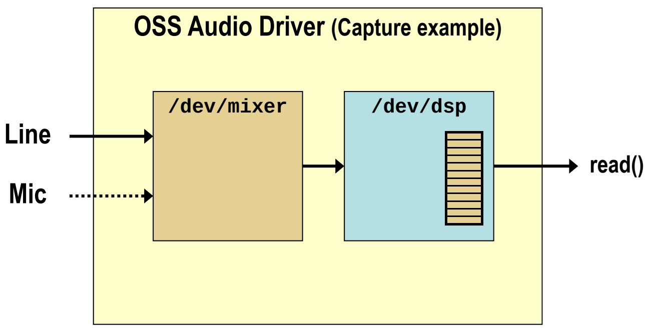
Linux Embedded System Design Workshop

Outline

- Driver Basics
- Linux Audio Drivers
  - Open Sound System (OSS)
  - Adv Linux Sound Arch (ALSA)
- Digital Media App Interface (DMAI)
- Linux Signal Handler
- Lab Exercise

Linux OSS Driver



OSS Driver consists of two main parts
- Mixer – allows for one or more inputs
- Sound device

Used in older Linux distributions; e.g. MV Linux for DM6446/DM3xx

Outline

- Driver Basics
- Linux Audio Drivers
  - Open Sound System (OSS)
  - Adv Linux Sound Arch (ALSA)
- Digital Media App Interface (DMAI)
- Linux Signal Handler
- Lab Exercise

ALSA Applications

- The ALSA driver provides 3 Linux applications to exercise the driver.
- While not fancy, record/play app's are useful for testing. The mixer is useful for choosing inputs/outputs.
- The three app's are:

arecord	Record audio from an ALSA device to a file
aplay	Play recorded audio over an ALSA device
amixer	Select input sources and adjust relative volume levels

```
$ opkg update
$ opkg install alsa-utils-aplay alsa-utils-amixer
```

ALSA Library API

Information Interface /proc/asound
Status and settings for ALSA driver.

Control Interface /dev/snd/control*CX*
Control hardware of system (e.g. adc, dac).

Mixer Interface /dev/snd/mixer
Controls volume and routing of on systems with multiple lines.

PCM Interface /dev/snd/pcm*CXDX*
Manages digital audio capture and playback; most commonly used.

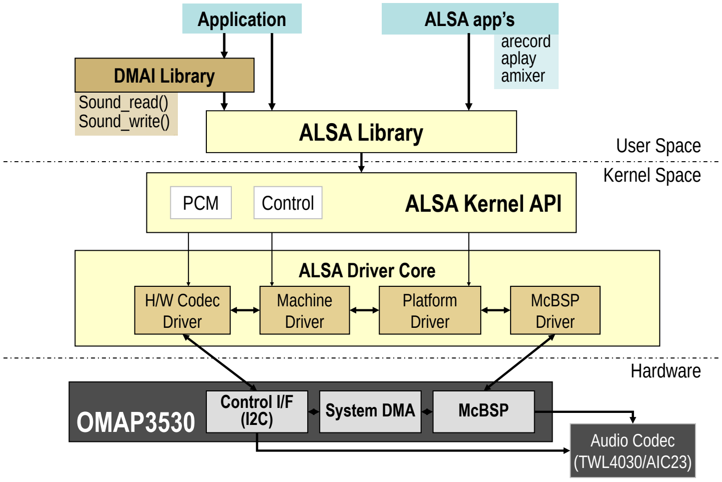
Raw MIDI Interface* /dev/snd/midi*CXDX*
Raw support for MIDI interfaces; user responsible for protocol/timing.

Sequencer Interface* /dev/snd/seq
Higher-level interface for MIDI programming.

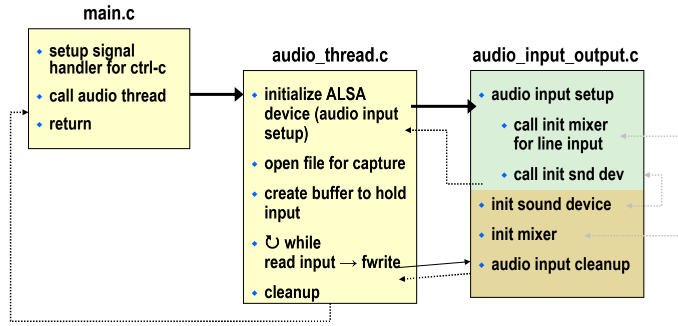
Timer Interface /dev/snd/timer
Timing hardware used for synchronizing sound events.

* Not implemented in current TI provided driver.

ALSA Implementation : Block Diagram



Example – Audio Capture



Notes:

- This is the example found in Lab 6a (v2.10 labs)
- Signal handler discussed later in chapter

Outline

- **Driver Basics**
- **Linux Audio Drivers**
  - Open Sound System (OSS)
  - Adv Linux Sound Arch (ALSA)
- **Digital Media App Interface (DMAI)**
- **Linux Signal Handler**
- **Lab Exercise**

Linux Signals

A signal is an event generated by Linux in response to some condition, which may cause a process to take some action when it receives the signal.

- **"Raise" indicates the generation of a signal**
- **"Catch" indicates the receipt of a signal**

Linux Signals

- A signal may be raised by error conditions such as:
  - Memory segment violations
  - Floating-point processor errors
  - Illegal instructions
- A signal may be generated by a shell and terminal handlers to cause interrupts
- A signal may be explicitly sent by one process to another

Signals defined in signal.h

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal

** Note, this is not a complete list*

Raising / Catching a Signal

- **Raising:**
  - A foreground process can be sent the SIGINT signal by typing Ctrl-C
  - Send to background process using the kill command:

    example: **`$ kill -SIGKILL 3021`**
- **Receiving/Catching:**
  - If a process receives a signal without first arranging to catch it, the process is terminated
  - SIGKILL (9) cannot be caught, blocked, or ignored

Handling a Signal

A program can handle signals using the signal library function:

```
#include <signal.h>

void (*signal (int sig, void(*func)(int)));
```

integer signal
to be caught or ignored

function to be called
when the specific
signal is received

main.c

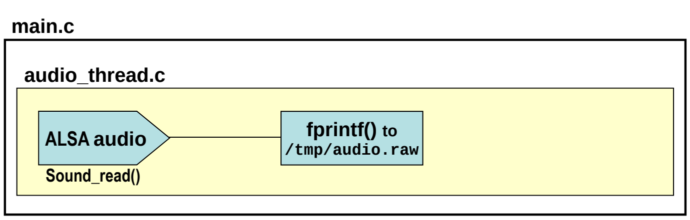
```
int main(int argc, char *argv[ ])
{
 int status = EXIT_SUCCESS;
   void *audioThreadReturn;

   /* Set the signal callback for Ctrl-C */
   signal(SIGINT, signal_handler);

   /* Call audio thread function */
   audioThreadReturn = audio_thread_fxn((void *) &audio_env);

   if( audioThreadReturn == AUDIO_THREAD_FAILURE ){
      DBG("audio thread exited with FAILURE status\n");
      status = EXIT_FAILURE;
   }
   else
      DBG("audio thread exited with SUCCESS status\n");

   exit(status);
}

/* Callback called when SIGINT is sent to the process (Ctrl-C). */
void signal_handler(int sig)
{
   DBG("Ctrl-C pressed, cleaning up and exiting..\n");
   audio_env.quit = 1;  // used as while loop condition
}
```

- setup signal handler for ctrl-c
- call audio thread
- return

Outline

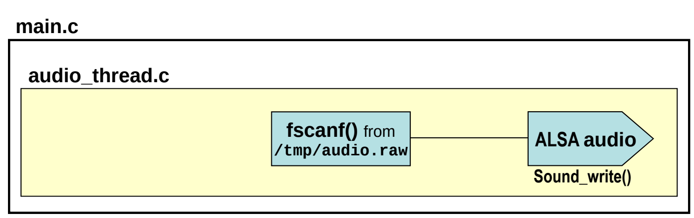
- **Driver Basics**
- **Linux Audio Drivers**
  - Open Sound System (OSS)
  - Adv Linux Sound Arch (ALSA)
- **Digital Media App Interface (DMAI)**
- **Linux Signal Handler**
- **Lab Exercise**

lab06a_audio_record



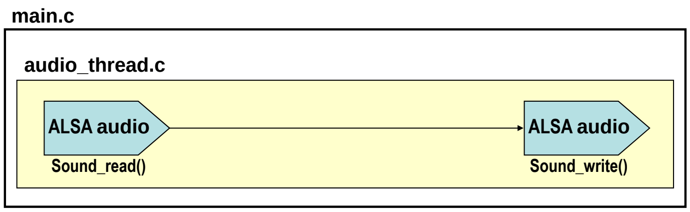
- **Goal: Analyze the function calls necessary to record audio from a line input to a file.**
- Inspection lab only.
  1. Inspect the source files in this application.
  2. Build and run the application: Result: capture audio into a file: audio.raw.
  3. Add a new DBG() statement and inspect how DBG/ERR macros work in the system.

lab06b_audio_playback



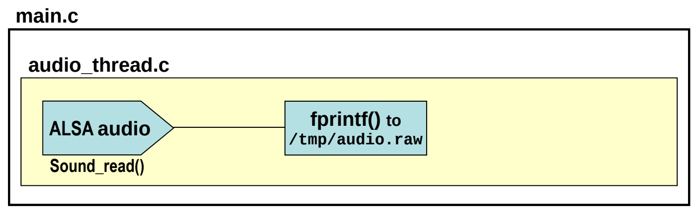
- **Goal: Analyze the function calls necessary to play back audio from a recorded file to the driver.**
- Inspection lab only.
  1. Inspect audio_thread.c and the associated helper functions.
     Sound_write() from DMAI library writes audio buffer to audio driver.
  2. Build and run the application.
  3. Result: Audio in audio.raw is sent to the audio driver.

lab06c_audio_loopthru



- **Goal: Combine the record (lab06a) and playback (lab06b) into an audio loopthru application.**
- Hey – YOU get to do this yourself (no more inspection stuff…)
  1. Answer a few questions about the big picture (covered in the next few slides…)
  2. Copy files from lab06b (playback) to lab06c (loopthru)
  3. Make code modifications to stitch the record to the playback (covered in the next few slides…).
  4. Build, run. Result: audio is recorded (from ALSA input), copied from in → out buffer, then played back (to ALSA output).

lab06a_audio_record



- **Goal: Analyze the function calls necessary to record audio from a line input to a file.**
- Inspection lab only.
  1. Inspect the source files in this application.
  2. Build and run the application: Result: capture audio into a file: audio.raw.
  3. Add a new DBG() statement and inspect how DBG/ERR macros work in the system.

main.c

line 42

```
int main( int argc, char *argv[] )
{
    int   status = EXIT_SUCCESS;
    void *audioThreadReturn;
    // Set the signal callback for Ctrl-C
    pSigPrev = signal(SIGINT, signal_handler);

    // Call audio thread function
    audioThreadReturn = audio_thread_fxn( (void *) &audio_env );

    if( audioThreadReturn == AUDIO_THREAD_FAILURE ) {
        DBG( "Audio thread exited with FAILURE status\n" );
        status = EXIT_FAILURE;
    }
    else
        DBG( "Audio thread exited with SUCCESS status\n" );

    exit( status );
}
```

audio_thread.c, 83

48 000

4

snd_pcm_t

"plughw:1,0"

48000

pcm.h

48 000/4

```
exact_bufsize = blksize/BYTESPERFRAME;

if( audio_io_setup( &pcm_capture_handle,
        SOUND_DEVICE,
        SAMPLE_RATE,
        SND_PCM_STREAM_CAPTURE,
        &exact_bufsize ) == AUDIO_FAILURE ) {
    ERR( "Audio_input_setup failed in audio_thread_fxn\n\n" );
      status = AUDIO_THREAD_FAILURE;
      goto cleanup;
    }
    DBG( "exact_bufsize = %d\n",
       (int) exact_bufsize);
```

```
#define ERR(fmt, args...) fprintf(stderr, "Error: " fmt, ## args)
```

```
#define DBG(fmt, args...) fprintf(stderr, "Debug: " fmt, ## args)
```

debug.h

```
// Enables or disables debug output
#ifdef _DEBUG_
    #define DBG(fmt, args...) fprintf(stderr, "Debug: " fmt, ## args)
#else
    #define DBG(fmt, args...)
#endif

#define ERR(fmt, args...) fprintf(stderr, "Error: " fmt, ## args)
```

audio_thread.c, 95

```
// The levels of initialization for initMask
#define      INPUT_ALSA_INITIALIZED       0x1
#define      INPUT_BUFFER_ALLOCATED       0x2
#define      OUTPUT_FILE_OPENED           0x4
```

```
// Record that
initMask |

blksize = exact_bufsize*BYTESPERFRAME;
// Create input buffer to read into from input device
if( ( inputBuffer = malloc( blksize ) ) == NULL ) {
    ERR( "Failed to allocate memory for input block (%d)\n", blksize );
        status = AUDIO_THREAD_FAILURE;
        goto  cleanup ;
    }
DBG( "Allocated input audio buffer of size %d to address %p\n",
      blksize, inputBuffer );
// Record that input OSS device was opened in initialization bitmask
initMask |= INPUT_ALSA_INITIALIZED;
```

audio_thread.c, 115

```
// Open a file for record
outfile = fopen(OUTFILE, "w");

if( outfile == NULL ) {
        ERR( "Failed to open file %s\n", OUTFILE );
        status = AUDIO_THREAD_FAILURE;
        goto  cleanup ;
    }

DBG( "Opened file %s with FILE pointer = %p\n", OUTFILE, outfile );

// Record that input OSS device was opened in initialization bitmask
    initMask |= OUTPUT_FILE_OPENED;
```

audio_thread.c, 130

```
// Thread Execute Phase -- perform I/O and processing
while( !envPtr->quit ) {
    // Read capture buffer from ALSA input device
    if( snd_pcm_readi(pcm_capture_handle, inputBuffer,
                blksize/BYTESPERFRAME) < 0 ) {
       snd_pcm_prepare(pcm_capture_handle);
       ERR( "<<<<<<<<<<<<<<< Buffer Overrun >>>>>>>>>>>>>>>\n");
       ERR( "Error reading the data from file descriptor %d\n",
              (int) pcm_capture_handle );
       status = AUDIO_THREAD_FAILURE;
       goto  cleanup ;
    }
    if( fwrite( inputBuffer, sizeof( char ),
               blksize, outfile ) < blksize ) {
       ERR( "Error writing the data to FILE pointer %p\n", outfile );
       status = AUDIO_THREAD_FAILURE;
       goto cleanup;
        }
    }
    DBG( "Exited audio_thread_fxn processing loop\n" );
```

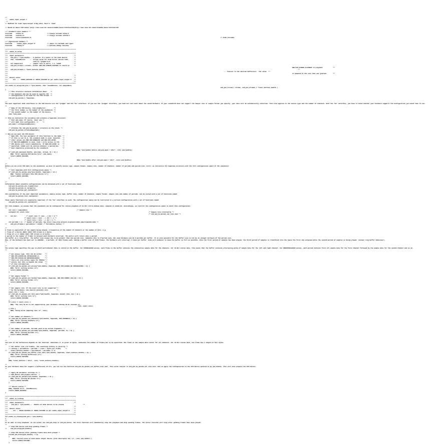
audio_thread.c. 163

```
cleanup:
    DBG( "Starting audio thread cleanup to return resources to system\n" );

    // Close the audio drivers
    // ***********************
    //  - Uses the initMask to only free resources that were allocated.
    //  - Nothing to be done for mixer device, as it was closed after init.

    // Close input device
    if( initMask & INPUT_ALSA_INITIALIZED )
       if( audio_io_cleanup( pcm_capture_handle ) != AUDIO_SUCCESS ) {
            ERR( "audio_input_cleanup() failed for file descriptor %d\n",
               (int) pcm_capture_handle );
            status = AUDIO_THREAD_FAILURE;
        }
```

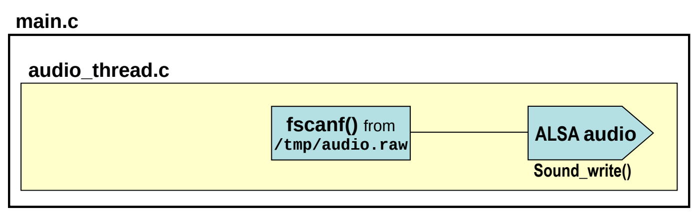
audio_thread.c, 180

```
 // Close output file
    if( initMask & OUTPUT_FILE_OPENED ) {
        DBG( "Closing output file at FILE ptr %p\n", outfile );
        fclose( outfile );
    }
// Free input buffer
    if( initMask & INPUT_BUFFER_ALLOCATED ) {
        DBG( "Freeing audio input buffer at location %p\n", inputBuffer );
        free( inputBuffer );
        DBG( "Freed audio input buffer at location %p\n", inputBuffer );
    }

    // Return from audio_thread_fxn function
    // *************************************

    // Return the status at exit of the thread's execution
    DBG( "Audio thread cleanup complete. Exiting audio_thread_fxn\n" );
    return status;
}
```

audio_input_output.c

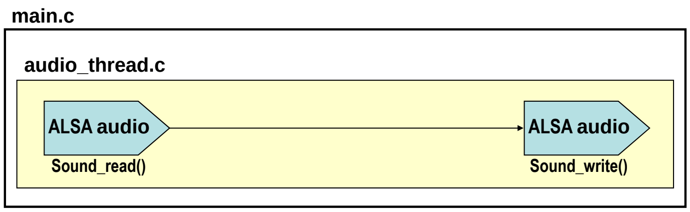


lab06b_audio_playback



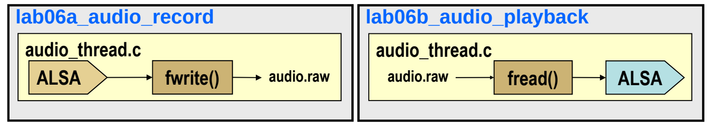
- **Goal: Analyze the function calls necessary to play back audio from a recorded file to the driver.**
- Inspection lab only.
  1. Inspect audio_thread.c and the associated helper functions.
     Sound_write() from DMAI library writes audio buffer to audio driver.
  2. Build and run the application.
  3. Result: Audio in audio.raw is sent to the audio driver.

lab06c_audio_loopthru



- **Goal: Combine the record (lab06a) and playback (lab06b) into an audio loopthru application.**
- Hey – YOU get to do this yourself (no more inspection stuff…)
  1. Answer a few questions about the big picture (covered in the next few slides…)
  2. Copy files from lab06b (playback) to lab06c (loopthru)
  3. Make code modifications to stitch the record to the playback (covered in the next few slides…).
  4. Build, run. Result: audio is recorded (from ALSA input), copied from in → out buffer, then played back (to ALSA output).

lab06c_audio_loopthru



Lab 6a

Which function gets an audio buffer?

Get data:

____

Put data:
fwrite() inputBuffer → outFile

Lab 6b

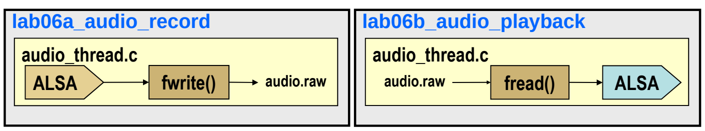
Which function puts to the ALSA driver?

Get data:
fread() inputFile → outputBuffer

Put Data:

____

lab06c_audio_loopthru



Lab 6a

Which function gets an audio buffer?

Get data:
Sound_read() inputFd → inputBuffer

Put data:
fwrite() inputBuffer → outFile

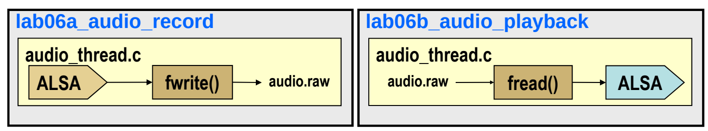
Lab 6b

Which function puts to the ALSA driver?

Get data:
fread() inputFile → outputBuffer

Put Data:
Sound_write() outputBuffer → outputFd

lab06c_audio_loopthru



Lab 6a

Which function gets an audio buffer?

Get data:
Sound_read() inputFd → inputBuffer

Put data:
fwrite() inputBuffer → outFile

~~Lab 6b~~

Which function puts to the ALSA driver?

Get data:
~~fread() inputFile → outputBuffer~~

Put Data:
Sound_write() outputBuffer → outputFd

For Lab06c:
- Take the code from lab06b and copy to Lab06c.
- Replace the fread() in Lab06b with the read() from Lab06a.