Programmer’s Guide

Purpose of this Guide

The purpose of this section is to give detail of the way in which *Sound Studio* has been implemented. It aims to provide sufficient information to the programmer, so that the source code is accessible. This section does not propose to give details of individual functions and procedures, these are well documented in the source code. A directory of procedures and functions will be provided as a map to the listing. A knowledge of TCL/Tk is assumed.

Architecture

The architecture of *Studio* is a patchwork of software bound together by a TCL/tk shell script. The TCL/tk script sets up the user interface and defines the call-back routines. These call-back routines make use of command-line applications including system commands to perform their function. The overall structure is illustrated in figure 1.

Data Flow

There are two types of data flow in *studio*, internal and external. The internal data flow of *Studio* is provided by global variables. This is really the only sensible way to handle data flow in a TCL/Tk application. The external form of data flow is that of obtaining data from external applications. This is provided by ensuring that the application provides information to the standard output. This can then be stored in a TCL variable, in the following manner.

set *TCL_Variable* [exec *APP_Name -App_Options*]

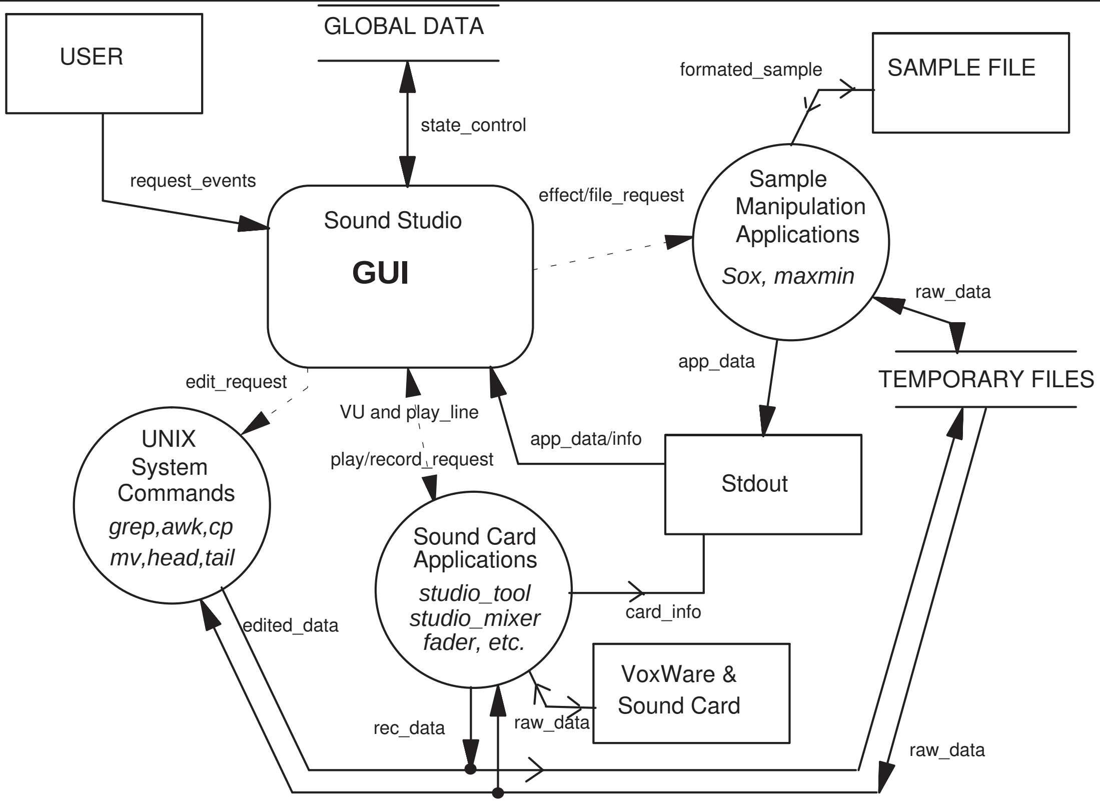


figure 1 - *The overall structure of Studio*

USING TCL ARRAYS

As has been explained the data flow in this application is largely provided by global variables. The number of global variables required is large thus making it difficult to oversee and remember all the global names.

To overcome this problem TCL arrays were used. Using TCL array has two main advantages. Firstly, it allows the data organised into groups of similar data. Secondly, it reduced the number of `global` declarations required within each procedure, since only the array name needs to be declared.

An additional advantage is that the indices of TCL arrays may be text, thus keeping the purpose of each array item clear. Performing operations on a complete array is easy using the `array names` command within a `foreach` statement.

Table 1 contains an overview of the groups of data in *Studio*. A detailed list of all global variables used is found in table 3 in the listing directory.

Array Name	Content
CARD	The diagnostic data of the sound card.
ConfigInfo	Descriptive strings for configuration file and use in the user interface for setting colours etc.
ConfigSet	The configuration options which affect the look of the GUI, e.g. colours and fonts.

FILE	File names and paths for components of *Studio*.
FLAG	The state flags.
FX	The effect parameter values and application switches.
HELP	The global variables belonging to the on-line help module.
INFO	Formatted strings for display in the InfoBar.
LINE	Information about play-position line
LOAD	Sampling parameters of the working sample file.
MACHINE	Machine-dependent (Solaris or Linux) information
MIXER	Variables for the mixer board, including volume settings for the mixer devices.
PACK	Information for the packing order of sections.
PLAY	Global variables required for the functioning of the estimated time.
PLOT	Global variables required for editing plot manipulation. Including position of insertion points.
SAMPLE	Sampling parameters of the original sample file.
VU	Pertaining to the appearance and function of the VU meters.

table 1 - *Arrays in Studio*

Temporary Files

Studio makes use of temporary files to handle the sample data. This decision arose out of the necessity to perform *Sox* operations on the sample data in the form of a UNIX pipe. There are five temporary files. These are stored in the directory specified in the *studio* start-up script. The directory /tmp is ideal for this purpose. The temporary files are created with a long file name, which contains the system clock time in seconds in it's name. Thus a set of temporary files exists for each instance of *Studio* running.

The temporary files are:

1. TMP1. This is the *working file.* The formatted sample file is piped though *Sox* upon loading and the raw sample data is stored in this file. When the sample data is edited it is generally stored here.
2. TMP2. This is the *clipboard file*. This contains the data that has been copied to the clipboard. This file is also used to hold the appropriate period of silence when the Edit→Insert Silence menu option is invoked.
3. TMP3. This is the *Sox error file*. The sample data that is extracted by *Sox* is sent to the standard error file, not standard out. Thus it seems that the only way to get to this information is to store it in this file and then read it into TCL.
4. TMP4. This is the *undo file*. A copy of TMP1 is stored here when an editing operation is about be performed, so that the changes may be reversed by copying this file back to TMP1.

5. TMP5. This is the *overflow file*. There is often need for a *temporary* temporary file. This is especially the case when using the preview mode.
6. TMP6. This is the *status-scan file.* This file was introduced at version 0.2.6, and makes possible the avoidance of the `-e stat` flags of the ***sox*** program. Using these flags causes some problems with more recent versions of ***sox***. Having required the use of this file, it is also used to break up the susequent invocations enabling extra functionality when saving to `raw`/`wav` file types.
7. TMP7 is a miscellaneous scratch file. It is used for grabbing outputs from, eg, sox for later parsing, and also as the record destination when the set-level operation (middle click on the Record button) is being performed.

Editing Method

Studio edits the sample file by manipulating the above temporary files.

Generating the Plot

The plot is drawn on a canvas widget. It consists of one line per horizontal point in the canvas co-ordinate system. The vertical co-ordinates are determined by the maximum and minimum points of a block of the sample. The program **maxmin.c** was written solely for the purpose dividing the sample file into blocks and scanning each block for the maxima and minima. It then prints the results to stdout in TCL list format. This output is normalised and inverted due to the reverse nature of the X- co-ordinate system (see figure 2). The control of the block size is left to the caller to maxmin. Thus the resolution of the plot may be adjusted, as in the Zoom function. It should be noted that drawing into the canvas is time-consuming. It is for this purpose that *Studio* has restricted the size of the canvas to not extend beyond its physical size.

Maxmin requires bit resolution and number of channels in the sample, since theses vary the size of the samples.

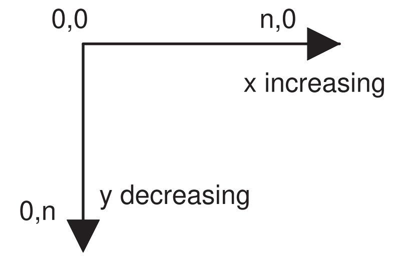


figure 2 *-X Co-ordinate system*

Graphical editing

All the graphical editing is performed in terms of the canvas. Each line in the plot represents a block of sample data. Upon loading the complete sample is fit within the boundaries of the canvas. Thus the initial resolution is determined by the size of the sample file.

When marking, the boundary positions of the marked area are stored as canvas co-ordinates [PLOT(Startx), Plot(Endx)]. These are then used to determine the amount of data to be skipped at the head and tail of the file to perform the editing operations of *copy*, *delete* and *paste*. These editing operations operate on the working file using the programs ***head***, or ***tail***.

Zoom Levels

The *Zoom* operation complicates the editing concept, since the blocks size (bytes per vertical line) changes as the resolution is increased.

To keep track of each zoom operation the context of the current zoom level is stored when a *Zoom-Up* operation is performed. By context is meant the positions of the insertion points and the block size. This is stored in a global called PLOT(Zoom*ZOOM_LEVEL*), where *ZOOM_LEVEL* is a discrete number $\geq$ 0.

With each context change the co-ordinate system is shifted; the left insertion point is adjusted to the x=0 position. Thus it is necessary to record this offset [Plot(Offset)]. Figure 3 illustrates this point.

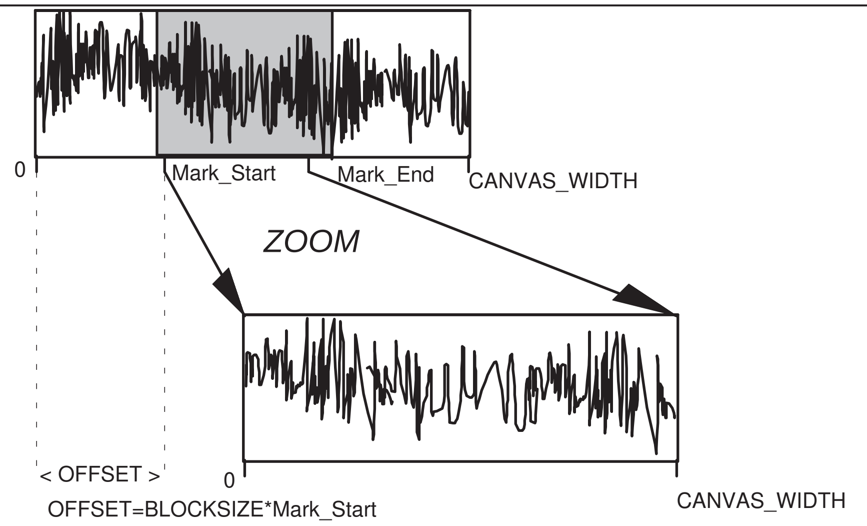


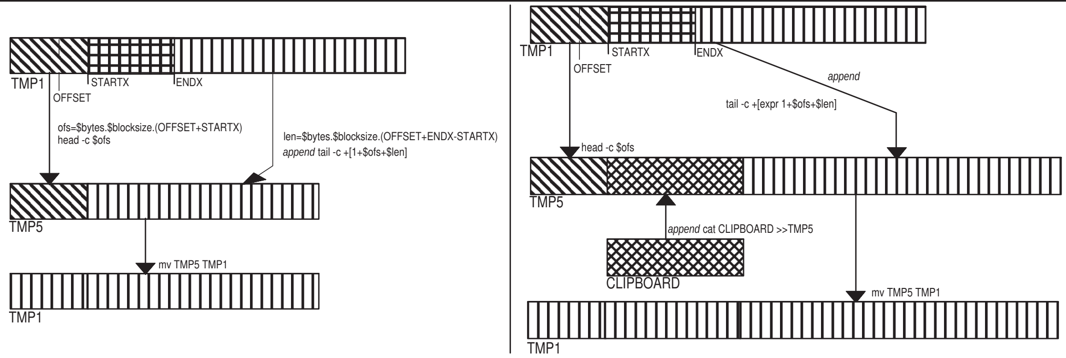
figure 3 - *Graphical editing*.

Furher, as of version 0.8, it is possible to scale the view of the editing canvas. This facility is available in addition to the zooming function described above. When working on a large file, it is always advisable to zoom to the area of interest, as the updating of the canvas and application of effects is slower when the whole file is considered rather than a zoomed section. Magnification does not section the file in the way that zoom does; it is purely a graphical convenience.

The selection of magnification level is by pull down menu, which permits the user to jump several magnification steps at once. This avoids *studio* having to rescan the file as the user repeatedly presses the "magnify" button, as would be the case if it were implemented with a similar user interface to the zoom-in and zoom-out functions.

Using head and tail

The way in which `head` and `tail` are used in the context of manipulating the working file is illustrated with the examples of the *Delete* and *Paste* operations in figure 4.



(a) Delete (b) Paste

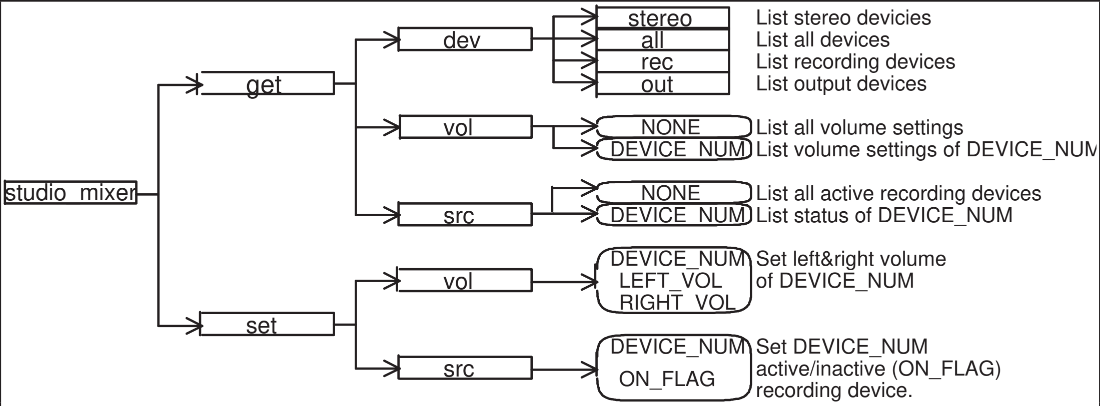
figure 4 *- Deleting and Pasting using head and tail*

Sound Card Interface

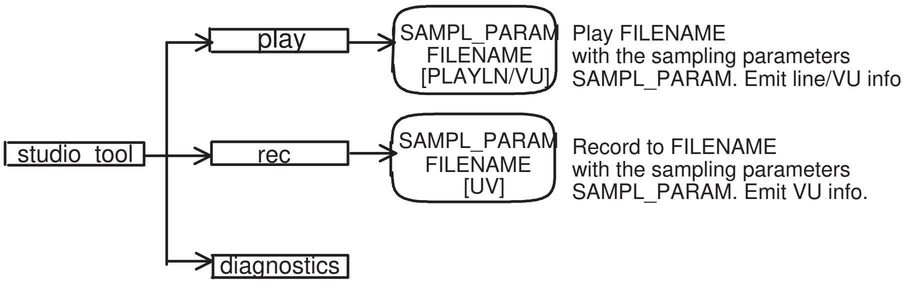
The sound card interface is a straightforward C-implementation of the content of the VoxWare Hacker’s Guide. There are two programs that have been written for the purpose of the sound card interface;

1. *studio_tool* (s_tool.c), the dsp interface and diagnostics.
2. *studio_mixer* (s_mixer.c), the mixer interface.

Both these have an interface that resembles the form of a TCL command. This is because they consist of a number of activities. A syntax guide is shown in figure 5.



(a) *studio*_mixer



(b) *studio*_tool

figure 5 - *Syntax of Sound Card Applications.*

The diagnostic routine simply loops through the sampling rates, channels and bit resolutions, to determine the limits of the card. This is useful for setting the parameter limits in the interface.

Module overview

Studio is split into several files. The C-code listings are naturally in separate files. The TCL/Tk listings are divided into files, or modules. These modules are organised to make it relatively easy to find the required code segments.

There is a module file for:

- each menu button in the menu bar.
- each *Studio* section.

The exception to this rule is the effect module, which combines the effect bar and effect menu button into one, since they are both very similar in function.

Additional modules are

- the initialisation module. This ties all the modules together.
- the miscellaneous module. This consists of routines that don't clearly fit into other modules, or are procedures that are self-sufficient, and thus re-usable.

Table 2 gives a summary of the *Studio* modules.

Module	File Name	Purpose
Edit menu	edit.tk	Provide call-backs to menu items
Effect	effect.tk	Provide call-backs to menu items
File menu	file.tk	Provide call-backs to menu items
Help menu	help.tk	Provide call-backs to menu items
Information Bar	info.tk	Create Information bar and associated procedures.
Initialisation	init.tk	Set-up initial values and bind modules together.
Menu Bar	menu.tk	Create Menu Bar and menus.
Miscellaneous	misc.tk	"Global" routines.
Mixer Board	mixer.tk	Create and set-up the mixer board. Provide the mixer routines.
Options menu	option.tk	Provide call-backs to menu items
Control Panel	playpanel.tk	Create and set-up the control panel. Provide the sound card routines.
Editing Plot	plot.tk	Create and set-up the canvas and associated procedures.
Start-up	*studio*	To allow entry of required paths, and call initialisation.

table 2 - *Module Summary*

Additionally, the Solais version of Studio has its own mixer module which communicates via small C programs with Sun's audio pseudo-device. The programmer is referred to the appropriate source files for further information.

Procedure Categories

Since *Studio* is not a single sequential program it is difficult to show how all the different procedures interact one with another. Thus to get a general idea, the procedures contained in these modules are categorised into four groups. The interaction between these types of procedure is illustrated in figure 6.

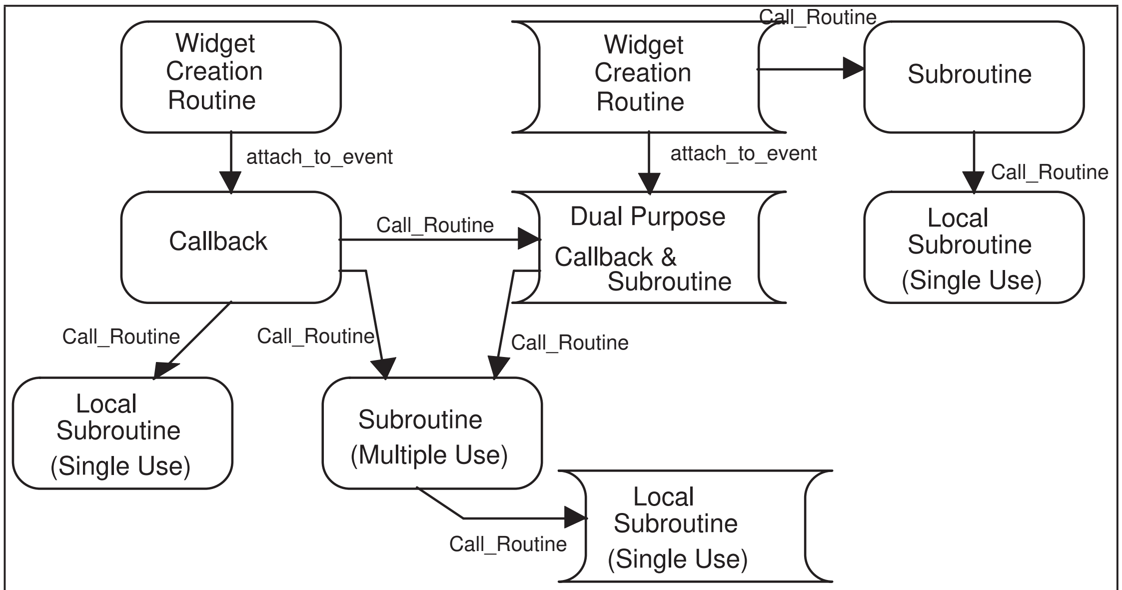


figure 6 - *Interaction between procedure categories*

1. Widget Creation Routines (WCR)

Each sound *studio* section is created by a widget creation routine. This is a procedure that performs one of three tasks. These tasks are,

- **Initialisation and configuration**. If the section does not yet exist, this task creates the section widgets, and attaches call-backs and variables to them. The widgets are always configured with the current contents of the ConfigSet array, when this task is called. It does not display the section.
- **Show the section**. This packs the section window into the root window of *studio*. It uses the contents of the PACK array to decide where its position in the packing order.
- **Remove the section**. This simply unpacks the section widgets from the root window.

Some WCRs divided their initialisation task into subroutines, particularly when numerous similar widgets are to be created, e.g. *MenuBar* and *Mixer*.

2. Call-backs

These are the top level of the sequential programs that are attached to widget events defined in the WCR. It is in these procedures that the actual processing of *Studio* lies.

3. Sub-routines

As is usual in sequential programs, often repeated tasks are assigned to subroutines. This category of procedures refers mainly to those subroutines that are used by more than one call-back.

4. Dual-purpose routines.

Dual purpose routines are a procedures that serve as a call-back and a subroutine. Examples of these are apparent in the edit module, where the Copy and Delete call--backs are used as a subroutine for the Cut call-back. Dual purpose routines usually have some parameter which switches the procedure from its call-back function to the subroutine function, since these can differ minutely.

5. Local subroutines.

> These subroutines are those which ideally only have scope in one single call-back or within a subroutine. However in practise it appears that procedures in TCL/Tk do not have scope like in C-programming; all procedures are global!

A list of all procedures, their purpose and category is found in table 4 in the listing index.

Techniques

Using TCL lists

TCL has an excellent facility for using lists. Thus in the case where many similar widgets were to be created, all it took was to create a list containing the features that are unique to each widget, and then loop using the `foreach` command and use `lindex` to access compound list items using command or variable substitution.

Here is a simple example from the procedure which sets up the effect bar section;

```
# Make the list
set effects {Filter Reverse Echo Vibrato}
# Loop through the list.
foreach Effect $effects {
        # Convert to lower case for widget path name.
        set effect [string tolower $Effect]
        # Create a checkbutton for the current list item
        checkbutton .effect.top.$effect -text $Effect
        pack .effect.top.$effect -side left \
                                        -padx 1
        #Bind button 3 to pop up the window to set the settings
        bind .effect.top.$effect <Button-3> "s$Effect"
}
```

Note that, in this case, the list is not complex and thus `lindex` was not used.

This technique was particularly useful in creating the menu bar and associated menus. Thus to modify a menu, it simply requires a list being changed. It also removed the tedium and enabled the automation of creating the mixer board sliders.

Keeping Track of States

Studio consists of many states. These many states are flagged in an array of variables titled FLAG. The number of states makes programming difficult, because there are so many `if` statements to keep track of. Of particular importance is ensuring that the user doesn't perform an action that isn't valid for a given state. For example, the user should not try to record whilst *Studio* is playing a sample. The sound card cannot do both at the same time!

The method used in *studio* to avoid this kind of conflict is by disabling the relevant widgets. The following are the techniques used, which ensure that the states of the relevant widgets change at the appropriate time.

Pop-up Windows

There are a number of pop-up window requiring the user to provide some information. When *Studio* is in this state, it is not desirable that another function is called, or even worse the same window is generated again, causing a proliferation of windows. To avoid this each dialogue window is created as a transient window and then requests the 'grab'. The grab is essentially an X flag which give all the focus of the application to this window. Thus in effect the parent window becomes disabled, until the pop-up window is destroyed. For example,

```
# Create toplevel window.
toplevel .echo -bg $ConfigSet(cBG1)
wm title .echo "Echo Settings"
# Set window as a pop-up child of the root window.
wm transient .echo .
# Stop entry from the main window. (grab).
catch { grab .echo }
```

Passive and Active State control

The other methods of state control actually change the state of specific widgets. This is somewhat messy since it requires the path name of the widget. This is particularly awkward because of the automated method in which many of the widgets are created. Fortunately there are only a few widgets that need to change state. These are,

- The control panel buttons (play, stop and record), depending on the play status.
- The file menu items, depending on whether a file is loaded.
- The edit menu items, depending on whether there is clipboard data and whether a section of the plot is marked.

In all these cases there is a procedure which changes the state of the corresponding widgets according to the status of the flags. The paths of these widgets are hard coded into these procedures. The way in which these procedures are called vary.

The procedure to change the control panel states is called by a code entry after the flag has been changed in the following *passive* manner:

```
set FLAG(Playing) 1
# Change Play Panel button states
PP_ButtonAble
```

In the latter cases, a more *active* method of doing this same task was discovered , using a variable trace. The `trace` command allows a procedure to be called every time a particular variable is written to. For example,

```
trace variable FLAG(Loaded) w FileMenuStates
```

This proves to be an efficient and effective way of tracking the states. The passive method is not really required and should be done away with in future modifications. A trace call-back provides information about the variable that has been accessed and in what way is has been accessed (read, write or kill). Thus a single state control mechanism may actually be written. Also for future work would be to over come the inflexibility of having to *hard code* the widgets' path names.

Generating Pipes

Studio makes use of a number of UNIX system commands, plus the use of Sox. The *grep, awk, dd*, *mv* and *cp* commands are all used. In many cases these are often use in pipes, just as they would be from the command line. Where this is of particular importance is in using *Sox* to add effects to the sample. In it's syntax *Sox* has only scope to add one effect per command call. Multiple effects may only be added by piping the sample data through a number of *Sox* calls. Due to TCl/Tk's simple variable substitution, it is possible to create a pipe dynamically so that only the selected effects are applied to the sample. A variable to contain the command string pipe is built up by adding the pipe section of each selected effect, and then appending the output file. Thus the constructed pipe string might be,

```
pipe = "sox -t .ub infile -t .ub - reverse | \
        sox -t .ub - outfile echo .5 .5"
```

This is then executed in the following manner,

```
eval exec $pipe
```

In addition, a small C program has been written to enable the fade-in and fade-out of samples. `fader.c`[1] has the following call syntax:

```
fader <channels> <sample_type> <fade_up_samples> <fade_down_samples>
```

where the sample type parameter may indicate signed or unsigned bytes or words according to the sox syntax. Fader imposes a liner fade up to a level of -60dB, then fades exponentially to the final level.

Configuration File

Sound Studio provides a mechanism which allows the user to set-up *studio* to his/her preference. This is through a configuration file that is set-up in the user's home directory. The main motivation for doing this was my dislike for the peach colour inherent to TCL/tk applications! Thus it seemed appropriate to create a mechanism allowing the user to change these colours and have these colour settings remain for subsequent running of *Studio*. Requesting a

[1] When compiling `fader.c` "by hand", be absolutely sure to use the compiler options listed in the install file. The program is forced to do type conversions between the sample types and double-precision floating point, its own internal working type. It is very important that, in particular, `char` types are *signed* be default. Use the following command for compilation with `gcc`:

```
gcc -O2 -fsigned-char fader.c -o fader -lm
```

"make default" menu entry has the effect of updating this file, provifing a user-friendly way of selecting and saving default settings.

The idea is simple:

- All the WCRs in *studio* use the contents of the ConfigSet array to configure the widgets.
- A dialogue box to allow the user to change the contents of this array is provided.
- The creation and configuration tasks of WCR are kept distinct from each other, so that widgets may be re-configured whilst *Studio* is running.
- The contents of the array may be stored in a configuration file upon user request.

This idea was extended to allow the user to store the view options, i.e. what sections of *Studio* are to be displayed and also the mixer sliders to be contained in the Mixer Board. However these options already had a place in the FLAG and MIXER array. This complicated the saving procedure slightly since now there are the contents of three arrays to be saved and it must be possible to save these independently of the others into the configuration file.

The general mechanism for doing this is illustrated in figure 7. As illustrated the configuration file is essentially a table with three columns. The index and description correspond to the ConfigInfo array. The first letter of these indices indicates which array the data belongs to. The characters after the first letter represent the index of the array to which the data belongs. This is all sorted out in the procedures `LoadSettings` and `SaveSettings` in the options module.

The description is provided to provide useful text for dialogue boxes[2] and also to make the configuration file readable.

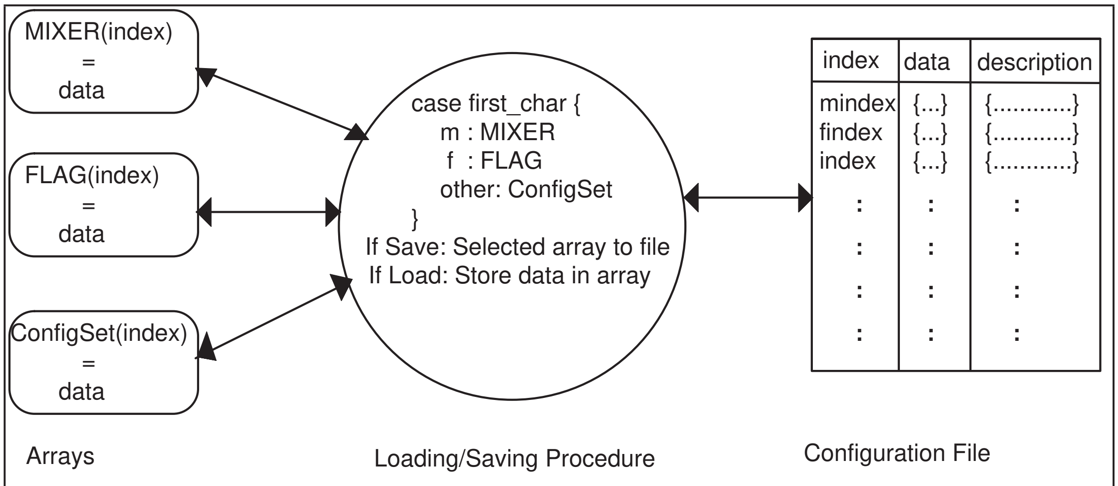


figure 7 - Configuration file Mechanism

It is hoped that this mechanism is sufficient for future extensions to the preference options. Unfortunately his *simple* job was a bigger task than anticipated and was therefore rushed!

[2] See the *colour* dialog box in the SnapShots section.

Icons

The icons used in *Studio* were generated by the X application called *bitmap.* There are two sizes of bitmap used in *Studio*.

- 64x64, for the iconified version of *Studio* and the *User Manual*.
- 16x16, for the button icons in the control panel.
- 32x32, for the magnification zoom buttons.

Each of these bitmaps are stored in files with the extension .ico, and are read into *Studio* using the *@filename* method. A way to combined all these bitmaps into one file was not found.

The Playline and VU Meters

As of version 1.0, Studio supports a playline which gives visual feedback as to the approximate play position on the plot panel, and VU meters with peak hold to assist in setting recording level and assessing playback volume.

Since TCL is merely a scripting language, it became clear very early on that it would not be feasible to implement either the playline or the VU meters in TCL alone, especially when dealing with the real time issues that both would raise. The most feasible solution was to have a C application do all the ‘‘number crunching’’ and pass information to studio which would then be used directly to produce the desired results. It seemed logical to modify the studio_tool application to not only play or record the required soundfile, but also to do the necessary working and send the required data back to studio.

This necessitated a change to the data flow, as previously studio only sent data out to the studio_tool application by actually executing it, with the command line parameters set within studio, and then possibly setting an internal variable equal to the output of studio_tool after it had finished executing. In this way, there was no communication between studio and studio_tool, as studio_tool was actually running. To implement both the playline and the VU meters a communication channel was needed to be established between studio and studio_tool. The responsible code reads as follows:

```
# execute studio_tool and open a channel
set pipe [open "|./$FILE(Tool) play $source $LOAD(bit) $LOAD(chnl)\
                            $LOAD(rate) meter 75 1 $linelist" r]
# Call function Reader whenever new data comes in the channel
fileevent $pipe readable [list Reader $pipe]
```

The fileevent command forces *Studio* into an event loop, calling the function `Reader` every time a new piece of data came in on the channel. `Reader` then reads the data from the channel, checks whether it is a VU update request or a play-line motion request, and performes the required task.

Preserving the functionality of studio_tool

studio_tool serves not only as the mechanism by which studio interfaces with the sound card, but also as a useful stand-alone program that could play and record sound files from the command line prompt. It was therefore required that *studio_tool* would only emit playline and VU data only if requested to do so in the command line arguments. In this way, *studio_tool* could still be used from the command line prompt, without flooding the screen with unnecessary data. For example:

```
% studio_tool play ultravibe.au 16 1 11025
```

will play the soundfile with the specified rate, precision and channels. This would be how studio_tool would be used from the command line prompt., whereas:

```
% studio_tool play ultravibe.au 16 11025 meter 20 1 line 499 109779
```

will play the soundfile, but also emit the VU meter and play-line data. The extra parameters in the command line were designed only to be used by *studio*, as their calculation is probably too cumbersome to be expected of a user.

Dealing with the Real Time Issues

Any program that deals with performing timing on audio data faces real time issues, which must be addressed. For both the playline and the VU meter, the program needs to know exactly how much of the soundfile had been processed at all times. Both Solaris and Linux had very convenient ioctl calls that returned this information. In Solaris the call is:

```
ioctl(fd,AUDIO_GETDEV,&audio_info)
```

which fills the structure `audio_info` with the current soundcard information. One of the fields of `audio_info` is `play.samples` which stores the number of samples that the soundcard had processed.

The ioctl command in Linux is:

```
ioctl(fd,SNDCTL_DSP_GETOPTR,&info)
```

which fills the structure `info` with soundcard information. One of the fields of `info` is bytes which stored the number of bytes that the soundcard had processed. Using the channels and precision to work out the sample size (in bytes), this value is easily converted to the number of samples processed.

In Solaris, the number of samples processed was not read from the soundcard, but from the pseudo audio-device, `/dev/audioctl`:

```
if (PLATFORM == OS_SUNOS)
{
      if ((fd2=open("/dev/audioctl", O_RDONLY,0))== -1)
      {
            printf("Read Failure\n");
            exit(-1);
      }
}
```

This is a control device for the sound hardware that can be used to check or set the status of the soundcard, but read() or write() system calls to it are ignored. The pseudo-device returns the number of samples in blocks of 8 samples, whereas using the soundcard the blocksize is 512 samples. Using the Pseudo device, *studio_tool* is able to track the number of samples processed more accurately.

In Linux, though, an extra few lines of code were required in order to set the fragment size on the soundcard buffer to 64 samples:

```
if (PLATFORM == OS_LINUX)
    {
        if (samplesize == 1) fragsize = 0x7fff0006;
        if (samplesize == 2) fragsize = 0x7fff0007;
        if (samplesize == 4) fragsize = 0x7fff0008;
        if (samplesize == 8) fragsize = 0x7fff0009;
        if (samplesize == 16) fragsize = 0x7fff0010;
        ioctl(fd, SNDCTL_DSP_SETFRAGMENT, &fragsize);
    }
```

Knowing the number of samples processed at all times proved to be vital in solving the real-time issues, as the soundcard itself (or its pseudo control device) is used to regulate the flow of output data.

Separate Processes Vs POSIX Threads

The play function in the previous version of *studio_tool* only performed one function: it played the soundfile. To generate the necessary output, the play function had to be modified to perform multiple tasks. These were (in execute-initiation order):

1. To play the soundfile
2. To calculate and output the play line data
3. To calculate and output the VU data
4. To calculate and output a regular decay on the VU signal

Obviously, these multiple processes would need to be run concurrently within the play function in order for the program to work as desired. This posed the question of whether to use separate processes or POSIX threads. At first it seemed that threads would have been the best solution, as the VU meter uses two separate processes that would be required to share certain variables. They are also much more lightweight, since a whole new process is not created, therefore threads do not require to be killed off, they are automatically killed when the main process terminates.

Unfortunately, the use of threads did not allocate processor time fairly to each of the processes; the busier tasks would "hog" the processor, and this would be wholly unsuitable for processes with real time issues. Subsequent improvements of the control algorithms prior to version 1.0 have enormously reducted the computational overhead by the introduction of appropriate sleep()s within the poll loops, so it is likely that this reason has been removed. However, these improvements were achieved only after the structure of the new *studio_tool* had matured.

Forking Hell

Separate processes are used to perform the multiple tasking. This raises two issues that need to be resolved. Firstly when a `fork()` system call was performed, the child takes a copy of all the parent's variables, but that is exactly what they are: copies. They are not shared. Changes made to a certain variable in the child do not appear in the same variable in the parent. To enable processes to communicate it is necessary to set-up as a shared structure. This is achieved by the following code:

```
/*  get unique key for shared memory block  */
if ((VUmemid = shmget (IPC_PRIVATE,
                                        sizeof(SharedVUVariables),0660)) == -1) {
        fprintf(stderr,"Shared Memory Failure");
        exit (-1);
}
  /* attach the variable VUdata to shared memory block */
VUdata  = (char *)shmat (VUmemid,(char *)0,0);
/* setup shared memory block as transient */
shmctl (VUmemid,IPC_RMID,NULL);
```

The `shmget()` call sets up a unique key (`VUmemid`) for a block of shared memory, which was attached to the variable VUdata by the `shmat()` call. Finally, the `shmctl()` call sets up the shared memory block as transient, which is used to prevent memory leaks.

The second issue is that of terminating the processes. When *studio_tool* is used by *studio*, it could be terminated in two ways. Firstly, it terminates when the soundfile had finished, but it could also be terminated by executing a system kill() command within studio (which happens when the stop button is pressed). This action actually sent *studio_tool* a SIGTERM signal, so *studio_tool* needs to be able to catch this signal and kill off all the child processes before quitting. This is achieved by the use of a `signal()` call directly after each `fork()` call, which nominates a quitting function to be executed when the signal SIGTERM was received by each process:

```
signal(SIGTERM,sigquit);
```

Also, the last command of each process is a kill() command, by which the process sent itself a SIGTERM signal on its natural completion:

```
kill(pid_number,SIGTERM);
```

The play function uses four `fork()` calls to execute each of its four tasks. The processes were dealt with in a last come-first served manner, so the code for the first process to be dealt with, which was the actual playing of the sound file, appears in the child of the last fork().

Working Out the Data for the Playline

After loading a sound file into studio, it displays a graphical representation of the sound file in the PLOT canvas. The basic function of the playline is to be a visual indicator of how much of the sound file has been played, as it is being played. In operation this is simply a vertical line that moves from left to right in the PLOT canvas at a speed that ensures that the location of the line is at all time representative of the current position in the sound file.

In order to calculate the required data for the play line, *studio_tool* needs to be passed two extra pieces of data by *studio*. These are the number of pixels in the PLOT canvas, and the total number of samples in the soundfile to be played.

The code which emits the output data for the playline is in the third fork(), so that it was dealt with directly after the soundfile was played. The output from this process is "l.n" (a letter l, a dot, and a number corresponding to the number of display pixels that the play-line should be moved). The play status is polled roughtly every 15ms, when the current audio sample number is read from the sound device.

Data required to Simulate the VU meter

A VU meter displays the average volume level of an audio signal. Its reading corresponds closely to the level sensing mechanism of the human ear, and so provides a very useful indication of the subjective loudness of an audio signal.

A VU meter has two attributes which determine its behaviour, the sample attack time, and its decay rate. In the real world a VU meter is a peak meter with very slow ballistics, therefore it is not very good at tracking fast changes in audio peak levels, as the meter is forced to "sample" the signal over a longer time period than a peak meter. This means that the needle is less active than that of a peak meter and in this way an average of the audio signal levels if displayed.

In order to calculate the data required to simulate a VU meter, *studio_tool* required two extra pieces of data, namely the sample attack time, and the decay rate, which were passed to it by *studio* as command line arguments. From the meter attack time, and the rate of the sound file the number of samples per attack period are calculated.

Since there are two aspects of simulating the VU meter, namely the actual amplitude output and the meter decay, two separate processes were instantiated. The first process is quite similar to the playline process in that, while the value of outputnow was zero, it checks the soundcard or control device on playback or the recorded file size on record.

The amplitude data in value, was converted to decibels by taking the $\log_{10}$ of the normalised amplitude and multiplying by 20. The result is stored if it exceeds the current VU reading and is emitted to `stdout` in the format "v.n". Sumultaneously, every time a VU decay period elapses, the decay process awakens and reduces the stored VU value by an appropriate amount.

Changes to the tk Script

Reading from the channel

When playing or recording, the version < 1.0 of *studio* interfaced with the soundcard by executing *studio_tool* from within its script. This command is replaced by an open command, which executes *studio_tool*, and set up a pipe and a fileevent command, which ran the TCL procedure `Reader` every time a new piece of data came in on the channel.

The function of the `Reader` procedure was to read a line of data from *studio_tool*’s output, parse it to decide what the purpose of the data was, and to use that data to perform the required task.

Moving the Play Line

The data that studio_tool passed to Studio relating to the play line was a series of numbers. These represented the number of pixels that the line needed to be moved in the PLOT canvas. The implementation of the line was very simple, as Tk has the ability to create a line (or any shape) in a canvas and move it with one command:

```
.canvasname move itemid x-pixels y-pixels
```

This means that Studio needs to create a line during its the PLOT canvas initialisation, and set a global variable equal to the line-id thus:set LINE(Id)

```
 [.plotbox.cs.canvas create line 0 0 0 50 -fill red]
```

Two counters of the number of pixels moved were also maintained. One may be used in resetting the line after playing has finished. The other is used to track the line at higher magnifications, by switching to the next portion of the PLOT canvas when the line approached the right hand side of the PLOT window.

Simulating the VU Meter

This is achieved with the help of two procedures. `SetupVU_Display` {}, creates the VU meter toplevel window, with the required number of channels, and `light`{} displays the required number of lit LEDs.

The LED lights were defined as movable objects, in the same way as playline, except they were created OFF the canvas, and moved on to the canvas as required.

Sox Version 15 upgrades

Implementing the support for the new version of *Sox* simply involved Studio executing "sox --h" during the initial load-up. *Sox* version 12.15 returns "SOX version 12.15" in the first line of the output, which Studio is able to catch and determine whether of not *Sox* was version 12.15 or not. If so, the global flag SOXVersion15 is set to 1, otherwise it was set to 0.

If SOXVersion15 was set to 1, Studio adds the extra-effect selectors to the Effects bar, and also to the Effects menu. The echo effect, which was already present in the older versions of *Sox*, are bound to a different callback function, due to the echo effect argumants being changed in Sox-12.15. Extra callback functions were also needed for each of the other new effects, which were almost identical to the callback functions for the older *Sox* effects.

When using the new 12.15 effects, *Sox* would possibly produce warning messages, indicating that the output soundfile had clipped in parts. Studio caught the output from the use of *Sox*, and produced a dialog box, that contained a text widget, which displayed these warnings. The global variable dh was set to ''don't know'' and a vwait command was used to halt everything until this variable had been changed.

Known BUGS

Format Conversion

Studio is not yet fully compatible with all *Sox* format conversions. This applies mainly to converting the raw data into the sample format requested of the user. There are a number of conversion that fail. These are,

- cdr. This needs the sampling parameters of cd.
- dat. This is very time consuming and fails.
- hcom. Requires sampling rates of, 5512,7350,11025, 20500 only.
- sf. Saves data successfully but format doesn't.
- smp. Saves incorrectly.

Another bug, or even maybe the same bug, is due to the way that *Studio* uses *Sox* to down-convert the loaded sample, if it's sampling parameters exceeds that of the sound card. This is to ensure that playback is possible. However, this proves a problem when loading a 2-channel sample on a 1-channel set--up. The conversion into *Studio* is fine. The sample, however, cannot be saved back into the 2-channel form, this is because the individual channel data is lost through averaging. This is really a problem of having this conversion in the wrong place. The averaging and down-conversion should really happen upon play-back and not upon loading. The flag, FLAG(Converted), is already in place to make this possible.

Studio does not provide any error detection of *Sox* errors, they are simply ignored. The reason for this is that *Sox* provides information through the standard error file. This information is not always error information. Thus handling the exception using `catch` is awkward. Currently the only compromise is to print the error information to standard out, until all the errors are ironed out.

From this it is clear that a review of the whole loading and saving mechanism is required.

Zoom

Currently *Studio* reverts to the root zoom level if an editing operation has been performed or effects added. This is because of the *FillPlot* call that is performed at the end of all these operations. This call is required to update the contents of the plot as well as recalculating the sample size. It is an awkward side effect that can be annoying to the user. It may be possible to overcome this problem at least in part if the zoom context where used in the *FillPlot* operation.

Playline in preview-mode

The playline does not traverse the audio plot in preview mode. This is simply because the variety of effects which Studio can accommodate, via *Sox* and otherwise, has too great an influence on the behaviour of the playline to make a general solution feasilble. For example, *Sox* can change the "tempo" of a file, or even cause it to be played backwards! Because the sound-processing program is not tightly bound to studio development, it was felt the best course was to suppress the playline appearance. This is really more of a limitation than a bug.

SOUND BLASTER 8

In testing, an 8-bit Sound Blaster card would record successfully at all sampling rates, but would only play-back, correctly, recordings sampled with a sampling rate of 12kHz or less. This appears to be a driver problem. The driver is set correctly to the correct sampling rate and responds that is set to the sampling rate, however it just doesn't playback correctly. It cannot be a sound card problem since recording and playback is successful at all available sampling rates in DOS sound sample applications.

Tests on a 16-bit Sound Blaster did not present this problem.

VoxWare Manual Error

In programming the mixer interface it was found that it is not possible to access the mixer though the dsp device as explained in the VoxWare Hacker's guide[3]. Hours of frustration resulted because of this error. The mixer functions seem only available by ioctl calls to the open mixer device.

---

[3] The VoxWare manual states (p.12): "The mixer ioctl calls are accessible through every device of the VoxWare driver."

Playline does not move smoothly (Linux)

studio_tool relies on an ioctrl() call to the sound drivers to obtain information about the current output sample. This is unavailable on some hardware, and under these circumstatnces the kernel falls back to making a "best guess" based on the number of buffer fragments written to the card so far. Unfortuately, the best guess is not always good enough to ensure the playline moves smoothly, especially on small files.

Playline stalls on moving the Mixer Controls (Solaris)

The hardware on which the program has been tested (Ultra workstations) supports high resolution sample counts (see above), but any access to the audio device resets the counter. This invalidates the current-play-position variables within *Studio* and causes the playline to malfunction for the rest of the playback period. Since Solaris is not our primary development platform, this bug is unlikely to be fixed in the near future.

Listing INDEX

As a further guide into the source code, the following lists provide information about the global variables and procedures.

global Variable Index

Table 3 shows the global variables used in *Studio* and their purpose.

Global Variable	Type	Description
CARD(Bits)	LIST	Bit resolutions available from sound card
CARD(Chnnls)	LIST	Channels (mono, stereo) available from sound card.
CARD(Driver)	CONST	VoxWare driver version.
CARD(HW***n***)	CONST	Name of sound card hardware detected. ***n***=0,1,2,....
CARD(Maxrates)	LIST	Maximum sampling rates of sound card, for each channel setting.
CARD(Minrates)	LIST	Maximum sampling rates of sound card, for each channel setting.
ConfigInfo(BDW)	CONST	Description of configuration option: Borderwidth
ConfigInfo(FNT_L)	CONST	Description of configuration option: LargeFont
ConfigInfo(FNT_S)	CONST	Description of configuration option: SmallFont

ConfigInfo(REL)	CONST	Description of configuration option: Relief for Sections.
ConfigInfo(cABG)	CONST	Description of configuration option: ActiveBack-Ground
ConfigInfo(cAFG)	CONST	Description of configuration option: ActiveFor-eGround
ConfigInfo(cBG1)	CONST	Description of configuration option: BackGround1
ConfigInfo(cBG2)	CONST	Description of configuration option: BackGround2
ConfigInfo(cBG3)	CONST	Description of configuration option: BackGround3
ConfigInfo(cDFG)	CONST	Description of configuration option: Disabled-ForeGround
ConfigInfo(cFG1)	CONST	Description of configuration option: ForeGround1
ConfigInfo(cFG2)	CONST	Description of configuration option: ForeGround2
ConfigInfo(cFG3)	CONST	Description of configuration option: ForeGround3
ConfigInfo(fVw_***Section_WCR***)	CONST	Description of configuration option: View Flag for a section. ***Section_WCR***=EffectBar,MenuBar,PlayPanel,-Mixer, InfoBar, PlotBox
ConfigInfo(mnoshow)	CONST	Description of configuration option: List of mixer devices not to be shown. [see MIXER(noshow)]
ConfigSet(BDW)	CONST	Value of configuration option: Borderwidth
ConfigSet(FNT_L)	CONST	Value of configuration option: LargeFont
ConfigSet(FNT_S)	CONST	Value of configuration option: SmallFont
ConfigSet(REL)	CONST	Value of configuration option: Relief for Sections.
ConfigSet(cABG)	CONST	Value of configuration option: ActiveBackGround
ConfigSet(cAFG)	CONST	Value of configuration option: ActiveForeGround
ConfigSet(cBG1)	CONST	Value of configuration option: BackGround1
ConfigSet(cBG2)	CONST	Value of configuration option: BackGround2
ConfigSet(cBG3)	CONST	Value of configuration option: BackGround3
ConfigSet(cDFG)	CONST	Value of configuration option: DisabledFore-Ground
ConfigSet(cFG1)	CONST	Value of configuration option: ForeGround1
ConfigSet(cFG2)	CONST	Value of configuration option: ForeGround2
ConfigSet(cFG3)	CONST	Value of configuration option: ForeGround3
DialogCfg	LIST	Configuration list for *dialog* and *filebrowser* calls.
FILE(CFG)	CONST	Path+Filename for the configuration file.
FILE(CURR)	CONST	Path+Filename of the current file.
FILE(Fader)	CONST	Path+Filename of the `fader` binary
FILE(Mixer)	CONST	Path+Filename of the mixer interface (*studio_mixer*)

FILE(PID)	CONST	The unique part of temporary file names. (*System clock*)
FILE(SRC)	CONST	Path of the modules of *studio*.
FILE(Sox)	CONST	Path+Filename of *Sox*.
FILE(TMP1)	CONST	Path+Filename of temporary file 1
FILE(TMP2)	CONST	Path+Filename of temporary file 2
FILE(TMP3)	CONST	Path+Filename of temporary file 3
FILE(TMP4)	CONST	Path+Filename of temporary file 4
FILE(TMP5)	CONST	Path+Filename of temporary file 5
FILE(Tool)	CONST	Path+Filename of the sound card interface. (*studio_tool*)
FLAG(Card_Exist)	BOOL	Flag indicating a sound card exists.
FLAG(Clip)	BOOL	Flag indicating the clipboard has contents.
FLAG(Converted)	BOOL	Flag indicating the sample has been converted for sound card compatibility.
FLAG(Edited)	BOOL	Flag indicating the working file has been edited.
FLAG(Loaded)	BOOL	Flag indicating a working file exists.
FLAG(Marked)	BOOL	Flag indicating a marked area exists in the editing plot.
FLAG(Mixer_Exist)	BOOL	Flag indicating a sound card mixer is available.
FLAG(Playing)	BOOL	Flag indicating playback/recording in progress.
FLAG(Recording)	BOOL	Flag indicating the application is recording.
FLAG(Vw_***Section_WCR***)	BOOL	Flag indicating the ***section*** is visible. ***Section_WCR***=EffectBar,MenuBar,PlayPanel,-Mixer, InfoBar, PlotBox.
FX(ECHDelay)	VAR	Slider value of the echo parameter: Volume.
FX(ECHVolume)	VAR	Slider value of the echo parameter: Volume.
FX(FLTCentre)	VAR	Slider value of the filter parameter: Centre Frequency.
FX(FLTtype)	VAR	Slider value of the filter parameter: Filter Type
FX(Preview)	BOOL	Flag indicating preview mode.
FX(VIBDepth)	VAR	Slider value of the vibrato parameter: depth.
FX(VIBSpeed)	VAR	Slider value of the vibrato parameter: speed.
FX(***effect_name***)	BOOL	Flag indicating that ***effect*** is to be applied. ***effect_name***=echo, filter, reverse, tempo, vibrato, volume.
FX(fadein)	VAR	fade-in time for the fade effect in centiseconds
FX(fadeout)	VAR	Fade-out time in centiseconds
FX(list)	LIST	List of available effects.
FX(tempoVal)	VAR	Slider value for the tempo effect.
FX(volumeVal)	VAR	Slider value for the volume effect..

HELP(AFG)	CONST	ActiveForeGround colour for the Help window.
HELP(BG)	CONST	BackGround colour for the Help window.
HELP(COM)	CONST	ForeGround colour of contents text.
HELP(FG)	CONST	ForeGround colour for the Help window.
HELP(HIL)	CONST	ForeGround colour of highlighted text.
HELP(Log)	VLIST	Stores the position in the Help widget that have been viewed via. Contents.
HELP(UserGuide)	CONST	Path+Filename of the user guide text.
INFO(Length)	STR	InfoBar string: Length of current sample.
INFO(Rate)	STR	InfoBar string: Sample Rate of current sample.
INFO(Sample)	STR	InfoBar string: File name of current sample.
INFO(Stereo)	STR	InfoBar string: Channels in current sample.
INFO(Type)	STR	InfoBar string: Bit resolution of current sample.
LINE(Id)	VAR	Play-line handle for use in graphics operations.
LINE(On)	BOOL	Whether the play-line is visible.
LINE(PixelsMovedInPlot)	VAR	Number of pixels traversed by the playline.
LINE(Startx)	VAR	Initial position of play-line at commencement of *studio_tool*
LINE(TotalPoxelsMoved)	VAR	Pixels moved by play-line
LOAD(bit)	VAR	Sampling parameter: Bit Resolution.
LOAD(chnl)	VAR	Sampling parameter: Channels.
LOAD(format)	VAR	Sampling parameter: Format name.
LOAD(maxvol)	VAR	Sampling parameter: *Sox* maximum volume index.
LOAD(rate)	VAR	Sampling parameter: Sampling Rate
LOAD(size)	VAR	Sampling parameter: a letter indicating the size of each sample, e.g. b=byte (see *Sox* documentation)
LOAD(style)	VAR	Sampling parameter: a letter indicating the data style of each sample, e.g. u=unsigned (see *Sox* documentation)
MACHINE(PLATFORM)	CONST	The type of host operating system under which *studio* is running. Currenly valid strings are `Linux` and `SunOS`
MIXER(All)	LIST	Index numbers of all available Mixer devices.
MIXER(I_chnnls)	LIST	List of input channels to be shown in Mixer board.
MIXER(O_chnnls)	LIST	List of input channels to be shown in Mixer board.
MIXER(flag***DevName***)	BOOL	The flag indicating a recording source is active. ***DevName***=an element of MIXER(labels).
MIXER(label***DevName****n*)		Slider value of a mixer device channel. ***DevName***=an element of MIXER(labels). *n*=0,1.

MIXER(labels)	LIST	List of mixer device names, returned from mixer interface.
MIXER(noshow)	LIST	List of mixer devices not to be shown.
MIXER(show***DevName***)	BOOL	Flag indicating a device has a slider in the Mixer Board. ***DevName***=an element of MIXER(labels).
PACK(***Section_WCR***)	CONST	Path of the section after which ***Section*** should be packed. ***Section_WCR***=EffectBar,MenuBar,PlayPanel,-Mixer, InfoBar, PlotBox
PACK(list)	LIST	List of section_WCRs, for generating PACK(***Section_WCR***) globals.
PACK(path)	LIST	List of section paths, for generating PACK(***Section_WCR***) globals.
PID	VLIST	The process id's of background process pipes.
PLAY(ButtList)	LIST	List of PlayPanel button names.
PLAY(Time)	VAR	The value of the *Play Time* field, in the control panel.
PLAY(TimeMark)	VAR	The value of the system clock at the time the PP_Playing routine was called during playback, in order to change PLAY(Time) every second.
PLOT(Blocksize)	VAR	The number of bytes represented by a single vertical line of the editing plot.
PLOT(End)	STR	The position of insertion point 2: in a time format for display.
PLOT(Endx)	VAR	The position of insertion point 2: it's x-coord in the canvas.
PLOT(Height)	CONST	Height of plot canvas.
PLOT(Magnification)	VAR	Stores the current magnification factor of the plot canvas. Currently restricted to 1, 2, 4 or 8.
PLOT(MarkBox)	CONST	Handle to the marking rectangle in the plot canvas.
PLOT(MaxBlockSize)	CONST	The block size which fits the currents sample entirely within the canvas boundaries.
PLOT(Offset)	VAR	Number of bytes in sample preceding the point at the x=0 point in the plot canvas.
PLOT(Start)	STR	The position of insertion point 1: in a time format for display.
PLOT(Startx)	VAR	The position of insertion point 2: it's x-coord in the canvas.
PLOT(Total)	VAR	The actual number of vertical lines drawn in the plot canvas.
PLOT(Width)	VAR	Width of plot canvas. May be greater the the visible width if the magnification feature is in use.
PLOT(ZoomCount)	VAR	Number of Zoom levels
PLOT(Zoom***n***)	VLIST	Context of the last zoom level. ***n***=0,1,2,3...

SAMPLE(Size)	VAR	Number of samples in the sample file.
SAMPLE(bit)	VAR	Sampling parameter: Bit Resolution.
SAMPLE(chnl)	VAR	Sampling parameter: Channels.
SAMPLE(format)	VAR	Sampling parameter: Format name.
SAMPLE(rate)	VAR	Sampling parameter: Sampling Rate
SAMPLE(size)	VAR	Sampling parameter: a letter indicating the size of each sample, e.g. b=byte (see *Sox* documentation)
SAMPLE(style)	VAR	Sampling parameter: a letter indicating the data style of each sample, e.g. u=unsigned (see *Sox* documentation)
VU(On)	BOOL	Whether the peak meter is enabled
VU(Peak***channel***)	VAR	Current peak value for channel ***channel***
VU(PeakOn)	BOOL	Whether the peak hold feature is visible
VU(channels)	VAR	The current number of channels being displayed by the VU meter (if VU(On))
VU(xpos)	VAR	X position of the VU meters' window
VU(ypos)	VAR	Y position of the VU meters' window
button	VAR	A global of the *dialogue* subroutine.

table 3 *- Index of Global Variables*

Variable type abbreviations: CONST=Constant. VAR=Variable. STR=Formatted string for display. LIST=a TCL list; VLIST=a dynamic list; BOOL=a flag.

Procedure Index

The index in table 4 is to be used as a guide to finding procedures within the source code. The category is provided to give an indication of its position in the procedure hierarchy.

Procedure name	Category	Module file name
About	CB	help.tk
AddFX	SUB	effect.tk
AddSilence	CB	file.tk
Bit2SoxSize	LOC: Load	file.tk
CARDSetter	SUB	misc.tk
CheckGlob	LOC:filebrowser	misc.tk
ColourList	SUB	option.tk
Colours	CB	option.tk
Convert	CB	file.tk
Copy	CB/SUB	edit.tk

Category abbreviations:

WCR	Widget Creation Routines
CB	Callback
CB/SUB	dual purpose routines
SUB	Subroutine
LOC	Local Subroutine

Entries applicatble only to non-Linux implementations are italicised.

CreateMenu	SUB	menu.tk
CreateWidgets	SUB	menu.tk
Cut	CB	edit.tk
Delete	CB/SUB	edit.tk
DisplayWarnings	WCR	effects.tk
EditMenuStates	TCB	edit.tk
EffectBar	WCR	effect.tk
EndCallBack	LOC:Select	edit.tk
Exit	CB	file.tk
FB_Cancel	LOC:FileBrowser/2	misc.tk
FB_Exit	LOC:FileBrowser/2	misc.tk
FB_RefreshList	LOC:FileBrowser/2	misc.tk
FB_Select	LOC:FileBrowser/2	misc.tk
FXApply	CB	effect.tk
FileBrowser	SUB	misc.tk
FileBrowser2	SUB	misc.tk
FileMenuStates	TCB	file.tk
FillPlot	SUB	plot.tk
GetArrayName	SUB	option.tk
GetSoxVersion	SUB	misc.tk
GetWordSize	SUB	edit.tk
HoldWindow	SUB	misc.tk
Info	CB	file.tk
Info2Set	SUB	info.tk
InfoBar	WCR	info.tk
InfoFill	SUB	info.tk
Insert	CB	edit.tk
light	LOC:Reader	lineVU.tk
ListBoxFill	SUB	option.tk
Load	CB/SUB	file.tk
LoadSettings	SUB	option.tk
MIX_DeviceGet	SUB	mixer.tk
MIX_Reset	SUB	mixer.tk
MarkPlot	SUB	plot.tk
MatchList	LOC:Setter	option.tk
MenuBar	WCR	menu.tk
MenuGetUndl	LOC	menu.tk
MenuItemDisable	SUB	menu.tk

Mixer	WCR	mixer.tk
MixerSliders	SUB	mixer.tk
MkSlider	SUB	mixer.tk
New	CB/SUB	file.tk
OrderPacking	SUB	misc.tk
PP_ButtonAble	SUB	playpanel.tk
PP_Recording	SUB	playpanel.tk
Paste	CB	edit.tk
Peak_Reset	BC	lineVU.tk
Play	CB	playpanel.tk
PlayPanel	WCR	playpanel.tk
PlayPort	*CB*	*SUNmixer.tk*
PlotBox	WCR	plot.tk
Preview	LOC	option.tk
PutPeak	SUB	lineVU.tk
Reader	SUB	lineVU.tk
RecSrcSet	SUB	mixer.tk
Record	CB	playpanel.tk
RecPort	*CB*	*SUNmixer.tk*
Resetter	SUB	option.tk
SETscaleLim	LOC:Settings	option.tk
sChorus	CB	neweffects.tk
sEcho15	CB	neweffects.tk
sFlanger	CB	neweffects.tk
sPhaser	CB	neweffects.tk
sReverb	CB	neweffects.tk
SetVUParamters	CB	lineVU.tk
SetupVU_Channels	WCR	lineVU.tk
Save	CB	file.tk
SaveSettings	SUB	option.tk
Saver	SUB	file.tk
Sec2Time	SUB	misc.tk
Select	CB	edit.tk
SelectApply	LOC:Select	edit.tk
SetInitially	LOC:SaveSettings	option.tk
Setter	SUB	option.tk
Settings	CB	option.tk
Show_VY	CB	option.tk

StartCallBack	LOC:Select	edit.tk
Stop	CB	playpanel.tk
SoxWarnings	SUB	effects.tk
SUN_GetMixInfo	*SUB*	*SUNmixer.tk*
SUN_Mixer	*CB*	*SUNmixer.tk*
SUN_SetSliders	*SUB*	*SYBmixer.tk*
SUNPlayBalanceS-etter	*CB*	*SUNmixer.tk*
SUNRecBalanceSet-ter	*CB*	*SUNmixer.tk*
SUNRecSetter	*CB*	*SUNmixer.tk*
SUNSetter	*CB*	*SUNmixer.tk*
Trim	CB/SUB	edit.tk
Trim_Stop	CB	playpanel.tk
Undo	CB	edit.tk
UserManual	CB	help.tk
VolSet	SUB	mixer.tk
WaitPlease	SUB	misc.tk
Zoom	CB	edit.tk
bindScale	SUB	misc.tk
dialog	SUB	misc.tk
expandIt	LOC:CreateMenu	menu.tk
forAllMatches	SUB	help.tk
formatText	SUB	help.tk
loadFile	SUB	help.tk
parse_sox	LOC	file.tk
sEcho	CB	effect.tk
sFade	CB	effect.tk
sFilter	CB	effect.tk
sReverse	CB	effect.tk
sVibrato	CB	effect.tk
sVolTemp	CB	effect.tk
scrollcb	LOC:plotbox	plot.tk
setPlotMag	CB/SUB	plot.tk
setRanges	LOC:sFilter	effect.tk
setupText	SUB	help.tk
vEffects	CB	option.tk
vInfo	CB	option.tk
vMixer	CB	option.tk

vPlot	CB	option.tk

table 4 *- List of studio procedures*