Cuestiones sobre el limitador de sonido LM9

Alejandro Ruiz Becerra

20 de mayo de 2021

Resumen

En este documento se transcriben una serie de cuestiones planteadas durante la reunión del equipo que forma parte del proyecto que tuvo lugar el día 3 de Mayo de 2021, y se pretende dar una solución o respuesta a dichas cuestiones.

1. Generación de ruido rosa

La estructura del limitador de sonido se compone de varios procesos relacionados, y en cierto modo dependientes, los cuales se han denominado módulos. De entre ellos, el módulo calibrador se encarga de generar los ficheros de calibración para las líneas derecha e izquierda y el micrófono. Estos ficheros de calibración se almacenan en el directorio dedicado a los ficheros de configuración del limitador de sonido (/var/slr) con el nombre de calibration#, donde # representa un número entero, normalmente 0, 1 o 2. Estos números identifican la línea a la que pertenece la calibración, siendo:

- 0: micrófono
- 1: línea izquierda
- 2: línea derecha

El proceso de calibración requiere de la emisión de ruido rosa para producir un resultado satisfactorio. En el modelo anterior, el LM7, la emisión de ruido rosa y la calibración se realizaba en PHP, por lo que el proceso de calibración estaba fuertemente ligado a la interfaz web del limitador de sonido. Sin embargo, en el modelo LM9, se han creado un programa en C++ exclusivamente para este propósito, aunque no es responsable de la emisión de ruido rosa, luego, ¿cómo se emite ruido rosa en este modelo?

Para averiguarlo se ha realizado una búsqueda sobre todo el código del LM9, usando las palabras clave 'pink' o 'rosa', con el fin de localizar los trozos de código que puedan tener alguna relación con la generación o emisión de ruido rosa. Los resultados de la búsqueda nos dicen que mientras la palabra clave 'rosa' no arroja ningún resultado, la palabra clave 'pink' nos lleva al dato miembro 'pinkNoiseLevel' de la clase Configuración. Esta variable es accedida en tres lugares distintos dentro del código. Sin embargo, esto solo indica el volumen de ruido rosa (presuntamente), y no implica la generación de ruido rosa.

Los accesos/modificaciones de 'pinkNoiseLevel' se dan en los siguientes lugares:

- Configuracion.{h, cpp}: definición del dato miembro como tipo float e inicialización de la misma a 0 dentro del constructor.
- Configurador.cpp::configuraDeFlujo(): se lee y aplica una configuración desde fichero. Si una de las líneas contiene la palabra 'PinkNoiseLevel', se escanea esa línea y se actualiza el valor del dato miembro 'pinkNoiseLevel' al valor leído.

  Un ejemplo sería: PinkNoiseLevel 27.5
- getConfig: solo consulta y muestra por pantalla

1.1. Conclusiones

Tal y como se ha comentado esta variable de tipo float indicaría, en todo caso, el volumen del ruido rosa, pero no implicaría directamente la emisión de ruido rosa por parte del limitador de sonido.

Por tanto, y tomando el modelo LM7, la opción más plausible es que la emisión de ruido rosa sigue estando controlada por la interfaz web mediante código PHP, y que los botones de la interfaz activen y desactiven la emisión de ruido rosa de forma directa.

Por otra parte, hay referencias a ficheros PHP que no están presentes en el código y cuyos nombres nos llevan a pensar que tienen una relación directa con la calibración del limitador de sonido.

- calibrationControl.php: lo tenemos, ya que viene del LM7.
- audioControl.inc: nuevo en LM9, faltante.
- audioInstallation.inc: nuevo en LM9, faltante.

2. Salida del audio procesado

Una de las principales cuestiones de la reunión fue cómo se realiza la atenuación efectiva. Para resolver esta cuestión nos vamos al código del limitador y lo analizamos.

```
1  void loop() {
2    while (true) {
3      // Ciclo de espera de 1 segundo donde:
4      //  - Se lee la atenuacion de entrada
5      //  - Se atenua frecuencialmente
6      for (int i = 0; i < UnderSecondTicks; i++) {
7        usleep(UnSegundoEnUsecs / UnderSecondTicks);
8
9        // Global control
10       applyGlobalControl();
11
12       // Apply correction over input to minimize pump effect
13       // applyPumpEffectCorrection();
14
15       // Apply frequency control
16       applyFastFrequencyControl();
17     }
18
19     // Muestra estado cada minuto
20     showGlobalData();
21     showFreqAttenuation();
22   }
23 };
```

Listing 1: Bucle principal del limitador de sonido

Podemos ver que el bucle principal se basa en dos funciones auxiliares, applyGlobalControl() y applyFastFrequencyControl().

2.1. Control global

En el caso del control global, se calcula la atenuación necesaria en cada momento según los valores de referencia de la líneas izquierda y derecha, los valores reales en ese momento de tales líneas, y el valor máximo de emisión para ese momento del día según la normativa aplicada.

Este valor de atenuación, denominado desiredAt, se envía al puerto serie junto a la cadena: **max -desiredAt 10 10 0 5**

```
1  sprintf(command, "echo max %.1f 10 10 %.1f 5 > /var/slr/hardware/
     port", -desiredAt, 0.0f);
2  system(command);
```

Listing 2: Control global

2.2. Control frecuencial

En el caso del control frecuencial se leen los stream de audio (fast) calibrados que genera el programa *analyzer*, se calcula cuánto se sobrepasa cada banda del ecualizador del máximo permitido por la normativa, y se calcula la atenuación necesaria a aplicar en cada una de las bandas.

Este tipo de control solo se aplica si en la configuración del limitador se ha activado el control frecuencial.

```
1  void frequencyControl() {
2    . . .
3
4    // Obtenemos el maximo frecuencial y la curva NC
5    float maximoGlobal = config.maximoEnEmision(time(NULL));
6    float recpetionMaximum = config.maximoEnRecepcion(time(NULL));
7    int ncCurveIndex = SelectNCCurveByGlobal(recpetionMaximum);
8
9    . . .
10
11   // Obtenemos lo que se sobrepasa
12   for (int i = 0; i < Spectrum_BandCount; i++)
13     overTake[i] = control[i] - maximo[i];
14
15
16   // Overtake en bandas de equalizador
17   // (El mayor sobrepaso por cada banda de octava)
18   for (int i = 0; i < EcualyzerBandCount; i++) {
19     int idx = i*3;
20     eqOvertake[i] = max(overTake[idx],
21                         overTake[idx+1],
22                         overTake[idx+2]);
23   };
24
25   // Control real!
26   for (int i = 0; i < EcualyzerBandCount; i++) {
27     // Attenuation process
28     if (eqOvertake[i] > 0)
29       atenuacion[i] += eqOvertake[i] / FrecuentialAttackRate;
30     // Banda no ha superado el maximo
31     else
32       atenuacion[i] += eqOvertake[i] / FrecuentialReleaseRate;
33
34     if (atenuacion[i] > config.atenuacionMaximaPorBanda)
35       atenuacion[i] = config.atenuacionMaximaPorBanda;
36     if (atenuacion[i] < 0)
37       atenuacion[i] = 0;
38   };
39
40 }
```

Listing 3: Control frecuencial

Aquí, el atenuacion es un puntero a un vector de floats, por lo tanto, lo que se

recibe es una posición de memoria y todos los accesos a atenuacion modifican el vector original. El vector que se pasa como argumento a esta función es float[10] equalization::HardwareStatus, es decir, se está modificando la ecualización del limitador de sonido de forma directa.

Es importante anotar que existe una zona de memoria compartida, en la que el vector equalization de HardwareStatus reside. El resto de procesos adjuntan el espacio de memoria compartida a sus espacios de memoria locales, de forma que pueden leer y escribir en ella, por tanto, cuando en la función frecuencyControl() se modifica este vector, se está modificando en el espacio de memoria compartida.

Por último, falta propagar estos valores para que tengan efecto. Aquí es donde entra en juego la clase UpdateHardwareStatus y su método UpdateHardwareToFitStatus(). La clase UpdateHardwareStatus contiene un conjunto de métodos que realiza operaciones sobre objetos de tipo HardwareStatus, y que en la mayoría de los casos suponen la escritura de un comando en el puerto serie, de forma que lo recoja el **Arduino** y éste actúe en consecuencia.

El programa HardwareServer, en su bucle infinito, llama al método UpdateHardwareToFitStatus() y le pasa dos objetos de tipo HardwareStatus, el estado actual del limitador, y el estado futuro o deseado, de forma que compruebe qué valores son necesarios modificar para movernos del estado actual al siguiente. En este método se recorren las bandas del ecualizador y se comprueban si son valores difieren, en cuyo caso es necesario aplicar una modificación en esa banda, llamando al método UpdateEq() de esta misma clase, el cual se encarga de escribir en el puerto serie un comando para modificar la ecualización de dicha banda.

```
1  snprintf(command, 119, "%s %d %.1f", EqCommand, band,
     nextAttenuation);
2  serialPort.WriteLn(command);
```

Listing 4: Escritura en puerto serie en UpdateEq()

Un ejemplo de este comando sería: 'Eq banda at', donde banda es un número entero en el rango [0-10] y at un valor de tipo flotante indicando el valor de dicha banda de ecualización.

2.3. Conclusiones

Para el control global, se envía la atenuación en forma de comando al fichero '/var/slr/hardware/port'. Otros programas, como HardwareServer, usan este fichero como puerto al abrir el puerto serie, y parece ser la vía de comunicación **Arduino-Limitador**.

De esto se deduce que el limitador hace uso de un PGA controlado directamente

por un **Arduino**, que recibe la atenuación necesaria a aplicar desde el limitador, quién calcula este valor.

En la parte del control frecuencial, la secuencia de ejecución termina con la escritura de un comando en el puerto serie, por lo tanto también se deduce que el control de la ecualización para el control frecuencial queda controlado directamente por el **Arduino**, mientras que el Limitador es el responsable de mantener el control lógico, calculando los valores a aplicar en cada banda de ecualización y enviando órdenes al **Arduino** para que aplique los nuevos valores.

3. Estado del micrófono

En el limitador no solo de controla las lecturas del micrófono, desde la tarjeta de sonido (dispositivo PCM) sino que también si detecta si micrófono se encuentra físicamente conectado al equipo o no. En el código eso se traduce literalmente en un valor booleano llamado micCableIsConnected, el cual es un dato miembro del la clase HardwareStatus. Esta variable solo cambia verdadero en la función UpdateInputData() del fichero HardwareServer.cpp.

La secuencia de ejecución para actualizar el estado del micrófono sería la siguiente:

1. El programa HardwareServer abre el puerto serie indicando como puerto de comunicación el fichero '/var/slr/hardware/port'. Luego, crea una hebra que ejecuta la función ReadFromHardware() en un bucle infinito.
2. ReadFromHardware() lee de puerto serie y llama a la función UpdateInputData() en el caso de que la cadena leída contenga la palabra 'Status'.
3. UpdateInputData() recibe la cadena y un objeto de tipo HardwareStatus (recordemos que micCableIsConnected pertenece a esta clase), la parsea y actualiza el objeto HardwareStatus a los valores leídos de la cadena.

La cadena se parsea con sscanf(), y sigue el siguiente formato:

```
{
  "Status": {
    "version": %5s,
    "ChannelPressure": [%f,%f,%f,%f],
    "MicConnection":%1s,
    "InputPressure": [%f,%f],
    "attenuation": [%f,%f,%f]
  }
}
```

Listing 5: Formato de la cadena leída desde el puerto serie

Se puede identificar claramente que la cadena se ajusta al formato JSON.

3.1. Conclusiones

Como todo parece indicar que el limitador hace uso de un **Arduino** que realiza las operaciones a bajo nivel como controlar los LEDs o el display LCD, se deduce que también controla si el micrófono está o no conectado físicamente, y transmite esta información mediante el puerto serie, transmitiendo la información en una cadena de texto que sigue el formato visto anteriormente.

4. Scripts de arranque

El directorio scripts/ que encontramos en el código del limitador LM9 contiene un conjunto de scripts en varios lenguajes de programación, principalmente bash y PHP. Todos estos scripts están destinados a residir en el directorio /bin del sistema, de forma que puedan ser invocados de manera directa, es decir, sin especificar su path.

Entre estos scripts, algunos resultan de especial interés.

4.1. startUp

Es el script de arranque principal, en el que se realizan todas las tareas de inicialización del sistema, como configurar las interfaces de red, cargar los módulos del Kernel, generar las estructuras de carpetas necesarias, gestionar los permisos y compilar y mover los ejecutables a su destino final (/bin).

De entre todas las sentencias nos llama especialmente la atención la carga de un módulo de Kernel llamado SoundCapturer, el cual se carga o descarga con las órdenes modprobe y rmmod junto a otros módulos como snd-usb-audio y snd-mixer-oss.

Llama también la atención el uso de los ficheros especiales /dev/mmcblk0p7 y /dev/mmcblk0p4, los cuales corresponden a particiones dentro de la tarjeta SD en dispositivos Raspberry Pi (Fuente: What is /dev/mmcblk0p2 ?)

4.1.1. Conclusiones

Por convención, todos los módulos Kernel de Linux se nombran usando exclusivamente minúsculas, y en el caso de que el nombre se componga de varias

palabras, la separación entre ellas se realiza usando barras bajas (snake_case) o guiones (kebab-case).

Por tanto, la inconsistencia en el nombre del módulo SoundCapturer nos hace pensar que no se trata de un módulo Kernel oficial. Esto, junto con una búsqueda de dicho módulo en Internet sin resultados, nos lleva a pensar que este módulo Kernel ha sido implementado por los desarolladores del limitador de sonido LM9.

4.2. runVersionCommands

La última sentencia del script startUp invoca a este script, el cual se encarga de lanzar todos los procesos que componen el limitador de sonido en la secuencia correcta. La salida de estos procesos normalmente se descarta redireccionando la salida estándar (stdout) y el salida de error estándar (stderr) a /dev/null.

Para lanzar los procesos se delega en scripts auxiliares que sí que lanzan los procesos finalmente.

4.3. runLinesAnalyzer

El proceso analyzer puede recibir dos argumentos en su llamada, el primero es el nombre del dispositivo de sonido del cual leer el audio, y el segundo, el número de canales del stream de audio.

```
1  #!/bin/bash
2
3  read HW </var/slr/channels/lines.hw
4  echo "Lines Analyzer"
5
6  # amixer -D $HW sset "PCM Capture Source" 'Line'
7
8  while true; do
9    analyzer $HW 2
10 done
```

Listing 6: Script runLinesAnalyzer

El script lee el nombre del dispositivo de sonido a usar para la entrada estéreo de audio desde el fichero, y luego invoca al proceso analyzer pasando el nombre del dispositivo PCM y el número de canales como argumentos.

4.4. runMicAnalyzer

Para analizar los valores del micrófono se levanta un nuevo proceso analyzer de forma casi idéntica a como se hace en el script runLinesAnalyzer. El- micrófono es mono y no estéreo, es decir, solo tiene un canal, por lo que en este caso no se indica el número de canales del stream de audio al levantar el proceso, y solo se indica el dispositivo PCM del cual leer.

Adicionalmente, antes de levantar el proceso analyzer, se ajusta el nivel de captura del micrófono al 50 % con el comando amixer.

```
#!/bin/bash

read HW </var/slr/channels/mic.hw
echo "Mic Analyzer"

amixer -D $HW sset "Mic" capture 50%

while true; do
  analyzer $HW
done
```

Listing 7: Script runMicAnalyzer

4.5. refreshAnalyzer

El proceso analyer se compone en esencia de un bucle infinito dentro de su main() en el cual se lee el audio desde el dispositivo PCM indicado, se aplican las calibraciones almacenadas en los ficheros de calibración, y se almacena el audio en ficheros para ser utilizados por otros procesos del limitador. Por tanto, si el proceso analyzer es un bucle infinito, ¿por qué en los scripts también hay bucles infinitos que levantan instancias del programa analyzer?

La respuesta está en el script refreshAnalyzer, también un bucle infinito el cual 'mata' los analizadores de sonido si se cumple alguna de las siguientes condiciones:

- El valor del micrófono es menor a 10dBA.
- Han pasado 30 segundos.
- Se leen 5 valores iguales consecutivos desde el micrófono
- Se leen 5 valores iguales consecutivos desde la línea derecha o izquierda.

Por tanto al 'matar' al proceso analyzer su bucle infinito se rompe y se vuelve al bucle infinito de los scripts runLinezAnalyzer y runMicnAnalyzer, los cuales vuelven a levantar los procesos de análisis.

4.6. llControl

Este script se encarga principalmente de controlar los leds RGB y el display LCD, por lo que deduzco que el nombre del script proviene de LcdLedControl, y fue simplificado a llControl.

En este script llama la atención el código comentado, ya que hay instrucciones que hacen uso del dispositivo /dev/ttyACM0, lo cual aporta pruebas sólidas de la existencia de un Arduino como parte del hardware del limitador de sonido LM9 (Fuente: What is Dev ttyACM0?).