International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

ALSA Debugging Tools and Techniques in Linux Kernel

Anish Kumar

Email: *yesanishhere[at]gmail.com*

Abstract: ***This document provides different tools and techniques useful for debugging audio issues in Linux, covering problems from booting to shutdown. This guide is not exhaustive but aims to explain potential audio issues or bugs that can arise when bringing up audio drivers on Linux or fixing existing audio corruption issues during playback or capture.***

Keywords: ALSA, Audio, Linux kernel, Debug, XRUN

1. Introduction

Audio debugging in Linux systems requires a systematic approach to identify and resolve issues at various stages of the audio pipeline. This paper presents a comprehensive guide to audio debugging tools and techniques, covering bootup issues, runtime debugging, and specific problems such as audio corruption and XRUNs.

1.1 Bootup/Bringup Issues

In order to provide audio services, the ALSA sound card is the first component that gets registered in the system. If you are not hearing boot-up sounds, check the following in the procfs filesystem:

a) **/proc/asound/**: This is a virtual filesystem in Linux that provides information about the ALSA sound system. It allows users to interact with and obtain details about audio devices.
b) Check if **/proc/asound/cards** is populated. If it is not, this means sound card registration has failed. All sound cards are listed there. This entry can be useful for:
   - Verifying if a sound card has been properly registered and is present in the system.
   - Finding the index of a given card. Card indexes are dynamically assigned as cards are registered. User-space software may assume cards have certain indexes, which could change depending on runtime dynamics. For example, index 0 may be assigned to card B instead of card A if card A's registration is deferred due to a missing dependency (e.g., an I2C codec that has not been probed yet).

If the sound card is registered, check its capabilities and devices for playback. There are various devices that can be connected to the system, such as wired headsets, Bluetooth headsets, and internal speakers. If you are looking to play audio on a wired headset, look for the corresponding entry in:

a) /proc/asound/pcm: This lists all PCM devices for all sound cards present in the system. Each PCM device has a directory structure like: /proc/asound/cardX/pcmY[p,c]/. Here, X is the card index number (as found in /proc/asound/cards) and Y is the PCM device index. The [p,c] indicates the direction of the PCM stream:
   - **p**: Playback stream. Represents the PCM device used for audio output (e.g., sending audio data to speakers).
   - **c**: Capture stream. Represents the PCM device used for audio input (e.g., receiving audio data from a microphone).

For playback, the relevant PCM device should be listed as: /proc/asound/card0/pcm0p0c0.

Example output from my system:

```
00-00: CX20632 Analog : CX20632 Analog : playback 1 : capture 1
00-02: HDMI 0 : HDMI 0 : playback 1
```

This output indicates that I have only one sound card in my system, with starting index 0, capable of both playing and capturing audio. The same sound card also provides HDMI playback capability, but no ARCIN, so there is no capture path.

1.2 How Sound Card Registration Happens

If the sound card is not registered, or if it is registered but the corresponding device node is not created, it is important to understand how the ASoC core creates these nodes. In summary, it creates PCM nodes once it finds all the CPU DAI, codec DAI, and platform DAI associated with the particular snd_soc_dai_link instance.

For example, in the kernel source file littlemill.c [3]:

```
SND_SOC_DAILINK_DEFS(cpu,
  DAILINK_COMP_ARRAY(COMP_CPU("samsung-i2s.0")),
```

```
  DAILINK_COMP_ARRAY(COMP_CODEC("wm8994-codec", "wm8994-aif1")),
  DAILINK_COMP_ARRAY(COMP_PLATFORM("samsung-i2s.0")));

SND_SOC_DAILINK_DEFS(baseband,
  DAILINK_COMP_ARRAY(COMP_CPU("wm8994-aif2")),
  DAILINK_COMP_ARRAY(COMP_CODEC("wm1250-ev1.1-0027", "wm1250-ev1")));
```

The SND_SOC_DAILINK_DEFS macro helps define the parameters for a digital audio interface (DAI) link in a structured way. The macro typically includes definitions for several important components of a DAI link:

- **CPU DAI**: The data interface associated with the CPU that handles audio data. It defines how the CPU interacts with the audio subsystem.
- **Codec DAI**: The data interface for the audio codec, which converts digital audio data to an analog signal (and vice versa). This is crucial for playback and capture.
- **Platform DAI**: Represents the platform layer that connects the CPU and codec DAI. It handles the specifics of data transfer between the CPU and codec.
- **Stream Names**: Defines the names for playback and capture streams, which are used by the audio subsystem for routing audio data.

This can be simplified as follows for easier understanding:

```
SND_SOC_DAILINK_DEFS(my_dai_link,
  SND_SOC_DAILINK_REG(cpu_dai),
  SND_SOC_DAILINK_REG(codec_dai),
  SND_SOC_DAILINK_REG(platform_dai));
```

When **devm_snd_soc_register_card** is called, the ASoC core checks if the CPU, codec, and platform DAI probe callbacks are successful. If any of these fail, the sound card registration fails.

There are two probes involved: one for the registration of the device (which could be I2C, I2S, or any hardware device) and the other is the probe callback from the ASoC core. Both must succeed for the devices to show up in the proc filesystem. For example, if the codec driver is I2C-based, the first probe for the I2C driver should succeed, and as part of this, the ASoC component gets registered. When the card is being registered by the machine driver, the ASoC core checks if the codec component probe is successful. The same applies to other CPU and platform components.

1.3 Runtime Debugging

Once the sound card and corresponding PCM nodes appear in the procfs filesystem, basic audio booting is successful. However, this does not guarantee successful playback using the aplay command. For example:

```
aplay -D hw:0,0 sound.wav
```

if audio does not play, check the following:

- **Unsupported Sample Rate**: Ensure the audio file's sample rate matches the hardware's supported rates. Use:`aplay -f S16_LE -c 2 -r 44100 sound.wav`
- if the hardware does not support the requested sample rate, it may lead to playback issues.
- **Incorrect Channels**: Verify that the number of channels in your audio file matches the hardware's supported configuration (e.g., stereo vs. mono).
- **Bit Depth Issues**: Ensure the bit depth of the audio file is supported by the hardware.
- **Device Configuration**: Confirm that the card and device numbers are correct by checking your device list with: `aplay -l`
- **Format Conversion**: If the format is unsupported, convert it to a compatible format (e.g., WAV).

1.4 XRUN Debugging

a) **What is XRUN?**

XRUN refers to an overflow or underflow condition in the audio buffer during playback or recording.

b) **Types of XRUN**

- **Overflow (XRUN)**: This occurs when the audio playback buffer runs out of data to play. This situation arises when the application is not supplying data to the buffer quickly enough, leading to dropouts or gaps in the sound.
- **Underflow (XRUN)**: This happens when the audio capture buffer fills up before the application can process the incoming data. If the application does not read the data quickly enough, it causes data loss, and the capture stream fails to record audio smoothly.

c) **Causes of XRUNs**

- **High CPU Load**: If the system is under heavy load, it may not process audio interrupts on time, leading to missed interrupts.
- **Incorrect Period Size**: A period size that is too small may result in system scheduling delays, causing interrupts to be handled late and leading to frame drops and ultimately XRUNs.
- **Inefficient Audio Processing by Userspace**: If userspace tasks take too long to execute, this can delay the feeding or retrieval of data from the buffer.
- **I/O Latency**: Latency from I/O in single-threaded, non-optimized audio applications can cause significant problems related to XRUN. Due to large I/O latencies, the ALSA buffer is likely to overflow or underflow. Applications like aplay, arecord, tinyplay, and tinyrecord may suffer from this issue because they are single-threaded.
- **Misconfiguration**: Improper settings in the audio application or driver configuration can contribute to XRUN occurrences.

d) Debugging XRUN

Diagnosing the exact cause of XRUNs can be challenging. However, the following steps can help identify the issue:

1) **Reduce CPU Load**: Stop any applications other than the

playback application to see if the XRUNs cease.
2) **Increase Period Size**: Adjusting the period size can give the system more time to handle interrupts.
3) **Test with Special Files**: If I/O latency is suspected, try playing audio via special files like **/dev/zero** (e.g., `aplay -D hw:0,0 -f CD -d 30`), as this reads data from RAM rather than I/O.
4) **Change Userspace Application**: Switch to a different application to see if it is the source of the problem.
5) **Use ftrace**: Profile the call flow of the ALSA drivers using ftrace to determine if any function calls are taking too long to execute during XRUN events.
6) **Check ALSA Documentation**: Refer to the ALSA XRUN Debugging Guide [1] for additional troubleshooting steps.
7) **Inspect procfs Filesystem**: Use the command:

```
cat /proc/asound/cardX/pcmY/subZ/status
```

to dump the hardware and software pointers of the ring buffer used by ALSA. Check if the pointers are incrementing correctly. If they do not change, it indicates that the underlying hardware is unable to process data, which could be due to misconfiguration or some bug in interrupt processing.

e) **Example Status Output**
The following command displays the status output for a specific PCM device:

```
# cat /proc/asound/card0/pcm0p/sub0/status
```

The output is as follows:

```
state: RUNNING
owner_pid: 1234
trigger_time: 1000000.1000000
tstamp: 1000000.1000000
delay: 1234
avail: 0
avail_max: 0
hw_ptr: 64
appl_ptr: 4192
```

Description of Fields:
- **state**: Current state of the PCM device (e.g., RUNNING).
- **owner_pid**: Process ID of the owner of the PCM stream.
- **trigger_time**: Time when the last trigger occurred.
- **tstamp**: Timestamp of the last audio sample processed.
- **delay**: Current delay in microseconds.
- **avail**: Available frames in the buffer.
- **avail_max**: Maximum frames that can be made available.
- **hw_ptr**: Hardware pointer indicating the position in the buffer.
- **appl_ptr**: Application pointer indicating the position in the buffer.

f) Fixing XRUNs
- **Optimize Application**: Convert single-threaded applications like aplay/arecord/tinyplay/tinyrecord to multi-threaded ones. This allows one thread to read multimedia data while another thread writes to the hardware. An additional RAM circular buffer may be needed to manage I/O latency. The thread reading from I/O should read larger amounts of data into the circular buffer, while the thread writing to hardware reads from this buffer.
- **Increase Period Size**: Adjusting the period size can help accommodate delays in audio processing, allowing the CPU more time to handle interrupts.
- **Optimize System Performance**: Reducing CPU load or prioritizing audio processing can help mitigate XRUNs [1].

1.5 Audio Pops and Clicks

For detailed information about audio pops and clicks, refer to the Audio Pops and Clicks Documentation [2].

a) **Debugging Pop Issues**
- To diagnose and resolve pop issues, consider the following steps:

Check Sequencing:
- Ensure that the sequencing mentioned in the above document is being followed correctly.

Device Switching:
- If pops occur during the switch from one device to another, test by playing zero data to determine if the issue persists: `aplay -f dat /dev/zero`
- If the pops continue, it indicates that the device switch is causing the problem.

Register Write Events:
- If pops occur during register writes due to triggered events, you can introduce a delay between register writes with the following command:
- `echo 1000 > /sys/kernel/debug/asoc/name-of-the-sound-card/dapm_pop_time`
- This command introduces a delay between register writes, which may help in identifying the specific register causing the pop.

2. Conclusion

Audio debugging in Linux systems requires a methodical approach and understanding of various components in the audio pipeline. By following the techniques and utilizing the tools described in this paper, developers can effectively diagnose and resolve audio-related issues in Linux environments. Regular system monitoring, proactive debugging, and staying updated with the latest ALSA developments are key to maintaining optimal audio performance in Linux systems.

References

[1] ALSA XRUN Debugging Guide. Available: https://www.alsa-project.org/main/index.php/XRUN_Debug
[2] Audio Pops and Clicks Documentation. Available:
[3] Little Mill Documentation. Available: https://www.kernel.org/doc/Documentation/sound/alsa/ASoC/LittleMill.txt