

Returns <SBKRef>, an SBKRef Object that the SBKLink is pointing to, if any.

Arguments:

<controlid> is an SBKLink control id.

Returns:

SBKRef an SBK Object.

The value returned by this function should always be checked using objectIsValid(). The return value will not be valid if the SBKLink is not pointing to some SBK.

getSBKPatches(<SBKRef>) -> lPatchObjects

Description:

This function returns a list of Patch Objects contained in the SBK pointed to by **<SBKRef>**.

Arguments:

<SBKRef> is an SBKRef Object. The SBK to which it points is the SBK from which the list of Patch Objects will be extracted.

Returns:

lPatchObjects is a list of Patch Objects contained in the SBK to which **<SBKRef>** points.

Where there are no patches contained in the SBK **<SBKRef>**, an empty list {} will be returned by this function. Empty lists may be tested for by using the length() function.

getSBKSamples(<SBKRef>) -> lSampleObjects

Description:

This function returns a list of Sample Objects contained in the SBK pointed to by **<SBKRef>**.

Arguments:

<SBKRef> is an SBKRef Object. The SBK to which it points is the SBK from which the list of Sample Objects will be extracted.

Returns:

lSampleObjects is a list of Sample Objects contained in the SBK to which **<SBKRef>** points.

Where there are no samples contained in the SBK **<SBKRef>**, an empty list {} will be returned by this function. Empty lists may be tested for by using the length() function.

getWindowControlType(<cControlId>) -> nControlType

Description:

This function determines the type of a macro window control as specified by the control id **<controlid>**

It supports all of the Macro Window Controls.

Arguments:

<cControlId> is a string type parameter and represents a valid macro window control id.

It may be any one of the following control types.

Button
Checkbox
Radiobutton
Slider
SBKLink
Text
ValueIn
NoteIn
TextIn
Listbox
Combobox

Returns:

nControlType is a numeric value and represents a macro window control as per the following table. Note also that constants have been provided for this function so that you may use the name instead of the number in your macro code.

Control Type	Returned Value	Constant Name
Button	0	controlTypeButton
Checkbox	1	controlTypeCheckbox
Radiobutton	2	controlTypeRadiobutton
Slider	3	controlTypeSlider
SBKLink	4	controlTypeSBKLink
Text	5	controlTypeText
ValueIn	6	controlTypeValueIn
NoteIn	7	controlTypeNoteIn
TextIn	8	controlTypeTextIn
Listbox	9	controlTypeListbox
Combobox	10	controlTypeCombobox

Note that if a control having the control id **<controlid>** is not contained in the macro window, a run time error will occur.

Example:

```
showType()
{
var idlist, id, controlnames, controltype;

idlist:= {"Button1","Checkbox1","Radiobutton1"};

for each id in idlist do
```

```
{
controltype:= getWindowControlType(id);
message(id + " is a " idlist[controltype]);
}
}
```

getWindowControlRange(<controlid>) -> InRangeValues

Description:

This function returns the upper and lower range of a control **<controlid>** as specified in the macro design window by the *From:* and *To:* fields.

The following controls are supported..

Sliders

NoteIn

ValueIn

Arguments:

<controlid> is the control id of the control for which you want to determine the upper and lower range.

Returns:

InRangeValues is a list containing two elements. The first of which will be the *From:* value as specified in the macro design window and the second element will be the *To:* value as specified in the macro design window. The element values will be numeric.

Where no range values are assigned to the control **<controlid>**, the two elements will both have the value 0.

```
importGUS(<SBKRef>,
          <cGusFileName>,
          <nPatchStart>,
          <nBankNumber>) -> lPatchObjects
```

Description:

This function will import a .pat file specified by **<cGusFileName>** into the SBK specified by **<SBKRef>**. The patches created on the import will be in bank **<nBankNumber>**, and will be numbered from **<nPatchStart>** onwards. If **<nPatchStart>** 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 **<nPatchStart>**.

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.

Arguments:

<SBKRef> is an SBKRef object which represents the SBK file into which the GUS file will be imported.

<cFileName> is a string type parameter which represents the name of the GUS file to import.

<nPatchStart> is a numeric parameter. Imported will be numbered from this value onward.

<nBankNumber> is a numeric parameter. This is the bank number to which the imported patches will be assigned, If this parameter = 128, then the patches will be percussion patches.

Returns:

lPatchObjects is a list of the patches that have been imported.

```
importKRZ(<SBKRef>,
          <cFileName>,
          <nPatchStart>,
          <nPatchBank>) -> lPatchObjects
```

Description:

This function will import a .krz file specified by **<CFileName>** into the SBK specified by **<SBKRef>**.

The patches created on the import will be in bank **<nPatchBank>**, and will be numbered from **<nPatchStart>** onwards. If **<nPatchBank>** 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 **<NPatchStart>**.

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.

Arguments:

<SBKRef> is an SBKRef object. It represents the SBK file into which the Kurzweil patches will be imported.

<cFileName> is a string value. It represents the name of the Kurzweil file to import.

<nPatchStart> is a numeric value. Imported patches will be numbered from this value onward.

<nPatchBank> is a numeric value. Imported patches will be assigned this bank number.

Returns:

lPatchObjects is a list of patch objects. If an error occurs this list will contain one invalid patch object. Validity of objects can be tested with the function objectIsValid().

importWave(<SBKRefDestination>,
<cWavFileName>,
<cInstrumentName>,
<nFilter>)-> instobject

Description:

This function imports a .wav file **<cFileWaveName>** into **<SBKRefDestination>**. Both the instrument created and the imported samples will have the name **<cInstrumentName>**.
When importing a stereo .wav file, the left and right channels will be assigned to the same instrument object but will be panned full left and full right respectively. In the case of a stereo .wav file, the sample names as specified by **<cInstrumentName>**, will be suffixed with an L and an R respectively.

Arguments:

<SBKRefDestination> is the SBK file into which the .wav file will be imported.
This is an SBKRef object type. An SBKRef can be obtained for an SBK file with the windowInput() function.

<cWavFileName> is the name of the .wav file to import. This a string data type.

<cInstrumentName> is the name that will given to the resulting instrument object and to the imported samples. This a string data type.

<nFilter> is used to ignore stereo/mono .wav files. This parameter can also be used to ignore left/right channels within a stereo .wav file. This is a numeric data type. These are the filter values for **<nFilter>**. By adding their values together one may specify exactly what kind of .wav data to import or ignore.

Filter Table:

0: Import any .wav file.
1: Do not import mono .wav files.
2: Do not import the left channel of a stereo .wav file.
4: Do not import the right channel of a stereo .wav file.

Instead of using these numeric values, one may also use these corresponding constants if you prefer.

importWaveAny
importWaveNoMono
importWaveNoLeft
importWaveNoRight

Returns:

instobject

This function returns an instrument object which is in turn connected to the samples contained within the .wav file imported by this function.
Where a .wav file is not imported, the instrument object returned will be invalid. To test the validity of this instrument object use the objectIsValid() function.

Example:

To ignore mono .wav files and to ignore the Right channel of stereo .wav files we need to calculate the value of **<nFilter>**. To do this we add the values as specified in the filter table. 1(no mono) + 4(no right channel) = 5.
<nFilter> in this case would therefore = 5.
The result would be that we would only import the left channel data of stereo .wav files.

importWave(mysbk,"c:.wav","newinst",5);

init()

Description:

This function is one of the callback functions provided by the Esbeekay Macro Language. If this function is declared within a macro, it will be called automatically when:

: The Macro is installed.
: When the Esbeekay program is started, provided the macro(s) in which this function is declared, is/are installed.

Arguments:

None.

Returns:

None.

itemsInList(<cControlId>) -> nNumberOfItems

Description:

This function returns the number of items contained in either a Listbox or ComboBox.
It is identical to the function itemsInWindowControl()

Arguments:

<cControlId> is a string type parameter. It represents a valid controlid for a Listbox or a ComboBox.

Returns:

nNumberOfItems is a numeric value and represents the number of items contained in either a Listbox or ComboBox represented by **<cControlId>**.

Example:

```
countItemsInMyList()
{

var mylist,count;

mylist:= {"Apples","Pears","Oranges"};
windowOutput("Listbox1",mylist);
count:= itemsInList("Listbox1");
message(numberToString(count,0));

}
```

itemsInWindowControl(<cControlId>) -> nNumberOfItems

Description:

This function returns the number of items contained in either a Listbox or ComboBox.
It is identical to the function itemsInList().

Arguments:

<cControlId> is a string type parameter. It represents a valid controlid for a Listbox or a ComboBox.

Returns:

nNumberOfItems is a numeric value and represents the number of items contained in either a Listbox or ComboBox represented by **<cControlId>**.

Example:

```
countItemsInMyList()
{

var mylist,count;

mylist:= {"Apples","Pears","Oranges"};
windowOutput("Listbox1",mylist);
count:= itemsInWindowControl("Listbox1");
message(numberToString(count,0));

}
```

length(<cString | lList>)->nLength

I**NOTE !!** This function supercedes the function listLength() In order to maintain future compatibility, it should be used in preference to listLength().

Description:

This function returns the length of a string or the number of elements in a list.

Arguments:

<cString> is a string parameter.
<lList> is a list

Returns:

nLength is either the length of a string **<cString>** or the number of elements in a list **<lList>**

Macro Language Library Functions In Alphabetical Order

Function parameter and return syntax rules

makePitchAndRoot
makeVelocityRange
message
notifyInstChange(sbkObject)
notifyPatchChange(sbkObject)
notifySampleChange(sbkObject)
notifySBKDataChanged
notifyWindowClose
notifyWindowOpen
numberToString
objectIsValid
openMacroWindow
removeAllParams
removeParam
setAllParams
setGroupCallback
setObjectData
setObjectProperties
setParam
setSBKLinkManual
subString
valid
windowInput
windowInputValidate
windowOutput

Macro Language Library Functions

Function parameter and return syntax rules

I/O Functions

- beep
- callDLL
- callDLL32
- callMacro
- displayHelp
- filePrompt
- importGUS
- importKrz
- importWave
- message

Object traversal functions

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

Object creation functions

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

Object manipulation

- copyFromSBK
- deleteObject
- getObjectData
- getObjectProperties
- setObjectData
- setObjectProperties

Parameter manipulation

- getAllParams
- removeAllParams
- setAllParams

- getParam

removeParam
setParam

getDefaultISParams
getDefaultPIParams

decodeKeyRange
makeKeyRange

decodeVelocityRange
makeVelocityRange

makePitch
makePitchAndRoot

Window control functions

closeMacroWindow
getSBKLink
getWindowControlRange
getWindowInputError
getWindowControlType
itemsInWindowControl
openMacroWindow
setGroupCallback
setSBKLinkManual
windowInput
windowInputValidate
windowOutput

Sample space functions

addSBKData
delSBKData
getSBKDataLen
notifySBKDataChanged

General functions

filenameFromPath
itemsInList
length
listLength
objectIsValid
numberToString
subString
valid

listLength(<lList>)->nListLength

NOTE !! This function has been superceded by the function length(). In order to maintain future compatibility, it is recommended that you no longer use **listLength()** and that you change your existing macro's to use length() at your earliest convenience. While **listLength()** is supported in this version, it may be removed from future versions.

Description:

This function return the number of elements contained within a list.

Arguments:

<lList> is the list for which you want to establish the number of elements.

Returns:

nListLength is the number of elements contained in **<lList>**
Where a list is empty, this function returns 0.

makeKeyRange(<nFromValue>,<nToValue>) -> nParameterValue

Description:

This function returns a parameter value corresponding to a key range.

The Parameter Value for keyrange is stored as a single value in an SBK file.
The Esbeekay program works with key ranges as Upper and Lower range values which is far more convenient for the user.

This function is therefore required to convert these range values to a single number parameter value which can then be used to set the appropriate parameter.

Arguments:

<nFromValue> This is a numeric value and represents the lowest note to which an object will respond.

<nToValue> This is a numeric value and represents the highest note to which an object will respond.

Returns:

nParameterValue is a numeric value which is created from **<nFromValue>** and **<nToValue>**. This is the value you would use to set the *keyRange* parameter if you wanted to adjust the keyrange of an object.

The following functions are closely related to this function.

decodeKeyRange()

setParam()

getParam()

makePitchAndRoot(<nFrequency>,<nRootKey>) -> nParameterValue

Description:
This function returns a parameter value **nParameterValue** for the parameter 0x37 *pitch* derived from a desired playback frequency **<nFrequency>** (in hertz) and a required Root Key **<nRootkey>**.

This function is something of a holdover from early SBK files. In earlier files, the Root Key and Playback Frequency, were encoded as one unified value which was stored in parameter number 0x37 *pitch*.

Later SBK's make use of two parameters to perform this same function which is far easier to work with.

Parameter 0x3a *rootKey* holds the Root Key value while parameter 0x37 *pitch* holds the Playback Frequency.

Users are currently advised to specify the Root Key by setting parameter 0x3a *rootKey* and to specify Playback Frequency by setting parameter 0x37 *pitch*.

Arguments:

<nFrequency> is a numeric value. It represents the frequency at which the sample will play back at midi note number **<nRootKey>**. It would under usual circumstances equal the frequency at which the sample was recorded. eg. 44100. This establishes a tonal centre for sample reproduction.

<nRootKey> is a numeric value. It is a midi note number. It is at this note that the sample will reproduce at frequency **<nFrequency>**.

Returns:

nParameterValue is a numeric value. This is the value that should be written to parameter 0x37 if you wish to support the unified parameter system.

makePitch(<nFrequency>) -> nParameterValue

Description:

This function will return a parameter value for the pitch parameter (0x37) based on a required playback frequency **<nFrequency>**.

In earlier SBK files parameter number 0x37 *pitch* combined sampling rate, root key and a pitch adjustment. You would then need to compute a parameter value which is a combination of the Root Key and playback frequency. See also makePitchAndRoot().

Since these early days, the new pitch parameter *rootKey* 0x3A has been discovered. This system works by setting parameter 0x37 *pitch* to the playback frequency and parameter 0x3a *rootKey* to the root key value. While SBK's using the old system are still supported, users are best advised to do the following ...

Leave parameter 0x37 alone, unless you want to change the playback rate of the sample.

Use parameter 0x3A to set the Root Key value.

Arguments:

<nFrequency> is the frequency value for which you want to return a value for parameter 0x37 *pitch*.
<nFrequency> should be specified in hertz eg. 44100 and not 44,1

Returns:

nParameterValue is the value calculated from the required playback frequency **<nFrequency>**.

makeVelocityRange(<nFromValue>,<nToValue>) -> nParameterValue

Description:

This function returns a parameter value corresponding to the velocity range.

The Parameter Value for velocity range is stored as a single value in an SBK file.
The Esbeekay program works with velocity ranges as Upper and Lower range values which is far more convenient for the user.

This function is therefore required to convert these range values to a single number parameter value which can then be used to set the appropriate parameter.

Arguments:

<nFromValue> This is a numeric value and represents the lowest velocity value to which an object will respond.

<nToValue> This is a numeric value and represents the highest velocity value to which an object will respond.

Returns:

nParameterValue is a numeric value which is created from **<nFromValue>** and **<nToValue>**. This is the value you would use to set the *velocityRange* parameter if you wanted to adjust the velocity range of an object.

The following functions are closely related to this function.

decodeVelocityRange()

setParam()

getParam()

message(<cTextToDisplay>)

Description:

This function displays a string of text in a dialog box. The dialog box has an OK button.

Arguments:

<cTextToDisplay> is a text string which will be displayed in the dialog box.

Returns:

None.

notifyInstChange(<SBKRef>)

Description:

This function is one of the callback functions provided by the Esbeekay Macro Language. If this function is declared within a macro, it will be called automatically when:

: An SBK File is opened or closed.
: When there is a change to an instrument object within the SBK file referenced by **<SBKRef>**.

Arguments:

<SBKRef> is an SBKRef object. An SBKRef to an SBK file is obtained with the function windowInput()

Returns:

None.

notifyPatchChange(<SBKRef>)

Description:

This function is one of the callback functions provided by the Esbeekay Macro Language. If this function is declared within a macro, it will be called automatically when:

: An SBK File is opened or closed.
: When there is a change to a patch object within the SBK file referenced by **<SBKRef>**.

Arguments:

<SBKRef> is an SBKRef object. An SBKRef reference to an SBK file is obtained with the function windowInput()

Returns:

None.

notifySampleChange(<SBKRef>)

Description:

This function is one of the callback functions provided by the Esbeekay Macro Language. If this function is declared within a macro, it will be called automatically when:

: An SBK File is opened or closed only if the SBK file contains samples.
: When there is a change to a sample object within the SBK file referenced by **<SBKRef>**.

Arguments:

<SBKRef> is an SBKRef object. An SBKRef to an SBK file is obtained with the function windowInput()

Returns:

None.

notifySBKDataChanged(SBKRef)

Description:

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

Arguments:

<SBKRef> is an SBKRef Object referring to the SBK file whose sample space block you want to update.

Returns:

None.

notifyWindowClose()

See Also: notifyWindowOpen()

Description:

This function is one of the callback functions provided by the Esbeekay Macro Language. If this function is declared within a macro, it will be called automatically when:

The macro window for the macro in which this function is contained, closes.

Manually closing a macro window with closeMacroWindow() will also generate a call to this function.

Arguments:

None.

Returns:

None.

notifyWindowOpen()

See Also: notifyWindowClose()

Description:

This function is one of the callback functions provided by the Esbeekay Macro Language. If this function is declared within a macro, it will be called automatically when the macro window for the macro in which this function is contained, opens.

This function if applicable, will be called immediately after the init() function. When a macro is being created with an initially open window, the sequence is as follows:

1) Window Opened.
2) init() is called.
3) This function, notifyWindowOpen() is called.

Manually opening a macro window with openMacroWindow() will also generate a call to this function.

In this manner, init() can properly initialize window controls.

Arguments:

None.

Returns:

None.

numberToString(<nNumberToConvert>,<nDecimalPlaces>)->cReturnValue

Description:

This function will return a value **<cReturnValue>** which is the string representation of the number **<nNumberToConvert>** with **<nDecimalPlaces>** decimal places.
When you want to convert a number to a string, this function will allow you to accomplish the necessary type conversion.

Arguments:

<nNumberToConvert> is the number for which you want a string result.
<nDecimalPlaces> Is the number of decimal places you require in the resultant string.

Returns:

cReturnValue is a string value being the representation of **<nNumberToConvert>**.

Example:

```
var number, string;

number:= 5;

string:= numberToString(number,0);
message(string);
```

```
objectIsValid(<patchobject |
              conpiobject |
              instobject |
              conisobject |
              sampleobject>) -> nValue
```

Description:

This function determines the validity of an object.

Arguments:

The single argument may be any one of the following objects.

<patchobject>
<conpiobject>
<instobject>
<conisobject>
<sampleobject>

Returns:

nValue is a numeric value. If the object is valid, nValue will be 1 (True) and 0 (False) if the object is invalid.

openMacroWindow(<cMacroFileName>)

Description:

This function opens the macro window of any macro that you specify. The macro that you specify must however be a floating macro and must have been installed.
closeMacroWindow() may be used to close an open macro window.

Arguments:

<cMacroFileName) is the name of the macro window that you wish to open. It is a string type parameter.
<cMacroFileName> should not contain the pathname of the macro. The macro filename extension is optional. It is therefore expected that the macro whose window you wish to open by using this function, should reside in the same directory as the macro that invokes this function.

Returns:

None.

removeAllParams(<patchobject | instobject>])

Description:

This function turns off the 'Enable Global Parameters' Checkbox in the parameter dialog for the specified object.

Arguments:

This single argument to this function may be any one of the two following objects.

<patchobject> Is a patch object
<instobject> Is an instrument object

Returns:

None.

```
removeParam(<patchobject |
            instobject |
            conpiobject |
            conisobject>,
              <cParameterName |
            nParameterNumber>)
```

Description:

This function will remove the parameter **<cParameterName>** from a specified object. **<cParameterName>** may be the parameter name as a string or it may be the parameter number **<nParameterNumber>** in either Hex or Decimal format. Eliminating a parameter, gives the default behavior for the specified object for that parameter.

Arguments:

This function supports any one of the following objects.

<patchobject> Is a Patch Object
<instobject> Is an Instrument Object
<conpiobject> Is a Connection from Patch To Instrument Object
<conisobject> Is a Connection from Instrument To Sample Object

<cParameterName> may be any one the parameter names contained in the parameter list. If you prefer, you may use the parameter number **<nParameterNumber>** instead of the parameter name as a string. Either the Hex or Decimal value may be used. Numeric Values for the parameters are given in the parameter list.

Returns:

None.

Symbols & Conventions

So that we can specify items in a general way, we use symbols and metasymbols. For example, when a keyword or metasymbol is enclosed in square brackets, it is optional, or where keywords or metasymbols are separated by | one you may use one or the other. Here is a complete list of symbols used in syntax.

Symbols	**Description**
<>	Indicates user input item
()	Indicates function argument list
[]	Indicates optional item or expression grouping
{}	Indicates a list
->	Indicates a function return value
...	Indicates repeating elements
,	Indicates list item or variable separator
||	Indicates a block statement
|	Indicates one of a choice.
;	Indicates end of statement line.

MetaSymbols

Metasymbols are used to describe the nature of syntax elements. They are constructed with a data type designator as a prefix followed by a logical descriptor. Data type designators are shown in the table below.

Data type designators are always shown in lower case and the logical descriptors are always in upper case. A logical descriptor describes the meaning of the argument that the metasymbol represents.

For example, cName represents a string type while cFileName represents a file whose name is of the string type.

Where an argument supports more than one data type, a prefix symbol is used for each data type or alternatively, the | symbol denotes a choice.

For Example:

someFunction((cnlArgument))

So from this we know that the argument to this function may be any one of three data types. namely: String ,Numeric or List.

Or..

someFunction(<cName | nNumber>) -> nResult

From this we know that the argument to this function may be either cName or nNumber and that this function will return a value of numeric type.

Where an argument may be one of any type, the descriptor is prefixed by ?

Datatype Prefix	Description
l	List
c	String Expression
n	Numeric Expression
?	Any Type
SBKRef	A reference to an SBK File.
patchobject	Patch Object
instobject	Instrument Object

sampleobject	Sample Header Object
conpiobject	Connection From Patch to Instrument Object
conisobject	Connection From Instrument To Sample Object
controlid	A control id for a Macro Window control

```
setAllParams(<patchobject |
             instobject |
             conpiobject |
             conisobject>,
             <lParameterList>)
```

Description:

This function set all of the parameters for a specified object.

Arguments:

The specified object may be any one of the following..

<patchobject> is a Patch Object
<instobject> is an Instrument Object
<conpiobject> is a Connection from Patch To Instrument Object
<conisobject> is a Connection from Instrument To Sample Object

<lParameterList> is a complete parameter list. ie, a list of values for EVERY possible parameter.

Returns:

None.

setGroupCallback(<controlid>,<cFunctionName>)

Description:

This functions groups radio button controls on a macro window. It does this by specifying the function that a radio button should call when clicked. This process overrides a radio buttons usual behavior in this respect which is to invoke a function having the same name as the control id of the radio button.

When the specified function **<cFunctionName)** is invoked by clicking a specified radio button, the invoked function **<cFunctionName>** will take one argument. This argument will be the control id of the radio button that caused the function **<cFunctionName>** to be invoked. The called function therefore has the ability to identify the radio button that caused it to be invoked.

To group radio buttons together, you must state this function once for each radio button that you want assigned to the group. By specifying the same function name **<cFunctionName>** in each instance, the radio buttons will call the same function when clicked. The radio buttons that you have specified are thereby grouped together.

This allows you not only to mimic the usual operation of radio buttons in a group. (ie. Only one button in a group may be turned on at a time.) but also to exercise accurate control over a group of radio buttons.

Arguments:

<controlid> is the control id of a radio button that you want to call function **<cFunctionName>** when clicked.

<cFunctionName> is a string type. It is a function name and represents the function that will be called when a radio button having the control id **<controlid>** is clicked.

Returns:

None.

Example:

```
MyFunc()
{
setGroupCallback("RadioButton1","ButtonHandler");
setGroupCallback("RadioButton2","ButtonHandler");
setGroupCallback("RadioButton3","ButtonHandler");
}

ButtonHandler(theButton)
{
windowOutput("RadioButton1", "RadioButton1" = theButton);
windowOutput("RadioButton2", "RadioButton2" = theButton);
windowOutput("RadioButton3", "RadioButton3" = theButton);
}
```

```
setObjectData(<patchobject |
              instobject |
              sampleobject>,
             <lProperties>)
```

Note !! this function has been superceded by setObjectProperties(). This function should not be used for new macro's. While the functionality of this function is identical to setObjectProperties(), we would urge you to change your current macro's in order to maintain future compatibility.

Description:

This function sets the properties of a specified object.

Arguments:

The object argument to this function may be one of the following ..

<patchobject> is a patch object.
<instobject> is an instrument object.
<sampleobject> is a sample object.

<lProperties> is a list of properties. You must supply a complete property list. You may not supply one property for an object, even if you want to change just one property. See example.

Returns:

None.

Example:

```
ChangePatchName()
{
var newProperties, oldProperties, thisobject;

thisobject:= getAttachedObject();
oldProperties:= getObjectProperties(thisobject);
newProperties:= {"Snare Drum"};
newProperties:= newProperties + oldProperties[1] + oldProperties[2]
setObjectProperties(thisobject,newProperties);
}
```

```
setParam(<patchobject |
          instobject |
          conpiobject |
          conisobject>,
            <nParameterNumber |
          cParameterName>,
        <nParameterValue>)
```

Description:

This function sets the parameter indicated by **<nParameterNumber>** or **<cParameterName>** to **<nParameterValue>** for any one of the allowable objects.

Arguments:

These are the objects in an SBK that have parameters. Any one of these objects may be specified for this function.

<patchobject> is a patch object
<instobject>is an instrument object
<conpiobject> is a connection from patch to instrument object
<conisobject> is a connection from instrument to sample object

The parameter whose value you want to change may be specified by either it's number **<nParameterNumber>** or it's name **<cParameterName>**. If you prefer to use numbers to specify a required parameter, you may use either it's hex value or it's decimal equivalent.

<nParameterValue> is a numeric value and is the value to wish to assign to the parameter you specify.

Returns:

None.

```
setObjectProperties(<patchobject |
                    instobject |
                    sampleobject>,
                    <lProperties>)
```

Note !! this function supercedes setObjectData().

Description:

This function sets the properties of a specified object.

Arguments:

The object argument to this function may be one of the following ..

<patchobject> is a patch object.
<instobject> is an instrument object.
<sampleobject> is a sample object.

<lProperties> is a list of properties. You must supply a complete property list. You may not supply one property for an object, even if you want to change just one property. See example.

Returns:

None.

Example:

```
ChangePatchName()
{
var newProperties, oldProperties, thisobject;

thisobject:= getAttachedObject();
oldProperties:= getObjectProperties(thisobject);
newProperties:= {"Snare Drum"};
newProperties:= newProperties + oldProperties[1] + oldProperties[2]
setObjectProperties(thisobject,newProperties);
}
```

setSBKLinkManual(<controlid>)

Description:

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.

Arguments:

<controlid> is a control id for an SBKLink control in a macro window.

Returns:

None.

subString(<cString>,<nStartPos>,<nCount>)->cSubString

Description:

subString() is a character function that extracts a substring from another character string.

Arguments:

<cString> is the character string to extract a substring from.

<nStartPos> is a numeric value and represents the starting position in **<cString>** for the extracted substring.

<nCount> is a numeric value and represents the number of characters to extract. If **<nCount>** is greater than the number of characters from **<nStart>** to the end of **<cString>**, the extra is ignored.

Returns:

cSubstring is a character string extracted from **<cString>**

Example:

```
getWordTwo()
{
var string_one, string_two;

string_one:= "I've got to know";
string_two:= subString(string_one,6,3);
message(string_two);                    % Displays 'got'
}
```

valid(<?AnyParameter>) -> nValue

Note !! This function has been superceded by the function objectIsValid(). In order to maintain future compatibility, it is recommended that you no longer use **valid()** and that you change your existing macro's to use objectIsValid() at your earliest convenience. While **valid()** is supported in this version, it may be removed from future versions.

Description:

This function returns 1 (True) if **<?anyParameter>** is valid and 0 (False) if not valid.

Arguments:

<?AnyParameter> is a parameter of any type.

Returns:

nValue is a non-zero (True) value if the parameter is valid and 0 (False) otherwise.

windowInput(<cControlId>)->?Value

Description:

This function will input a value from the macro window control that has a control id matching **<ccontrolId>**. The exact effect of this and the type of the return value depends on what type of control it is.

It is most often useful, and sometimes even necessary to determine the contents of a control before retrieving it's contents. This approach can both eliminate run time errors which a few controls generate under certain conditions, and assist you to write more robust macro's.

You will find the following functions especially useful in this respect...

getWindowControlRange()
getWindowInputError()
objectIsValid()
length()
windowInputValidate()

Arguments:

<cControlid> is a string type parameter which identifies the macro window control id whose value this function should return. Control Id's are user specified during macro window design.

Returns:

?Value is the value returned from this function. It's type is dependant upon the type of control that **<cControlid>** refers to.

Here are the return values and types for the different controls.

Checkbox

Returns a number type which will = 1 (True) if the checkbox is checked or 0 (False) if not.

Radiobutton

Returns a number type which will = 1 (True) if the Radiobutton is selected or 0 (False) if not.

Slider

Returns a number giving the position of the slider.

SBKLink

The function will return an SBKRef for the SBK that the link is pointing to, or an invalid value.

ValueIn

The function will return the number that the user has entered in the edit.
If the control is empty, a runtime error will occur.

NoteIn

The function will return the note number that the user has entered by keyboard or MIDI in the edit.
If the control is empty, a runtime error will occur.

TextIn

The function will return a string which will be the text the user has entered into the edit. It the control is empty, an empty string "" will be returned. You may test for an empty string value with the length() function.

Listbox

Returns a list containing the list indexes of the selected items in the list.
If no items are selected, an empty list {} will be returned. An empty list can be tested for with the length() function.

Combobox

Returns a list containing the list indexes of the selected item in the Checkbox.
If no item is selected, an empty list {} will be returned. An empty list can be tested for with the length() function.

windowOutput(<cControlId>,<?Value>)

Description:

This function will output a value to the macro window control **<cControlId>**. The exact effect of this, depends on the type of control.

Using this function on an unsupported control, will result in a runtime error.
Also, trying to write a value of the incorrect type to a control will generate a runtime error.

Arguments:

<?Value> is the value to write to the specified macro window control having the Control ID **<cControlID>**. Because different control types support different value types depending on their own type, the following list details the control types that are supported by this function and the values that they accept.

Checkbox

<?Value> is a number type. If **<?Value>** is non-zero, the checkbox is checked otherwise it will be unchecked.

Radiobutton

<?Value> is a number type. If **<?Value>** is non-zero, the Radiobutton is selected, otherwise it will be unselected.

Slider

<?Value> is a number type. **<?Value>** specifies the slider position.
If **<?Value>** is outside the range limits set for this control during macro Window Design, the value is simply ignored.

ValueIn

<?Value> is a number type. **<?Value>** will be displayed in the edit field of this control. If the number is outside the range limits set for this control during Macro Window Design, a runtime error will occur.

NoteIn

<?Value> is a number type. **<?Value>** will be displayed in the edit field of this control.

TextIn

<?Value> is a string type. **<?Value>** will be displayed in the edit field of this control.

Text

<?Value> is a string. **<?Value>** will be the text displayed in the control.

Listbox

<?Value> is a list of strings. Each string is an item that will displayed in the Listbox. The Listbox will display these strings.

Combobox

<?Value> is a list of strings. Each string is an item that will be displayed in the Combobox.

Returns:

None.

getWindowInputError(<cControlId>) -> nErrorValue

See Also: windowInputValidate()

Description:
This function is used to validate the contents of macro window controls.
Controls that could be empty when using windowInput() are..

ValueIn Controls
NoteIn Controls

These two controls should always be validated before using windowInput(). If windowInput() is used on these controls when they are empty, a runtime error will be generated.

You should therefore make a point of testing the value of these controls with this function before using windowInput().

This function can also test the values of other controls, but in this case, this function is used more for informational purposes.

Note that this function Cannot Test TextIn controls. A TextIn control when empty will return "" (an empty string) which is easily tested for by using the length() function.

Arguments:

<cControlId> may be any one of the following controls.

ValueIn Controls
NoteIn Controls
ListBox Controls
ComboBox Controls
SBKLink Controls

Returns:

nErrorValue is a numeric value and has a few options depending on the control type being tested with this function. Constants have also been supplied with this function, so you may prefer to use the provided constants instead of the numeric value when testing a condition.

Control Type	Possible Return Values	Constant Name
NoteIn	0	inputOk
NoteIn	-2	inputErrorNumberBelowRange
NoteIn	-3	inputErrorNumberAboveRange
NoteIn	-4	inputErrorNoNumberFound
ValueIn	0	inputOk
ValueIn	-2	inputErrorNumberBelowRange
ValueIn	-3	inputErrorNumberAboveRange
ValueIn	-4	inputErrorNoNumberFound
SBKLink	0	inputOk
SBKLink	-5	inputErrorSBKLinkEmpty
ListBox (Single Select)	0	inputOk
ListBox (Single Select)	-6	inputErrorNoSelection
ComboBox	0	inputOk
ListBox (Single Select)	-6	inputErrorNoSelection

windowInputValidate(<cControlId>) -> nValid

See Also: getWindowInputError()

Description:

This function determines whether the contents of the control, **<cControlId>** are valid.

Arguments:

<cControlId> represents the control id of a macro window control. It is a string value.

In the case of the ValueIn control that has upper and lower range limits that the user may set, this function will report the contents as invalid if the contents are outside of the specified range.

The following controls may be tested for valid contents.

NoteIn
ValueIn
Listbox (Single Select)
Combobox
SBKLink

Returns:

nValid is a numeric value. If the contents of the control being tested are valid, **nValid** will equal 1 or True. If the contents of the control being tested are invalid, **nValid** will equal 0 or False.

Macro Language Concepts

Function parameter and return syntax rules

Macro Windows In Depth

All About Functions

Operators
Variables (Scope, Declaration, Referring To)

Callback (How and why)

Commenting Your Macro Code
Line Continuation In Macro's

Control Structures (Loops)
Decision Making Structures (IF THEN)

Data Types (Numbers, Strings & Lists)

Objects
Lists In Depth
Object Parameters
Object Properties

Constants

Macro Callbacks

The Esbeekay Macro Language has certain library functions known as Callback Functions.

These functions, when declared in a macro, are called automatically when certain events take place in Esbeekay. These functions can be used to initialize macro variables, update lists and so on.

The following events and the functions which are automatically invoked upon the occurrence of these events are listed below.

Change to an	InstrumentObject	notifyInstChange()
Change to a	PatchObject	notifyPatchChange()
Change to a	SampleObject	notifySampleChange()
When a macro window closes		notifyWindowClose
When a macro window opens		notifyWindowOpen
The initial loading of Esbeekay.Exe		init()

Macro Callback Functions

Function parameter and return syntax rules

init

notifyInstChange(sbkObject)
notifyPatchChange(sbkObject)
notifySampleChange(sbkObject)

notifyWindowClose
notifyWindowOpen

Comments in Macro Programs

A comment is a line of text in a macro program which is ignored by the compiler. It's usual purpose is to make descriptive comments about the surrounding source code.

Comments may be formulated in various ways depending upon the effect that you want, but in all cases a comment is prefixed with %
There are no specific provisions for comment blocks in this language so for comments that span several lines you should use the following approach.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
% This is a comment in a macro program
% Macro to import instruments from one SBK into a new SBK
% Version 1.1
% John Doe
% 2/3/95
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
```

Comments may also be used *in line*
That is, a comment may appear on the same line as a program statement.

Because everything to the right of a % is seen to be a comment by the compiler, comments placed on the same line as a program statement should always appear to the right of a program statement and never to the left of a program statement.

This example is valid....

```
message("No patches found")        % Tell user that SBK has no patches.
```

If you do the following however, the statement *message("No patches found")*
will not execute. It will though, compile successfully.

```
% Tell user that SBK has no patches. message("No patches found")
```

Comments can be very useful when you come back to a macro at a later date and have to pick up your previous train of thought.
There is however one drawback. **The macro compiler does not count lines that consist only of comments.** Therefore when errors are reported by the compiler it flags the incorrect line number. If you can, create your macro first with no comments and no blank lines. Comment the macro when it is complete.
This shortfall will be addressed as soon as possible.

It has been noted also that in some instances where a comment is inserted after the last line of code, a syntax error is generated. This does not always occur and I have been unable to create this error at will. Just something to remember.

Constants

The Esbeekay Macro Language supports two global constants.

Constants are a representation of an actual value.

The two global constants supported are:

True = <> 0

False = 0

These values may be represented in your macro code by the constants.
These two constants are NOT case sensitive.

Example:

```
if objectIsValid() = False.
```

Further, you may find other local constants provided for certain functions.
These constants are documented along with their respective functions.
In all cases, the constant name represents a value which will be documented. The constant name name may be used instead of the numeric value in your macro code.

Continuation

Since the Esbeekay Macro Language interprets the semicolon ; as marking the end of a program statement, the language provides the facility to put more than one statement on a single line.

In some instances you may prefer to put more than one statement on a line for purposes of readability.

Example:

statement 1; statement 2; statement 3;

is equivalent to ..

statement 1;
statement 2;
statement 3;

Control Structures

There are times when you will want to execute a statement or group of statements repeatedly. This kind of construct is known as a loop. The **while** command is used for this purpose.

To execute a statement, or group of statements for every element contained in a list, the **for .. in .. do** command is used.

Descriptions of **while** and **for** follow.

while

The format of the **while** statement is..

```
while (expression)
        statement  1
```

or

```
while (expression)
        block statement
```

In the first format, statement 1 is executed if and only if <expression> is true. Statement 1 will continue to execute until <expression> is false.

To execute more than one statement if a particular expression is true, use the second format.
In a **while** loop, the loop body must contain a statement that alters the loop condition, otherwise, the loop will execute for ever.

Example:

A block statement for the **while** clause which beeps the speaker ten times.
The two statements, beep(); and counter:= counter -1; will be executed repeatedly until the while expression (counter > 0) is not true which will occur when counter = 0. At this point, the program will continue execution at the next line which displays 'finished' in a dialog box.
It is the braces {} which group the two statements together to form a statement block.

Note that <expression> (counter > 0) after being initially evaluated, is only evaluated at the completion of it's associated statement or statement block.
Note also that expressions for loops are always contained within parentheses () as in (counter > 0)

```
var counter;
counter:= 10

while (counter > 0)
       {
       beep();
       counter:= counter - 1;
       }
       message("finished");
```

for .. in .. do

The format of this command is ..

```
for <variable> in <listname> do
      statement
```

or ..

for <variable> **in** <listname> **do**
    statement block

This structure allows the user to execute a statement or statement for each element in <listname>. For each iteration, <variable> will take the value of the next element contained in <listname>.

As in the **while** command, this command allows either 1 statement or a statement block to be executed for every element contained in <listname>
If there are no elements contained in <listname>, the command is still valid. The associated statement or statement block will however not be executed.

Example using statement block :

Seeing as there are three elements in the supplied list, this loop will execute three times. This code would execute as follows..

Display 'apples'
beep
Display 'Oranges'
beep
Display 'Bananas'
beep

```
var fruits , type;
fruits:= "apples","Oranges","Bananas";

for type in fruits do
    {
    message(type);
    beep();
    }
```

or .. with a single statement.

```
for type in fruits do
    message(type);
```

Decision Making Structures

Decision making structures allow you to execute one or more program statements based on a condition.

Esbeekay provides one structure for this purpose.

The **if .. else** statement in the Esbeekay Macro Language takes the following format.

```
if <expression>
   statement 1
else
   statement 2
```

The **else** statement 2 portion is optional. In other words, this is a valid **if** statement.

if <expression> statement 1

In this case, statement 1 is executed if and only if <expression> is true.

To execute more than one statement if a particular expression is true, use a block statement.

Example:

A single statement for the **if** clause.

```
if (1=1)
      beep();
```

and a block statement for the **else** clause

```
else
      {
      beep();
      message("1 equals 1");
      }
```

Esbeekay Macro Functions

Function Calls

When you want to perform the same set of instructions on different sets of data or at different locations in your macro, you put those statements into a subroutine. Subroutines in the Esbeekay Macro Language are known as functions.

Functions are the basic building blocks of the Esbeekay Macro Language. These functions consist of a group of statements that perform a single task or action. They are similar to functions in C, Pascal or other major languages.
There is a complete library of predefined functions built into the language. You can also define your own functions.
The functions that you define will have a name (identifier) of your choice. They can receive values from other functions and they can also return values.

Functions defined within a macro have global scope. In other words, any one function within a macro, may call another function within that same macro.

Defining Functions.

A function definition consists of a function identifier (function name) with optional declared parameters contained within parentheses () following the function identifier.
The function then consists of a function body enclosed in {}.
The statements within the function body belong to the function and are the statements executed when the function is called by a statement.

A function definition begins with the function identifier and ends with the closing left brace }

Functions may be defined anywhere in a macro program but definitions may not be nested. A function definition has the following basic form:

```
<function identifier> ([<parameter,> ... <parameter>])
{
[<variable declaration>]
.
<executable statements>
.
}
```

The following is a typical example of a function definition.

```
MapSample(inst, at, low, high)
{
   var links, l;
   links := getInstToConISs(inst);
   for l in links do
      {
         setParam(l, "keyRange", makeKeyRange(at-low, at+high));
         setParam(l, "rootKey", at);
      }
}
```

Returning values from functions.

The form of the return statement is:

return [<value>];

In the above example function MapSample() , there is no return value. The **return** statement is optional as is returning values from a function. Where functions return a value, values of any data type can be returned using

the **return** statement as in ..
return <value>;

return statements can can occur anywhere in a function, allowing processing to be terminated before the function definition ends. **return** is limited by the fact that it can only return one value unless it returns a list. More than one return statement may be included within the function body.

Arguments And Parameters.

Passing data from a calling function to an invoked function involves two perspectives. One for the calling side and one for the receiving side.
On the calling side, the values passed are referred to as *arguments* or actual parameters. The following function call passes two arguments, One a string value and one a numeric value.

```
Display("hey You",1);
```

On the receiving side, the specified variables are referred to as parameters or formal parameters. For example, here the variables are specified to receive the string to display and a numeric value to indicate whether to sound the speaker.

```
Display(string,sound)

{
if (sound=1) beep();
message(string);
}
```

The specified receiving variables are placeholders for values obtained from the calling routine. When a function is called, the values specified as arguments of the functions invocation are assigned to the corresponding receiving values in the invoked function.

The specified receiving variables only have have scope within the invoked function.

Recursion:

Recursion is the calling of a function by a statement within that same function. When a function calls itself, it is said to recurse. A recursive call causes a new activation of the function. If the function includes a variable declaration, a new set of variables is created for each activation. A variable created by the function, is associated with the activation in which it is created and is visible in that activation and any lower level activations.

Macro Installation

Macros are installed from the Macro Support Dialog. This dialog is accessed from Esbeekay from the menu.

The <VIEW><MACRO> option will open the Macro Support Dialog.

The left hand side of the dialog shows a standard windows Open File dialog. Using the picklist in the Open File dialog, highlight the macro file you wish to install. You will notice that the macro description (if the macro has a description) is displayed at the lower left of the Macro Support Dialog. When you have highlighted the macro you wish to install, click the <INSTALL> button. The macro will appear in the right hand list along with the other installed macro's (if any). To "De-Install" a macro, highlight the macro of your choice in the right hand list and click the
<Uninstall> button. When you are finished, click the <DONE> button. Installed macro's will appear as an option on the Esbeekay <VIEW> menu, but only "Floating Macro's" are shown because Non- Floating macro's are started from Object Edit dialogs.

Lists In Depth

A list is a name for a group of items. Lists provide a convenient way of referring to a group of items. For instance ..
Let's say we want a group called Vegetables. This group could contain items such as peas,carrots..spinach. The first thing a macro needs to know is the name of the list. This is called the declaration. The declaration simply let's the macro know that we will be using a name. In our case that would be Vegetables. The macro then needs to know what items this group contains. So we need to say that the list Vegetables, contains peas and carrots and spinach.
We do this by means of assignment. In other words we assign peas, carrots and spinach to the list named Vegetables. This is how it's done.

```
Vegetables:= {"peas","carrots","spinach"}
```

the := is known as an assignment operator. It says that Vegetables contains everything to the right of :=

{} signifies the start and end of a list.
{"peas","carrots","spinach"} is therefore a valid list.
{} is also a valid list. It is however, empty.

To create an empty list you could do this ..

```
Vegetables:= {}
```

These items contained within the group are known as elements.
Each element in the group Vegetables, has a position in the group. peas is number 1 in the group while carrots is number 2 in the group. These elements are therefore said to be indexed.

To refer to list elements, you use the subscript operator []
You could do this for instance.

```
Vegetables[1]
```

This would return element number 1 contained within the list Vegetables.

Indexing of lists is from 0. The first element in a list is therefore referenced as [0]. Elements in a list can be of different types if you so desire. A list may therefore contain numbers, strings and even other lists.

Example1:

```
var mylist, item1, item2;          % Declare 3 variable names
mylist:= {};                       % Mylist is now an empty list
mylist:= {"Budgies",49};           % Mylist contains two elements
item1:= mylist[0];          % item1 is now = to "Budgies"
item2:= mylist[1];          % item2 is now = to 49.
```

Example2:

Lists may be concatenated as follows.

```
var list1, list2, list3;

list1:= {"Budgies",5};
list2:= {"Parrots","Canaries"};
list3:= list1 + list2;

% list3 is now equal to {"Budgies","Parrots","Canaries"}
```

Example3:

Lists, as previously mentioned, may also contain other lists. Consider the following:

```
var item1, list1;

items:= {"Apples","Oranges"}        % Item1 is now a list
list1:= {"Bananas",items}

% list1 is actually now equal to ...
% {"Bananas",{"Apples","Oranges"}}

% list[0] now equals "Bananas"
% list[1] now equals {"Apples","Oranges"} which is itself a list.
```

Existing list elements may not be replaced on an individual basis. This, for instance, is illegal.

```
item:= {"jack","john"};
item[1]:= "jill";
```

To accomplish this task, the following approach is required.
Copy the elements on an individual basis to a new list, making the element replacement as required. Assign the new list to the original list.

Objects

Objects are data types that reference the internals of an .SBK file. They can be passed as arguments and returned from functions just the same as the Primary data types.

They may be assigned to variables and they can also be elements in a list.

The hierarchy of an .SBK file in object form is ..

SBKRef
PatchObject
ConnectionFromPatchToInstrumentObject
InstrumentObject
ConnectionFromInstrumentToSampleObject
SampleObject

These objects define the .SBK file.
Manipulation of these objects involves changing their parameters and properties. Not all objects have properties and not all objects have parameters. Parameters are common in that, where an object has parameters, it's parameters will have the same names as parameters for other objects.
Properties are however, unique to an object type. A PatchObject for instance, does not have the same properties as a SampleObject.

Note that the SBKRef object has no properties and no parameters. It is used merely as a pointer to an SBK file. Seeing as Esbeekay supports multiple SBK files on the desktop, it is necessary to be able to specify which SBK contains the objects you wish to manipulate.

An SBKRef has	No Parameters No Properties
A PatchObject has	Parameters & Properties
ConnectionFromPatchToInstrumentObject	Parameters
InstrumentObject	Parameters & Properties
ConnectionFromInstrumentToSampleObject	Parameters
SampleObject	Properties

Supported properties and their associated objects:

Supported parameters:

Functions for manipulating object parameters:

getAllParams()
getParam()
removeAllParams()
removeParam()
setAllParams()
setParam()

Functions for manipulating object properties:

getObjectProperties()
setObjectProperties()

Functions to validate an object:

objectIsValid()

Operators

Operators, along with functions and variables, are the basic building blocks of expressions.

In the Esbeekay Macro Language, certain operators have different meanings depending on the data type of the operand(s). For example, the plus (+) operator can be used to concatenate strings as well as add numbers together.

All operators in the Esbeekay Macro Language require two arguments, called operands. The operators in this language are therefore known as binary operators. These binary operators use *infix* notation which means that the operator is placed between it's operands.

String Operators:

Symbol	Operation
+	Concatenate

List Operators:

Symbol	Operation
+	Concatenates lists or adds elements

Mathematical Operators:

The following table lists each mathematical operator and the calculation it performs. These operators require numeric type operands.

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division

Relational Operators:

The following table lists relational operators and their purpose.
The result of a relational operator is a logical value.

Symbol	Operation	May Be Applied To
=	Equal To	Any Data Type
!=	Not Equal To	Any Data Type
<	Less Than	Numbers And Strings
>	Greater Than	Numbers And Strings
<=	Less Than or Equal To	Numbers And Strings
>=	Greater Than or Equal To	Numbers And Strings

Logical Operators:

All of the logical operators require logical operands. The result of a logical operation is always a logical value.

Symbol	Operation
&	Logical AND
|	Logical OR

!	logical NOT

Assignment Operators:

The Esbeekay Macro Language has one assignment operator which requires a single variable as the first operand. The second operand may be any valid expression. The variable used as the first operand must have been previously declared with the var command. It does not have to be the same data type as the second operand.

Symbol	Operation
:=	Assign

Special Operators:

These are special operators that sometimes appear in expressions.

() Function Or Grouping

Parentheses () are used in expressions to either group certain operations in order to force a particular evaluation or to indicate a function call. When specifying the grouping operator, the item that falls within the parentheses must be a valid expression. Sub-expressions may be further grouped. With function calls, a valid name must precede the left parenthesis and the function arguments, if any, must be contained within the parentheses.

[] List Element

The subscript operator [] is used to reference a single list element. The name of a previously declared list must precede the left bracket and the list element index must appear as a numeric expression within the brackets.

{} List Definition

Curly Braces {} are used to create a list. In it's simplest form {} this signifies an empty list. ie. No elements in the list. List elements where specified must be contained within the braces and separated by commas.

Curly braces {} are also used to create block statements. A block statement is a collection of statements that can be put anywhere a single statement can.

A block statement takes this form.

{ <statement;> <statement;> ... <statement;> }

Supported Object Properties

This is a list of object properties that are currently supported by the Esbeekay Macro Language:

To retrieve these properties from an object, use the function getObjectProperties(). Seeing as the properties are returned in a list, you will need to use the list index to access them on an individual basis. The list indexes are documented alongside the property name.

Example:

```
properties:= getObjectProperties(mypatch);
message(properties[0])   % Displays Patch Name.
```

Object	**Property Name**	**List Index Data Type**
PatchObject	Patch Name	[0] Numeric
PatchObject	Patch Number	[1] Numeric
PatchObject	Bank Number	[2] Numeric (128=Percussion)
InstrumentObject	Instrument Name	[0] String
SampleObject	Header Name	[0] String
SampleObject	Ram/Rom Semaphore	[1] Numeric (1= rom,0 = ram)
SampleObject	Start Position	[2] Numeric
SampleObject	End Position	[3] Numeric
SampleObject	Loop Start Position	[4] Numeric
SampleObject	Loop End Position	[5] Numeric

Data Types

Data types in Esbeekay consist of both Primary and Secondary types.
For information on secondary data types, see objects.

These are the primary data types used in the Esbeekay Macro Language.

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 occurs automatically when necessary. This means that integer accuracy is maintained as long as possible.

EXAMPLE:

```
var patchnumber;
patchnumber:= 24          % patchnumber now = 24
```

String:

Strings can be up to 32767 bytes in length. Strings are enclosed in double quotes ""

EXAMPLE:

```
var name;
name:= "John Doe";        % Name now = "John Doe"
```

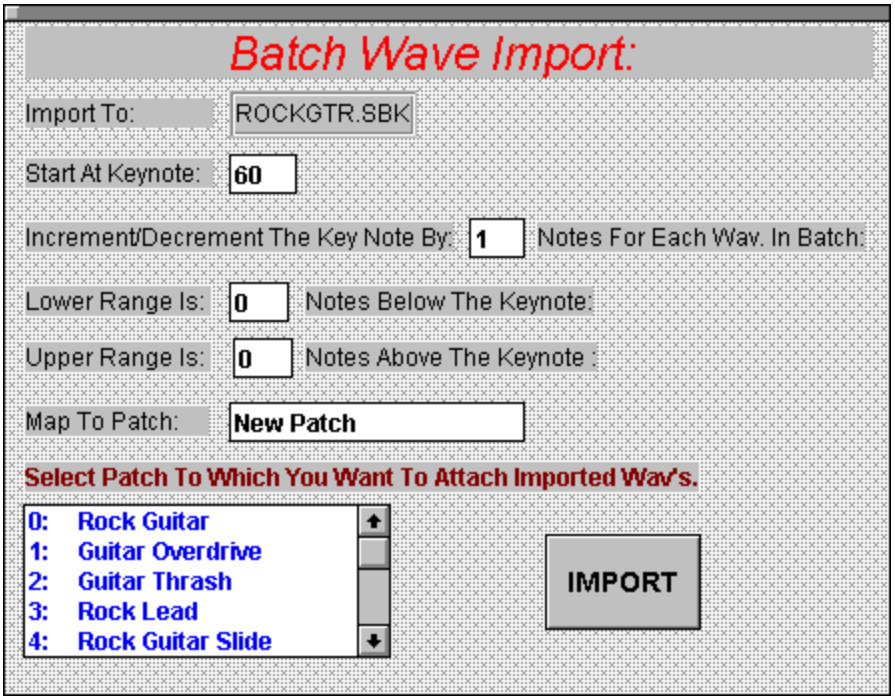
List:

Lists contain elements which are indexed. Lists may be thought of as arrays.

To reference list elements, you need to use the subscript operator []

Indexing of lists is from 0. The first element in a list is therefore referenced as [0]. Elements in a list can be of different types. A list may therefore contain numbers, strings and even other lists.
For further information on lists see Lists In Depth

A Typical Macro Window



A Typical Macro Program

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Macro to reverse samples using a DLL
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Global Variable Declaration
var listSBK, curSamples;

init()
{
   fillNames();        % Call the function Called fillNames()
}

fillNames()
{

var sbk, allsamps, s, names, sd;

   sbk                  := getSBKLink("SBKLink1");
   listSBK           := sbk;
   names            := ;
   curSamples := ;

   % Fill list box with names of RAM samples only
    if (valid(sbk)
   {
      allsamps := getSBKSamples(sbk);
      for s in allsamps do
      {
         sd := getObjectData(s);
         if (!sd[1])
         {
            names            := names + sd[0];
            curSamples := curSamples + s;
         }
      }
   }
   windowOutput("samplist", names);
}


{
   fillNames();
}

notifySampleChange(sbk)
{
   if (sbk = listSBK) fillNames();
}

reverse()
{
   var s, sr, from, to, sd;

   if (!valid(listSBK)) return;
   s := windowInput("samplist");
```

```
    if (s = -1)
    {
       message("Select a sample first");
       return;
    }
    sr := curSamples[s];
    if (!valid(sr)) return;

    sd      := getObjectData(sr);
    from := sd[2];
    to      := sd[3];
    callDLL("reverse", "reverse", listSBK, from, to);
    notifySBKDataChanged(listSBK);
}
```

Variables

A variable is a label that we give to an area of memory that contains a stored value.

Declaration Of Variables:

var <identifier> [,<identifier>] ... ;

In order to use a variable, it is necessary to first declare it. In simple terms, we assign a label or identifier to a memory area that will store a value.

Variables are declared using the keyword *var*

Examples:

var patchname, patchnumber, banknumber;

var SbkFiles;

Whenever you declare more than one variable, there must be a comma , inserted between the names. A variable declaration is a statement and must, as such, always be terminated with a semicolon ;
Case of variable labels may be upper, lower or a combination of both. Remember however, that to refer to the variable in your macro, you must use the exact same case as the label.

Assignment:

<identifier>:= <value>;

Once you have declared a variable it may be assigned a value. In other words we can store a value to a memory area but we refer to that area with the variable label.

First the declaration..

var patchname, patchnumber, banknumber;

Then the assignment..

patchname:= "drums 1";
patchnumber:= "25";
banknumber:= 0;

Typing of variables:

Variables in the Esbeekay Macro Language are **not typed** at declaration time. Variables can take on any valid type and can change their type during run time. Variable typing is therefore completely dynamic.

The following statements are valid.

var a;

a:= "Esbeekay";
a:= 5;

Scope Of Variables:

Local variables.

Variables declared within the body of a function are visible only within the function in which they are declared. Variables declared within a function can only be made visible to another function by passing them to another

function as parameters.

Global variables.

It is possible to declare variables in such a manner that they are visible to every function within a macro. Such a variable is said to have file-wide or global scope.

Declaration of global variables is achieved by declaring the variables outside of any function body at the beginning of the macro program.

Macro Windows

Every macro has a window defined which you must design using the window designer. The window designer is started from the View/Macros... dialog.

Macro Windows may be either **floating** or **attached**

Where the Macro Window is "floating", the macro can run with the window open or closed. When Esbeekay starts up, it will start each floating macro with the window in the state it was when Esbeekay was last shut down.

Where the macro window is "attached to" another window, such as a sample edit dialog, one instance of the macro will be opened for each sample edit dialog which is opened.

Each window can contain elements such as buttons and edit controls.
All of the controls have properties that can be specified as appropriate such as font, color and allowable input values.

Every macro window control has a textual identifier property. This is referred to as its "Control Id". Control id's are used in macro library functions to refer to controls.

They also have a special function for buttons, SBKLinks, list boxes 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 a control id of "load" (excluding the quotes), then if there is a function in the macro load() then that function will be called when the button is clicked.

To retrieve a value from a macro window control the windowInput() function is used. To set the value of a macro window control windowOutput() is used.

The following controls may be placed in a macro window.

Buttons:

Default Control Id: Button1
Can Be Resized: Yes

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. BMP Images are not automatically resized. You should make sure therefore, that the button and/or the graphic, is sized accordingly.
Where the button is defined to contain text, the text font and color may be specified. Where a function having the same name as the control id of the button is contained within the macro, that function will be called when the button is clicked.

Checkboxes:

Default Control Id: Checkbox1
Can Be Resized: Yes
Default Value: 0
windowInput() returns 1 when checked and 0 when unchecked.

Checkboxes have a text property in addition to their control id. This text will be displayed as part of the checkbox in the macro window. The font and foreground color of this text can be specified as required.

Radio Buttons:

Default Control Id:	RadioButton1
Can Be Resized:	Yes
Default Value:	0
windowInput() returns	1 when ON and 0 when OFF. (Default is 0)

Radio Buttons have a text property in addition to their control id. This text will be displayed as part of the Radio Button in the macro window. The font and foreground color of this text can be specified as required.

Where a function having the same name as the control id of the radio button is contained within the macro, that function will be called when the button is clicked.

Sliders:

Default Control Id:	Slider1
Can Be Resized:	No
Default Value:	The value entered into the From: Field during design.
windowInput() returns	The current value of the control.

Sliders have three properties in addition to their control id.

Notch Centered: When this property is checked, the slider will show a notch at the center point of the slider.

From: This is the value that the slider will return at it's lowest point. The default is 0. Negative numbers are allowable.

To: This is the value that the slider will return at it's highest point. The default value is 100. Negative numbers are allowable.

By making the From: value higher than the To: value, the slider can in effect be reversed. ie. High to low instead of low to high.

By double clicking on the slider control, a value can be entered directly.

Text:

Default Control Id:	Text1
Can Be Resized:	Yes
Default Value:	The text property of the control
windowInput() returns	The text property of the control.

Text controls are used solely for displaying text in the macro window. The font and foreground color of this text can be specified as required.

ValueInput:

Default Control Id: ValueIn1.
Can Be Resized: Yes
Default Value: EMPTY
windowInput() returns A numeric value as entered.

This control is used for retrieving a value directly from the user. This control has two properties that can be used to enforce allowable values that can be entered. From: & To:
An entered value that falls outside these limits as specified by From: & To: will generate a message dialog when the macro tries to retrieve the value of this control using windowInput().
Default values for From: & To: are -32768 and 32767 respectively. The font and foreground color of the value displayed in this control can be specified as required. If the user attempts to enter Alphbetic characters into this control, they are rejected.

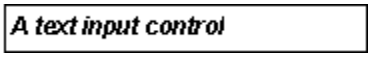
NoteInput:

Default Control Id: NoteIn1.
Can Be Resized: Yes
Default Value: EMPTY
windowInput() returns A numeric value as entered.

Values can be entered directly but the upper and lower ranges are set to 0 and 127 respectively.
An entered value that falls outside these limits as specified by From: & To: will generate a message dialog when the macro tries to retrieve the value of this control using windowInput().

If focus is placed in this control, it will accept MIDI notes as input as well as keyboard entered values.
The font and foreground color of the value displayed in this control can be specified as required.
If the user attempts to enter Alphbetic characters into this control, they are rejected.

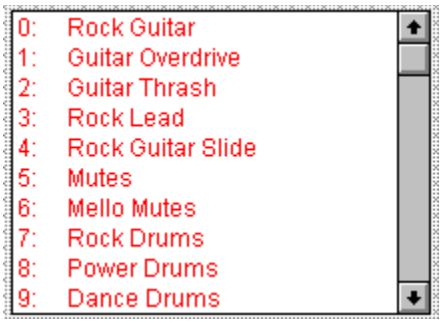
TextInput:



Default Control Id: TextIn1
Can Be Resized: Yes
Default Value: EMPTY
windowInput() returns The text value as entered

This control is used for inputting text. The font and foreground color of the entered text can be specified as required.

Listbox:



Default Control Id: Listbox1
Can Be Resized: Yes
Default Value: EMPTY
windowInput() returns The subscript of the selected list item

If multiple select is checked for a listbox control, the following keystrokes may be used:

<SHIFT><CLICK> Extends the selection.
<CTRL><CLICK> Toggle status of individual item.

This is how to put items into a ListBox for selection.

```
var mylist;
mylist:= {"Apples","Oranges","Bananas"};
windowOutput(<controlid>,mylist);
```

ComboBox:

Default Control Id: Combobox1
Can Be Resized: Yes
Default Value: EMPTY
windowInput() returns The subscript of the selected list item

This is how to put items into a ComboBox for selection.

```
var mylist;
mylist:= {"Apples","Oranges","Bananas"};
windowOutput(<controlid>,mylist);
```

SBKLink:

Default Control Id: SBKLink1
Can Be Resized: Yes
Default Value: EMPTY
windowInput() returns An SBKRef reference to an SBK File

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. Clicking on this control while there is more than one SBK on the desktop will clear the SBKLink.

Creating & Editing Macro's

This topic will explain how to open a code window for an existing macro or a new macro which you want to create. It will also explain how to open a macro window for an existing macro or for a new macro that you want to create. It will NOT explain how to write macro code. You should refer to the Basic Concepts menu for more detailed assistance with macro code.

Macro's are both created and edited from the Macro Support Dialog. The <VIEW><MACRO> option will open the Macro Support Dialog.

To Create A New Macro:

By means of the File Dialog, on the left hand side select a directory where you want the new macro to reside. Macro's in Esbeekay may reside in any directory of your choice. There is also an .INI file option to specify your default macro directory.

In the file field at the top left of the Macro Support Dialog, type in a name for your new macro. No file extension is necessary. Esbeekay will automatically append the correct extension which is .mac for Esbeekay macro's.

If you want to edit an existing macro, instead of typing in a new file name, just highlight the macro filename you want to edit.

Code or Window ?

To edit or create code for the selected or new macro, click the <CODE> button.
If this is for a new macro, Esbeekay will ask you if you want to create a new macro. After clicking <OK> in this instance, you will be presented with an empty Notepad. It is here that you write your macro code. Note the field at the bottom of the Notepad which is the optional macro description. If this is for an existing macro, the Notepad will contain the code for the macro that you selected. To study existing macro code is an excellent way to learn the Esbeekay macro language.

When you are finished with the code in the window, click <OK>. Esbeekay will compile the macro and show any errors you encounter. You should correct the errors and continue until there are no errors left.

If this is a new macro, you will very likely want to install the macro. See Installing Macro's.

To edit or create a Macro Window, click on the <WINDOW> button. You will be faced with a window which may or may not contain macro controls depending on whether this is for a new or existing macro.

For a new macro window, you will see the macro window itself displayed in the editing window. Usually Grey in color. An INI setting is available to change the default color for macro backgrounds.

You will notice that the macro window has "Handles" at it's edges. By dragging the handles, the background can be sized. By dragging from the center of the object, the object can be moved.

To place new controls on the macro window, click on the control buttons at the bottom of the dialog. The control will be placed on the macro window. You can then drag and size it to your requirements. Not all objects can be resized though.

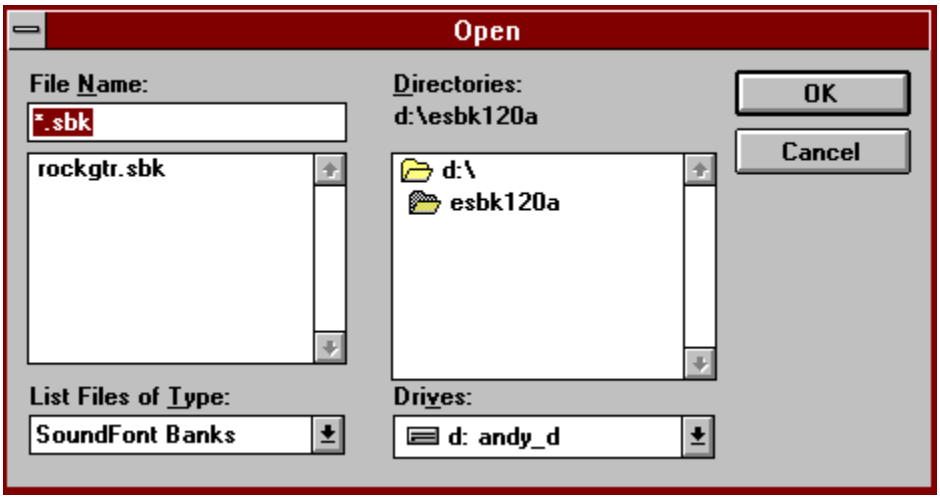
When you click on a control placed on the macro window you will see fields at the bottom of the dialog. These are the properties for this control. These properties may be altered to your liking. For a more complete information on the effect of properties on controls see Macro Windows.

The properties are not the same for all control types. They all have one thing in common though, and that is the Control Id. This is a label that you may type in to identify the control. Esbeekay does provide default Control ID's when you place a new control. You may just want to use these instead of typing in your own. In both cases, this is how the Esbeekay Macro Language identifies macro controls. A little later on, when you are studying the macro language, you will notice that many commands (functions) need to know the Control ID. for a control, in order to retrieve it's contents.

The macro window is how a user communicates with a macro. The controls are what the user sees and enters values into. The macro then retrieves these values from the controls and then makes changes to the SBK file accordingly. To get an idea of the elegance that can be achieved with the Window Designer have a look at a Typical Macro Window.

To delete a macro window control, just click on the control you want to delete and click the <DELETE> button.

When you are finished with you macro design, just click the <OK> button.


Open
File Name:
*.sbk
rockgtr.sbk
Directories:
d:\esbk120a
d:\
esbk120a
OK
Cancel
List Files of Type:
SoundFont Banks
Drives:
d: andy_d

Outline Of Macro's In Esbeekay

An Esbeekay macro program can manipulate objects within SBK files at the lowest level. An Esbeekay macro program can also perform tasks such as importing data into SBK files and copying data between SBK files.

Esbeekay macro's consist of a Macro Window and a Macro Program, stored in a single file with a .mac file extension. The macro window can contain controls such as buttons, list boxes, sliders, text fields etc. Each of these controls have a unique Control ID. The Control ID is a textual identifier that is assigned to a control during design of a macro window. Control ID's are used by the macro language to refer to these controls. Macro windows can appear as pop up windows (floating) which can be hidden or they can be attached to existing dialogs contained within the Esbeekay program. The macro program, while having the ability to retrieve values from controls within it's corresponding macro window, can also write new values to these controls.

In order to use a macro from within Esbeekay, the macro must be *installed*. See Macro Installation. The macro (if a floating macro) will then appear as an item on the *View* menu from where it can be executed. More than one macro may be installed. Once a macro is installed it will remain installed until deinstalled.

Whenever Esbeekay starts, it loads the *installed* macros. These macros are automatically started. It is possible to declare the optional init() function in a macro program. The init() function is called automatically when the macro starts. This provides a means of initializing the corresponding macro window controls with suitable default values.

An Esbeekay macro program, when executing, can be interrupted at any time by pressing <CTRL><C>

Supported Parameters

This is a list of parameters that are currently supported by the Esbeekay Macro Language:

Parameters named as NULL are either unused or unknown at this time.

Sample Names	Hex Values	Decimal Values
"sampleStartOffset"	// 0x00	0
"sampleEndOffset"	// 0x01	1
"localLoopStartOffset"	// 0x02	2
"localLoopEndOffset"	// 0x03	3
"pitchOffsetCent"	// 0x04	4
"LFO1Pitch"	// 0x05	5
"LFO2Pitch"	// 0x06	6
"EG1ToPitch"	// 0x07	7
"LPFFC"	// 0x08	8
"LPFFQ"	// 0x09	9
"LFO1ToLPF"	// 0x0A	10
"EG1ToLPF"	// 0x0B	11
NULL	// 0x0C	12
"LFO1ToVolume"	// 0x0D	13
NULL	// 0x0E	14
"chorus"	// 0x0F	15
"reverb"	// 0x10	16
"pan"	// 0x11	17
NULL	// 0x12	18
NULL	// 0x13	19
NULL	// 0x14	20
"LFO1Delay"	// 0x15	21
"LFO1Freq"	// 0x16	22
"LFO2Delay"	// 0x17	23
"LFO2Freq"	// 0x18	24
"EG1Delay"	// 0x19	25
"EG1Attack"	// 0x1A	26
"EG1Hold"	// 0x1B	27
"EG1Decay"	// 0x1C	28
"EG1Sustain"	// 0x1D	29
"EG1Release"	// 0x1E	30
NULL	// 0x1F	31
NULL	// 0x20	32
"EG2Delay"	// 0x21	33
"EG2Attack"	// 0x22	34
"EG2Hold"	// 0x23	35
"EG2Decay"	// 0x24	36
"EG2Sustain"	// 0x25	37
"EG2Release"	// 0x26	38
"keyToHold"	// 0x27	39
"keyToDecay"	// 0x28	40
NULL	// 0x29	41
NULL	// 0x2A	42
"keyRange"	// 0x2B	43
"velocityRange"	// 0x2C	44
NULL	// 0x2D	45
"forceKey"	// 0x2E	46
"forceVel"	// 0x2F	47
"volume"	// 0x30	48
NULL	// 0x31	49

NULL	// 0x32	50
"pitchOffset"	// 0x33	51
"pitchAdjust"	// 0x34	52
NULL	// 0x35	53
"loop"	// 0x36	54
"pitch"	// 0x37	55
"quartertone"	// 0x38	56
NULL	// 0x39	57
"rootKey"	// 0x3A	58

The Esbeekay Macro Language

Documentation for the macro language prepared by Andy Robinson.
Errors, suggestions and comments for the macro documentation to ..

INTERNET: APR@PIXIE.CO.ZA
CIS: 100100,1220
SNAIL: Box 3056, Symridge, 1420, R.S.A

Parameter EG1 Attack 0x1A

Envelope 1 controls both pitch and filter cutoff frequency.

Attack is the time taken for the envelope to rise from 0 to it's peak level.

It is suspected that if this value is set to 0, that the Low Pass Filter becomes velocity sensitive, responding to velocity values > 60 thus making the sound brighter as the key is struck harder.

Editing SoundFont Banks

The main window for SBK's 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 sbk's 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.

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]

AWEMAN32Directory	*Directory setting for Windows NT Beta Drivers*
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*
MacroDirectory	*Default directory in which user macros are located*

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. Multiple loads are allowable.*

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, Win32s v1.2, 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 you feel you can afford it, then by all means do send along a contribution. A suggested level would be 1 pound sterling. If however you do not pay anything, then that is fine too.

Click here for 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.

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 WAVE'S. 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.

Notes

The sample test play rate is supposed to work for any sampling rate. However, the AWE32 Windows Wave driver doesn't 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.

On Windows NT there are currently no AWE32 drivers, and testing patches does not work on that system. NT drivers are currently in developement as at the time of this writing.

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!

First, patch testing only works on Windows, not Windows NT because the NT drivers for the AWE32 are still in developement. At this time of release though, this situation may have changed.

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.

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-builder's point of view. There is a separate root key parameter, which can be set but in early SBK's 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.

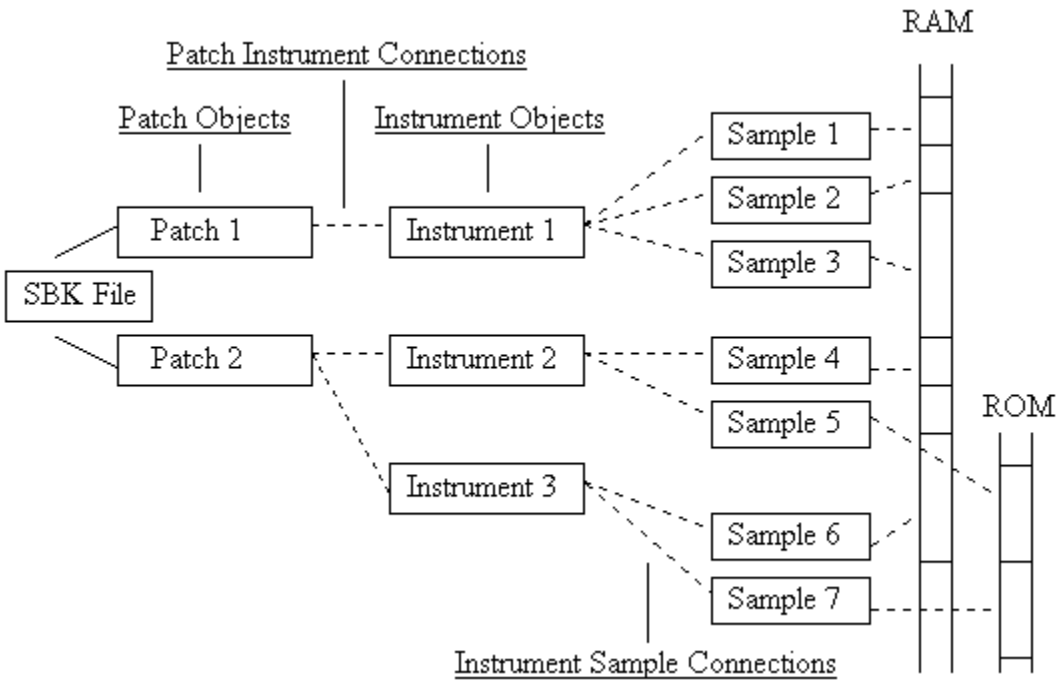
Looping

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.

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 SBK's 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 SBK's which come with the AWE32.

From examination of standard SBK's 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, LFO's 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 SBK's 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.

<u>Known parameters</u>

Quick Start Guide to using SBKs

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

1: 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).

2: Store the SBK in your SoundFont directory (probably) as synthusr.sbk. Then edit the file sbwin.ini in the Windows directory so that the line beginning with USER reads USER=C:.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.

Esbeekay v1.2 Help