I. Overview

A. Sound Objects

Patch

The patch (cPatch) is the low-level object. It's just a sample with some additional information, such as
- whether it loops
- whether it is streamed
- whether it is to be preloaded
- resource ID number

Channel

The channel has been removed from the design. We're using a GZSnd instead. See *Modifcations to GZSndSys*.

Track

A track is just a sequence of instructions. For example, a simple track might contain instructions to
- wait
- play a sample
- wait
- play another sample
- end

Track Player

The track player executes the instructions, and provides an interface.

Separating the track player from the actual track provides independent control of several copies of the same track.

HitMan

HitMan manages the hit system. It
- handles initialization
- loads of HIT and .ini files
- allocates sound objects
- manages the timer

B. Variables and attribute registers

A variable is a "thing that contains a value." An attribute is a variable that is part of the definition of a sound object. Both of these are refered to as "registers"; "variable registers" and "attribute registers". Registers are grouped into "register sets".

Register sets:
Sound object attributes (used in track and patch definitions)
Sound object variables (channel and track player, run-time)
Track attributes (track definition)
Track player attributes (run-time)
Track variables
Track defaults
Patch defaults

Not all of the registers are copied at run-time – only those not set to their defaults.

Thus, there may be several copies of the same variable. The channel has default values, the patch has values that overwrite the channel defaults, and then the track has sound object values of its own overwrite the patch and channel defaults.

Each attribute has a special value that means "default". This tells the track player not to copy use the value. Having a special value for default lets the composer specify that they want a value that happens to be the default.

Variable flow:
- channel set to defaults
- patch values not set to default are copied
- sound object values from track that are not set to defaults are copied to channel
- track player values set to defaults
- track attributes not set to defaults are copied to channel

Not all registers should be copied at at Play(). Paused, for example.

C. Interfaces

Patch

bool IsLooped()
bool IsStreamed()
bool IsPreloaded()
Sint32 ResourceId()
bool Load()
bool Unload()
bool Cache()
bool Uncache()

Channel

NoteOn()
NoteOff()
SetVolume()
SetPitch()

SetPan()
SetPosition()
SetFxType()
SetFxLevel()
Load() – calls cPatch::Load()
Unload() - ditto
bool Cache()
bool Uncache()

Track

Start(args)
Stop()
Update(args)
Sint32 ResourceId()
bool Load() – calls cChannel::Load()
bool Unload()
bool Cache()
bool Uncache()

Track Player

Start(args)
Stop()
Update(args)
bool Load() – calls cTrack::Load()
bool Unload()
bool Cache()
bool Uncache()

HitMan

Init()
Shutdown()
CreateTrackPlayer()
CreateChannel()
LoadHitFile()
LoadIniFile()

II. Track Player

The track player follows a loop of
- read command or note and args
- execute command

All commands and arguments are 1 byte. The "setl" command is the only one that can read a longword.

To execute the command, control is routed through a case statement that includes all the available commands. Each case reads in as many arguments as the command expects. Example:

```
switch( bCommand )
{
case( add )
      lDestReg = GetArg();    // Read dest register identifier
      lSrcReg = GetArg();     // Read src register identifier
      Sint32 lVal = GetRegVal(lDestReg) + GetRegVal(lSrcReg);
      SetReg( lVal );
      break;
case( note_on )
      lDuration = GetRegVal( GetArg() );
      NoteOn( lDuration );
case( end )
      End();
      break;
 . . .
```

Register access

GetRegVal() looks up a register by register ID and returns its value. The location of the variable varies.

Track parameters

See *Numbered Attribute Registers* in the Spec.

Format

<command> [<arg1> [<arg2> [<arg3> [<arg4>]]]]

Each line is a command byte followed by up to 4 argument bytes. Argument bytes enumerate registers.

Argument byte format

First 2 bits:
00 – global register
10 – local register from local track
11 – local register from target track if there is one, local track otherwise

This scheme allows us 64 local and 128 global registers.

Addressing modes

The addressing mode is “direct” by default. The “move” commands (moveb, movel) provide the only immediate addressing.

The stack

The stack lets you push and pop variables. It is also used for return addresses.

Instructions:
- push - pushes a single register value onto the stack
- push_mask(mask) - pushes a set of registers, identified by a mask
- pop – pops whatever has been pushed

The stack always knows what has been pushed. Push pushes a mask that is used to identify push calls as well as push_mask calls. You can't push a variable and pop the value into another variable. If this functionality is required we'll add a "popx" instruction or something like that.

Subroutine calls

[Issue – is it worth separating the subroutine and variable stacks?]

You can "call" a second track with the call command. The second track will return when it runs into a "return" command. You can push variables onto the stack with the "push", "push_vars", and "push_mask" commands.

```
call_mask          equ          55
call_push          equ          56
```

The stack:
stack-frame marker
var_mask
(vars)
return address

The "call" command pushes the return address and then stack frame identifier. The stack frame ID is just a special value that helps check for invalid stacks. The var mask has 1 bit for each register (the bit is set if the register was pushed).

- call – Push return address, 0, stack-frame marker. Jump to new track data
- return - Pop stack-frame marker, variables mask, variables, and return address. Jump to return address
- callpv – push return address, var_mask for variables 1-8, variables 1-8. stack-frame marker
- callmask – return address, var_mark specified by argument, variables specified by var_mask, stack-frame marker

Fades

A track can fade any variable of any sound object.

Target tracks

A track normally controls itself; however, you can tell a track to control another track. This means that when you set a register the register of the target track is set instead of the local register. Only the **load** instructions ignore the target track.

A track can explicitly load and set its own registers by using the **setll**, **setlt**, and **settl** instructions.

Run-time error checking

The track player checks for the following errors:
- stack errors (see "*Subroutine calls and the stack*")
- illegal instructions
- strange-looking long values (long values at the moment always seem to have the 2 MSBs set to 0)

If an error is encounter the track player calls the error routine. By default this routine kills the track an prints out an error message to the Gonzo debug window. HitMan can replace this routine, so that applications (i.e., the debugger) can handle the error more intelligently.

Breakpoints

There is a breakpoint command called "break". A debugger can replace a command with a break command. It needs to restore the instruction before that instruction can be executed. Whether the breakpoint needs to be restored after the instruction has executed depends on context – restore for normal breakpoints but not for "step over".

The debugger should maintain a hidden breakpoint for "step over".

The debugger needs the following information to implement a breakpoint:
- source file name
- source line number
- data address
- old instruction (where the break instruction is now)
- whether to restore the breakpoint after it has been executed
- enabled flag

The instruction
For step over the instruction must be restored

Addressing modes

The addressing mode is "direct" by default. The "move" commands (moveb, movel) provide the only immediate addressing.

A. Track Debugging

[Future]
Tracks provide some extra functionality for debugging.
Step – execute 1 command line
RunTo
GetCurrentAddr

GetRegister
SetRegister
The Interface
GetSourceFile(data addr)
GetSourceLine(data addr)
GetDataAddr(sourcefile line number, source filename)

The compiler generates the following debug information:
Data address for each line of source

Each source file generates a debug file (.hdbg) that contains the following information:

- source filename
- array of line numbers
- array of data points corresponding to each line number

Each track has a debug filename

When a track is compiled individually, it is given the same information for the temporary file.

III. HIT .ini File Format

```
[Options]
Version=1

[Patches]
;GUID=name,type,filename
0x01=PatchBoom,0,interface\boom.xa,looped,streamed,preloaded,3D option

 [Tracks]
;GUID=name,type,vol, pri,attack, decay, sample file GUID, etc
0x02=TrackBoom,0,1024,0,10,0,0,0x01
```

IV. HIT File Format

The HIT file contains only track data.

[Should we throw in samples? Not for now.]

Several HIT files can be loaded, so sound objects can be plugged in after release.

Format

```
char[4] lFileType = “.HIT”
Sint32 lNumSections
Sint32 alSectionOffset[lNumSections]
```

```
char[4] lSectionID = “TRAX”
Sint32 lNumLabels
Sint32 alTrackGUID[lNumLabels]
Sint32 alTrackDataOffset[lNumLabels]
<binary track data>
```