Linux Sound Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

ALSA KERNEL API DOCUMENTATION

1.1 The ALSA Driver API

1.1.1 Management of Cards and Devices

Card Management

int **snd_device_alloc**(struct device **dev_p, struct snd_card *card)

   Allocate and initialize struct device for sound devices

Parameters

struct device **dev_p
   pointer to store the allocated device

struct snd_card *card
   card to assign, optional

Description

For releasing the allocated device, call put_device().

int **snd_card_new**(struct device *parent, int idx, const char *xid, struct *module* *module, int
            extra_size, struct snd_card **card_ret)

   create and initialize a soundcard structure

Parameters

struct device *parent
   the parent device object

int idx
   card index (address) [0 ... (SNDRV_CARDS-1)]

const char *xid
   card identification (ASCII string)

struct module *module
   top level module for locking

int extra_size
   allocate this extra size after the main soundcard structure

struct snd_card **card_ret
   the pointer to store the created card instance

The function allocates snd_card instance via kzalloc with the given space for the driver to use freely. The allocated struct is stored in the given card_ret pointer.

Return

Zero if successful or a negative error code.

int **snd_devm_card_new**(struct device *parent, int idx, const char *xid, struct *module* *module, size_t extra_size, struct snd_card **card_ret)

managed snd_card object creation

Parameters

struct device *parent
    the parent device object

int idx
    card index (address) [0 ... (SNDRV_CARDS-1)]

const char *xid
    card identification (ASCII string)

struct module *module
    top level module for locking

size_t extra_size
    allocate this extra size after the main soundcard structure

struct snd_card **card_ret
    the pointer to store the created card instance

Description

This function works like *snd_card_new()* but manages the allocated resource via devres, i.e. you don't need to free explicitly.

When a snd_card object is created with this function and registered via *snd_card_register()*, the very first devres action to call *snd_card_free()* is added automatically. In that way, the resource disconnection is assured at first, then released in the expected order.

If an error happens at the probe before *snd_card_register()* is called and there have been other devres resources, you'd need to free the card manually via *snd_card_free()* call in the error; otherwise it may lead to UAF due to devres call orders. You can use *snd_card_free_on_error()* helper for handling it more easily.

Return

zero if successful, or a negative error code

int **snd_card_free_on_error**(struct device *dev, int ret)
    a small helper for handling devm probe errors

Parameters

struct device *dev
    the managed device object

int ret
    the return code from the probe callback

Description

This function handles the explicit *snd_card_free()* call at the error from the probe callback. It's just a small helper for simplifying the error handling with the managed devices.

Return

zero if successful, or a negative error code

struct snd_card ***snd_card_ref**(int idx)
    Get the card object from the index

Parameters

int idx
    the card index

Description

Returns a card object corresponding to the given index or NULL if not found. Release the object via *snd_card_unref()*.

Return

a card object or NULL

void **snd_card_disconnect**(struct snd_card *card)
    disconnect all APIs from the file-operations (user space)

Parameters

struct snd_card *card
    soundcard structure

    Disconnects all APIs from the file-operations (user space).

Return

Zero, otherwise a negative error code.

Note

The current implementation replaces all active file->f_op with special
    dummy file operations (they do nothing except release).

void **snd_card_disconnect_sync**(struct snd_card *card)
    disconnect card and wait until files get closed

Parameters

struct snd_card *card
    card object to disconnect

Description

This calls *snd_card_disconnect()* for disconnecting all belonging components and waits until all pending files get closed. It assures that all accesses from user-space finished so that the driver can release its resources gracefully.

void **snd_card_free_when_closed**(struct snd_card *card)
    Disconnect the card, free it later eventually

Parameters

struct snd_card *card
    soundcard structure

Description

Unlike *snd_card_free()*, this function doesn't try to release the card resource immediately, but tries to disconnect at first. When the card is still in use, the function returns before freeing the resources. The card resources will be freed when the refcount gets to zero.

Return

zero if successful, or a negative error code

void **snd_card_free**(struct snd_card *card)
    frees given soundcard structure

Parameters

struct snd_card *card
    soundcard structure

Description

This function releases the soundcard structure and the all assigned devices automatically. That is, you don't have to release the devices by yourself.

This function waits until the all resources are properly released.

Return

Zero. Frees all associated devices and frees the control interface associated to given soundcard.

void **snd_card_set_id**(struct snd_card *card, const char *nid)
    set card identification name

Parameters

struct snd_card *card
    soundcard structure

const char *nid
    new identification string

    This function sets the card identification and checks for name collisions.

int **snd_card_add_dev_attr**(struct snd_card *card, const struct attribute_group *group)
    Append a new sysfs attribute group to card

Parameters

struct snd_card *card
    card instance

const struct attribute_group *group
    attribute group to append

Return

zero if successful, or a negative error code

int **snd_card_register**(struct snd_card *card)

>   register the soundcard

Parameters

struct snd_card *card

>   soundcard structure

>   This function registers all the devices assigned to the soundcard. Until calling this, the ALSA control interface is blocked from the external accesses. Thus, you should call this function at the end of the initialization of the card.

Return

Zero otherwise a negative error code if the registration failed.

int **snd_component_add**(struct snd_card *card, const char *component)

>   add a component string

Parameters

struct snd_card *card

>   soundcard structure

const char *component

>   the component id string

>   This function adds the component id string to the supported list. The component can be referred from the alsa-lib.

Return

Zero otherwise a negative error code.

int **snd_card_file_add**(struct snd_card *card, struct *file* *file)

>   add the file to the file list of the card

Parameters

struct snd_card *card

>   soundcard structure

struct file *file

>   file pointer

>   This function adds the file to the file linked-list of the card. This linked-list is used to keep tracking the connection state, and to avoid the release of busy resources by hotplug.

Return

zero or a negative error code.

int **snd_card_file_remove**(struct snd_card *card, struct *file* *file)

>   remove the file from the file list

Parameters

struct snd_card *card

>   soundcard structure

struct file *file

>   file pointer

This function removes the file formerly added to the card via *snd_card_file_add()* function. If all files are removed and *snd_card_free_when_closed()* was called beforehand, it processes the pending release of resources.

Return

Zero or a negative error code.

int **snd_power_ref_and_wait**(struct snd_card *card)

   wait until the card gets powered up

Parameters

struct snd_card *card
   soundcard structure

Description

Take the power_ref reference count of the given card, and wait until the card gets powered up to SNDRV_CTL_POWER_D0 state. The refcount is down again while sleeping until power-up, hence this function can be used for syncing the floating control ops accesses, typically around calling control ops.

The caller needs to pull down the refcount via *snd_power_unref()* later no matter whether the error is returned from this function or not.

Return

Zero if successful, or a negative error code.

int **snd_power_wait**(struct snd_card *card)

   wait until the card gets powered up (old form)

Parameters

struct snd_card *card
   soundcard structure

Description

Wait until the card gets powered up to SNDRV_CTL_POWER_D0 state.

Return

Zero if successful, or a negative error code.

Device Components

int **snd_device_new**(struct snd_card *card, enum snd_device_type type, void *device_data,
                    const struct snd_device_ops *ops)

   create an ALSA device component

Parameters

struct snd_card *card
   the card instance

enum snd_device_type type
   the device type, SNDRV_DEV_XXX

void *device_data
    the data pointer of this device

const struct snd_device_ops *ops
    the operator table

Description

Creates a new device component for the given data pointer. The device will be assigned to the card and managed together by the card.

The data pointer plays a role as the identifier, too, so the pointer address must be unique and unchanged.

Return

Zero if successful, or a negative error code on failure.

void **snd_device_disconnect**(struct snd_card *card, void *device_data)
    disconnect the device

Parameters

struct snd_card *card
    the card instance

void *device_data
    the data pointer to disconnect

Description

Turns the device into the disconnection state, invoking dev_disconnect callback, if the device was already registered.

Usually called from *snd_card_disconnect()*.

Return

Zero if successful, or a negative error code on failure or if the device not found.

void **snd_device_free**(struct snd_card *card, void *device_data)
    release the device from the card

Parameters

struct snd_card *card
    the card instance

void *device_data
    the data pointer to release

Description

Removes the device from the list on the card and invokes the callbacks, dev_disconnect and dev_free, corresponding to the state. Then release the device.

int **snd_device_register**(struct snd_card *card, void *device_data)
    register the device

Parameters

struct snd_card *card
    the card instance

void *device_data
      the data pointer to register

Description

Registers the device which was already created via *snd_device_new()*. Usually this is called from *snd_card_register()*, but it can be called later if any new devices are created after invocation of *snd_card_register()*.

Return

Zero if successful, or a negative error code on failure or if the device not found.

int **snd_device_get_state**(struct snd_card *card, void *device_data)
      Get the current state of the given device

Parameters

struct snd_card *card
      the card instance

void *device_data
      the data pointer to release

Description

Returns the current state of the given device object. For the valid device, either **SNDRV_DEV_BUILD**, **SNDRV_DEV_REGISTERED** or **SNDRV_DEV_DISCONNECTED** is returned. Or for a non-existing device, -1 is returned as an error.

Return

the current state, or -1 if not found

Module requests and Device File Entries

void **snd_request_card**(int card)
      try to load the card module

Parameters

int card
      the card number

Description

Tries to load the module "snd-card-X" for the given card number via request_module. Returns immediately if already loaded.

void ***snd_lookup_minor_data**(unsigned int minor, int type)
      get user data of a registered device

Parameters

unsigned int minor
      the minor number

int type
      device type (SNDRV_DEVICE_TYPE_XXX)

Description

Checks that a minor device with the specified type is registered, and returns its user data pointer.

This function increments the reference counter of the card instance if an associated instance with the given minor number and type is found. The caller must call *snd_card_unref()* appropriately later.

Return

The user data pointer if the specified device is found. NULL otherwise.

int **snd_register_device**(int type, struct snd_card *card, int dev, const struct file_operations *f_ops, void *private_data, struct *device* *device)

    Register the ALSA device file for the card

Parameters

int type
    the device type, SNDRV_DEVICE_TYPE_XXX

struct snd_card *card
    the card instance

int dev
    the device index

const struct file_operations *f_ops
    the file operations

void *private_data
    user pointer for f_ops->open()

struct device *device
    the device to register

Description

Registers an ALSA device file for the given card. The operators have to be set in reg parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_unregister_device**(struct device *dev)
    unregister the device on the given card

Parameters

struct device *dev
    the device instance

Description

Unregisters the device file already registered via *snd_register_device()*.

Return

Zero if successful, or a negative error code on failure.

Memory Management Helpers

int **copy_to_user_fromio**(void __user *dst, volatile const void __iomem *src, size_t count)
    copy data from mmio-space to user-space

Parameters

void __user *dst
    the destination pointer on user-space

const volatile void __iomem *src
    the source pointer on mmio

size_t count
    the data size to copy in bytes

Description

Copies the data from mmio-space to user-space.

Return

Zero if successful, or non-zero on failure.

int **copy_to_iter_fromio**(struct iov_iter *dst, const void __iomem *src, size_t count)
    copy data from mmio-space to iov_iter

Parameters

struct iov_iter *dst
    the destination iov_iter

const void __iomem *src
    the source pointer on mmio

size_t count
    the data size to copy in bytes

Description

Copies the data from mmio-space to iov_iter.

Return

Zero if successful, or non-zero on failure.

int **copy_from_user_toio**(volatile void __iomem *dst, const void __user *src, size_t count)
    copy data from user-space to mmio-space

Parameters

volatile void __iomem *dst
    the destination pointer on mmio-space

const void __user *src
    the source pointer on user-space

size_t count
    the data size to copy in bytes

Description

Copies the data from user-space to mmio-space.

Return

Zero if successful, or non-zero on failure.

int **copy_from_iter_toio**(void __iomem *dst, struct iov_iter *src, size_t count)
    copy data from iov_iter to mmio-space

Parameters

void __iomem *dst
    the destination pointer on mmio-space

struct iov_iter *src
    the source iov_iter

size_t count
    the data size to copy in bytes

Description

Copies the data from iov_iter to mmio-space.

Return

Zero if successful, or non-zero on failure.

int **snd_dma_alloc_dir_pages**(int type, struct *device* *device, enum dma_data_direction dir,
                    size_t size, struct snd_dma_buffer *dmab)
    allocate the buffer area according to the given type and direction

Parameters

int type
    the DMA buffer type

struct device *device
    the device pointer

enum dma_data_direction dir
    DMA direction

size_t size
    the buffer size to allocate

struct snd_dma_buffer *dmab
    buffer allocation record to store the allocated data

Description

Calls the memory-allocator function for the corresponding buffer type.

Return

Zero if the buffer with the given size is allocated successfully, otherwise a negative value on
error.

int **snd_dma_alloc_pages_fallback**(int type, struct *device* *device, size_t size, struct
                    snd_dma_buffer *dmab)
    allocate the buffer area according to the given type with fallback

Parameters

int type
    the DMA buffer type

struct device *device
    the device pointer

size_t size
    the buffer size to allocate

struct snd_dma_buffer *dmab
    buffer allocation record to store the allocated data

Description

Calls the memory-allocator function for the corresponding buffer type. When no space is left, this function reduces the size and tries to allocate again. The size actually allocated is stored in res_size argument.

Return

Zero if the buffer with the given size is allocated successfully, otherwise a negative value on error.

void **snd_dma_free_pages**(struct snd_dma_buffer *dmab)
    release the allocated buffer

Parameters

struct snd_dma_buffer *dmab
    the buffer allocation record to release

Description

Releases the allocated buffer via snd_dma_alloc_pages().

struct snd_dma_buffer ***snd_devm_alloc_dir_pages**(struct device *dev, int type, enum
                                                    dma_data_direction dir, size_t size)
    allocate the buffer and manage with devres

Parameters

struct device *dev
    the device pointer

int type
    the DMA buffer type

enum dma_data_direction dir
    DMA direction

size_t size
    the buffer size to allocate

Description

Allocate buffer pages depending on the given type and manage using devres. The pages will be released automatically at the device removal.

Unlike snd_dma_alloc_pages(), this function requires the real device pointer, hence it can't work with SNDRV_DMA_TYPE_CONTINUOUS or SNDRV_DMA_TYPE_VMALLOC type.

Return

the snd_dma_buffer object at success, or NULL if failed

int **snd_dma_buffer_mmap**(struct snd_dma_buffer *dmab, struct vm_area_struct *area)
>   perform mmap of the given DMA buffer

Parameters

struct snd_dma_buffer *dmab
>   buffer allocation information

struct vm_area_struct *area
>   VM area information

Return

zero if successful, or a negative error code

void **snd_dma_buffer_sync**(struct snd_dma_buffer *dmab, enum snd_dma_sync_mode mode)
>   sync DMA buffer between CPU and device

Parameters

struct snd_dma_buffer *dmab
>   buffer allocation information

enum snd_dma_sync_mode mode
>   sync mode

dma_addr_t **snd_sgbuf_get_addr**(struct snd_dma_buffer *dmab, size_t offset)
>   return the physical address at the corresponding offset

Parameters

struct snd_dma_buffer *dmab
>   buffer allocation information

size_t offset
>   offset in the ring buffer

Return

the physical address

struct page ***snd_sgbuf_get_page**(struct snd_dma_buffer *dmab, size_t offset)
>   return the physical page at the corresponding offset

Parameters

struct snd_dma_buffer *dmab
>   buffer allocation information

size_t offset
>   offset in the ring buffer

Return

the page pointer

unsigned int **snd_sgbuf_get_chunk_size**(struct snd_dma_buffer *dmab, unsigned int ofs, unsigned int size)
>   compute the max chunk size with continuous pages on sg-buffer

Parameters

struct snd_dma_buffer *dmab
    buffer allocation information

unsigned int ofs
    offset in the ring buffer

unsigned int size
    the requested size

Return

the chunk size

1.1.2 PCM API

PCM Core

const char ***snd_pcm_format_name**(snd_pcm_format_t format)
    Return a name string for the given PCM format

Parameters

snd_pcm_format_t format
    PCM format

Return

the format name string

int **snd_pcm_new_stream**(struct snd_pcm *pcm, int stream, int substream_count)
    create a new PCM stream

Parameters

struct snd_pcm *pcm
    the pcm instance

int stream
    the stream direction, SNDRV_PCM_STREAM_XXX

int substream_count
    the number of substreams

Description

Creates a new stream for the pcm. The corresponding stream on the pcm must have been empty
before calling this, i.e. zero must be given to the argument of *snd_pcm_new()*.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_new**(struct snd_card *card, const char *id, int device, int playback_count, int
            capture_count, struct snd_pcm **rpcm)
    create a new PCM instance

Parameters

struct snd_card *card
    the card instance

const char *id
    the id string

int device
    the device index (zero based)

int playback_count
    the number of substreams for playback

int capture_count
    the number of substreams for capture

struct snd_pcm **rpcm
    the pointer to store the new pcm instance

Description

Creates a new PCM instance.

The pcm operators have to be set afterwards to the new instance via *snd_pcm_set_ops()*.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_new_internal**(struct snd_card *card, const char *id, int device, int playback_count, int capture_count, struct snd_pcm **rpcm)
    create a new internal PCM instance

Parameters

struct snd_card *card
    the card instance

const char *id
    the id string

int device
    the device index (zero based - shared with normal PCMs)

int playback_count
    the number of substreams for playback

int capture_count
    the number of substreams for capture

struct snd_pcm **rpcm
    the pointer to store the new pcm instance

Description

Creates a new internal PCM instance with no userspace device or procfs entries. This is used by ASoC Back End PCMs in order to create a PCM that will only be used internally by kernel drivers. i.e. it cannot be opened by userspace. It provides existing ASoC components drivers with a substream and access to any private data.

The pcm operators have to be set afterwards to the new instance via *snd_pcm_set_ops()*.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_notify**(struct *snd_pcm_notify* *notify, int nfree)

> Add/remove the notify list

Parameters

struct snd_pcm_notify *notify

> PCM notify list

int nfree

> 0 = register, 1 = unregister

Description

This adds the given notifier to the global list so that the callback is called for each registered PCM devices. This exists only for PCM OSS emulation, so far.

Return

zero if successful, or a negative error code

void **snd_pcm_set_ops**(struct snd_pcm *pcm, int direction, const struct snd_pcm_ops *ops)

> set the PCM operators

Parameters

struct snd_pcm *pcm

> the pcm instance

int direction

> stream direction, SNDRV_PCM_STREAM_XXX

const struct snd_pcm_ops *ops

> the operator table

Description

Sets the given PCM operators to the pcm instance.

void **snd_pcm_set_sync**(struct snd_pcm_substream *substream)

> set the PCM sync id

Parameters

struct snd_pcm_substream *substream

> the pcm substream

Description

Sets the PCM sync identifier for the card.

int **snd_interval_refine**(struct snd_interval *i, const struct snd_interval *v)

> refine the interval value of configurator

Parameters

struct snd_interval *i

> the interval value to refine

const struct snd_interval *v

> the interval value to refer to

Description

Refines the interval value with the reference value. The interval is changed to the range satisfying both intervals. The interval status (min, max, integer, etc.) are evaluated.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

void **snd_interval_div**(const struct snd_interval *a, const struct snd_interval *b, struct snd_interval *c)

    refine the interval value with division

Parameters

const struct snd_interval *a
    dividend

const struct snd_interval *b
    divisor

struct snd_interval *c
    quotient

Description

c = a / b

Returns non-zero if the value is changed, zero if not changed.

void **snd_interval_muldivk**(const struct snd_interval *a, const struct snd_interval *b, unsigned int k, struct snd_interval *c)

    refine the interval value

Parameters

const struct snd_interval *a
    dividend 1

const struct snd_interval *b
    dividend 2

unsigned int k
    divisor (as integer)

struct snd_interval *c
    result

Description

c = a * b / k

Returns non-zero if the value is changed, zero if not changed.

void **snd_interval_mulkdiv**(const struct snd_interval *a, unsigned int k, const struct snd_interval *b, struct snd_interval *c)

    refine the interval value

Parameters

const struct snd_interval *a
    dividend 1

unsigned int k
    dividend 2 (as integer)

const struct snd_interval *b
    divisor

struct snd_interval *c
    result

Description

c = a * k / b

Returns non-zero if the value is changed, zero if not changed.

int **snd_interval_ratnum**(struct snd_interval *i, unsigned int rats_count, const struct
                         snd_ratnum *rats, unsigned int *nump, unsigned int *denp)
    refine the interval value

Parameters

struct snd_interval *i
    interval to refine

unsigned int rats_count
    number of ratnum_t

const struct snd_ratnum *rats
    ratnum_t array

unsigned int *nump
    pointer to store the resultant numerator

unsigned int *denp
    pointer to store the resultant denominator

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_interval_ratden**(struct snd_interval *i, unsigned int rats_count, const struct
                         snd_ratden *rats, unsigned int *nump, unsigned int *denp)
    refine the interval value

Parameters

struct snd_interval *i
    interval to refine

unsigned int rats_count
    number of struct ratden

const struct snd_ratden *rats
    struct ratden array

unsigned int *nump
    pointer to store the resultant numerator

unsigned int *denp
    pointer to store the resultant denominator

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_interval_list**(struct snd_interval *i, unsigned int count, const unsigned int *list, unsigned int mask)

> refine the interval value from the list

Parameters

struct snd_interval *i
> the interval value to refine

unsigned int count
> the number of elements in the list

const unsigned int *list
> the value list

unsigned int mask
> the bit-mask to evaluate

Description

Refines the interval value from the list. When mask is non-zero, only the elements corresponding to bit 1 are evaluated.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_interval_ranges**(struct snd_interval *i, unsigned int count, const struct snd_interval *ranges, unsigned int mask)

> refine the interval value from the list of ranges

Parameters

struct snd_interval *i
> the interval value to refine

unsigned int count
> the number of elements in the list of ranges

const struct snd_interval *ranges
> the ranges list

unsigned int mask
> the bit-mask to evaluate

Description

Refines the interval value from the list of ranges. When mask is non-zero, only the elements corresponding to bit 1 are evaluated.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_hw_rule_add**(struct snd_pcm_runtime *runtime, unsigned int cond, int var, snd_pcm_hw_rule_func_t func, void *private, int dep, ...)

> add the hw-constraint rule

Parameters

struct snd_pcm_runtime *runtime
    the pcm runtime instance

unsigned int cond
    condition bits

int var
    the variable to evaluate

snd_pcm_hw_rule_func_t func
    the evaluation function

void *private
    the private data pointer passed to function

int dep
    the dependent variables

...
    variable arguments

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_mask**(struct snd_pcm_runtime *runtime, snd_pcm_hw_param_t
                                    var, u_int32_t mask)
    apply the given bitmap mask constraint

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

snd_pcm_hw_param_t var
    hw_params variable to apply the mask

u_int32_t mask
    the bitmap mask

Description

Apply the constraint of the given bitmap mask to a 32-bit mask parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_mask64**(struct snd_pcm_runtime *runtime,
                                      snd_pcm_hw_param_t var, u_int64_t mask)
    apply the given bitmap mask constraint

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

snd_pcm_hw_param_t var
    hw_params variable to apply the mask

u_int64_t mask
    the 64bit bitmap mask

Description

Apply the constraint of the given bitmap mask to a 64-bit mask parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_integer**(struct snd_pcm_runtime *runtime, snd_pcm_hw_param_t var)
    apply an integer constraint to an interval

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

snd_pcm_hw_param_t var
    hw_params variable to apply the integer constraint

Description

Apply the constraint of integer to an interval parameter.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_hw_constraint_minmax**(struct snd_pcm_runtime *runtime, snd_pcm_hw_param_t var, unsigned int min, unsigned int max)
    apply a min/max range constraint to an interval

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

snd_pcm_hw_param_t var
    hw_params variable to apply the range

unsigned int min
    the minimal value

unsigned int max
    the maximal value

Description

Apply the min/max range constraint to an interval parameter.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_hw_constraint_list**(struct snd_pcm_runtime *runtime, unsigned int cond, snd_pcm_hw_param_t var, const struct *snd_pcm_hw_constraint_list* *l)
    apply a list of constraints to a parameter

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

unsigned int cond
    condition bits

snd_pcm_hw_param_t var
    hw_params variable to apply the list constraint

const struct snd_pcm_hw_constraint_list *l
    list

Description

Apply the list of constraints to an interval parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_ranges**(struct snd_pcm_runtime *runtime, unsigned int cond,
                                snd_pcm_hw_param_t var, const struct
                                snd_pcm_hw_constraint_ranges *r)

    apply list of range constraints to a parameter

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

unsigned int cond
    condition bits

snd_pcm_hw_param_t var
    hw_params variable to apply the list of range constraints

const struct snd_pcm_hw_constraint_ranges *r
    ranges

Description

Apply the list of range constraints to an interval parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_ratnums**(struct snd_pcm_runtime *runtime, unsigned int cond,
                                snd_pcm_hw_param_t var, const struct
                                snd_pcm_hw_constraint_ratnums *r)

    apply ratnums constraint to a parameter

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

unsigned int cond
    condition bits

snd_pcm_hw_param_t var
    hw_params variable to apply the ratnums constraint

const struct snd_pcm_hw_constraint_ratnums *r
    struct snd_ratnums constriants

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_ratdens**(struct snd_pcm_runtime *runtime, unsigned int cond, snd_pcm_hw_param_t var, const struct *snd_pcm_hw_constraint_ratdens* *r)

    apply ratdens constraint to a parameter

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

unsigned int cond
    condition bits

snd_pcm_hw_param_t var
    hw_params variable to apply the ratdens constraint

const struct snd_pcm_hw_constraint_ratdens *r
    struct snd_ratdens constriants

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_msbits**(struct snd_pcm_runtime *runtime, unsigned int cond, unsigned int width, unsigned int msbits)

    add a hw constraint msbits rule

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

unsigned int cond
    condition bits

unsigned int width
    sample bits width

unsigned int msbits
    msbits width

Description

This constraint will set the number of most significant bits (msbits) if a sample format with the specified width has been select. If width is set to 0 the msbits will be set for any sample format with a width larger than the specified msbits.

Return

Zero if successful, or a negative error code on failure.

---

int **snd_pcm_hw_constraint_step**(struct snd_pcm_runtime *runtime, unsigned int cond, snd_pcm_hw_param_t var, unsigned long step)

 add a hw constraint step rule

Parameters

struct snd_pcm_runtime *runtime
 PCM runtime instance

unsigned int cond
 condition bits

snd_pcm_hw_param_t var
 hw_params variable to apply the step constraint

unsigned long step
 step size

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_pow2**(struct snd_pcm_runtime *runtime, unsigned int cond, snd_pcm_hw_param_t var)

 add a hw constraint power-of-2 rule

Parameters

struct snd_pcm_runtime *runtime
 PCM runtime instance

unsigned int cond
 condition bits

snd_pcm_hw_param_t var
 hw_params variable to apply the power-of-2 constraint

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_rule_noresample**(struct snd_pcm_runtime *runtime, unsigned int base_rate)

 add a rule to allow disabling hw resampling

Parameters

struct snd_pcm_runtime *runtime
 PCM runtime instance

unsigned int base_rate
 the rate at which the hardware does not resample

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_param_value**(const struct snd_pcm_hw_params *params, snd_pcm_hw_param_t var, int *dir)

 return **params** field **var** value

Parameters

const struct snd_pcm_hw_params *params
    the hw_params instance

snd_pcm_hw_param_t var
    parameter to retrieve

int *dir
    pointer to the direction (-1,0,1) or NULL

Return

The value for field **var** if it's fixed in configuration space defined by **params**. -EINVAL otherwise.

int **snd_pcm_hw_param_first**(struct snd_pcm_substream *pcm, struct snd_pcm_hw_params
                                *params, snd_pcm_hw_param_t var, int *dir)
    refine config space and return minimum value

Parameters

struct snd_pcm_substream *pcm
    PCM instance

struct snd_pcm_hw_params *params
    the hw_params instance

snd_pcm_hw_param_t var
    parameter to retrieve

int *dir
    pointer to the direction (-1,0,1) or NULL

Description

Inside configuration space defined by **params** remove from **var** all values > minimum. Reduce configuration space accordingly.

Return

The minimum, or a negative error code on failure.

int **snd_pcm_hw_param_last**(struct snd_pcm_substream *pcm, struct snd_pcm_hw_params
                                *params, snd_pcm_hw_param_t var, int *dir)
    refine config space and return maximum value

Parameters

struct snd_pcm_substream *pcm
    PCM instance

struct snd_pcm_hw_params *params
    the hw_params instance

snd_pcm_hw_param_t var
    parameter to retrieve

int *dir
    pointer to the direction (-1,0,1) or NULL

Description

Inside configuration space defined by **params** remove from **var** all values < maximum. Reduce configuration space accordingly.

Return

The maximum, or a negative error code on failure.

int **snd_pcm_hw_params_bits**(const struct snd_pcm_hw_params *p)
    Get the number of bits per the sample.

Parameters

const struct snd_pcm_hw_params *p
    hardware parameters

Return

The number of bits per sample based on the format, subformat and msbits the specified hw params has.

int **snd_pcm_lib_ioctl**(struct snd_pcm_substream *substream, unsigned int cmd, void *arg)
    a generic PCM ioctl callback

Parameters

struct snd_pcm_substream *substream
    the pcm substream instance

unsigned int cmd
    ioctl command

void *arg
    ioctl argument

Description

Processes the generic ioctl commands for PCM. Can be passed as the ioctl callback for PCM ops.

Return

Zero if successful, or a negative error code on failure.

void **snd_pcm_period_elapsed_under_stream_lock**(struct snd_pcm_substream *substream)
    update the status of runtime for the next period under acquired lock of PCM substream.

Parameters

struct snd_pcm_substream *substream
    the instance of pcm substream.

Description

This function is called when the batch of audio data frames as the same size as the period of buffer is already processed in audio data transmission.

The call of function updates the status of runtime with the latest position of audio data transmission, checks overrun and underrun over buffer, awaken user processes from waiting for available audio data frames, sampling audio timestamp, and performs stop or drain the PCM substream according to configured threshold.

The function is intended to use for the case that PCM driver operates audio data frames under acquired lock of PCM substream; e.g. in callback of any operation of snd_pcm_ops in process context. In any interrupt context, it's preferrable to use snd_pcm_period_elapsed() instead since lock of PCM substream should be acquired in advance.

Developer should pay enough attention that some callbacks in `snd_pcm_ops` are done by the call of function:

- .pointer - to retrieve current position of audio data transmission by frame count or XRUN state.
- .trigger - with SNDRV_PCM_TRIGGER_STOP at XRUN or DRAINING state.
- .get_time_info - to retrieve audio time stamp if needed.

Even if more than one periods have elapsed since the last call, you have to call this only once.

void **snd_pcm_period_elapsed**(struct snd_pcm_substream *substream)

> update the status of runtime for the next period by acquiring lock of PCM substream.

Parameters

struct snd_pcm_substream *substream
> the instance of PCM substream.

Description

This function is mostly similar to `snd_pcm_period_elapsed_under_stream_lock()` except for acquiring lock of PCM substream voluntarily.

It's typically called by any type of IRQ handler when hardware IRQ occurs to notify event that the batch of audio data frames as the same size as the period of buffer is already processed in audio data transmission.

int **snd_pcm_add_chmap_ctls**(struct snd_pcm *pcm, int stream, const struct snd_pcm_chmap_elem *chmap, int max_channels, unsigned long private_value, struct snd_pcm_chmap **info_ret)

> create channel-mapping control elements

Parameters

struct snd_pcm *pcm
> the assigned PCM instance

int stream
> stream direction

const struct snd_pcm_chmap_elem *chmap
> channel map elements (for query)

int max_channels
> the max number of channels for the stream

unsigned long private_value
> the value passed to each kcontrol's private_value field

struct snd_pcm_chmap **info_ret
> store struct snd_pcm_chmap instance if non-NULL

Description

Create channel-mapping control elements assigned to the given PCM stream(s).

Return

Zero if successful, or a negative error value.

void **snd_pcm_stream_lock**(struct snd_pcm_substream *substream)

> Lock the PCM stream

Parameters

`struct snd_pcm_substream *substream`

> PCM substream

Description

This locks the PCM stream's spinlock or mutex depending on the nonatomic flag of the given substream. This also takes the global link rw lock (or rw sem), too, for avoiding the race with linked streams.

void **snd_pcm_stream_unlock**(struct snd_pcm_substream *substream)

> Unlock the PCM stream

Parameters

`struct snd_pcm_substream *substream`

> PCM substream

Description

This unlocks the PCM stream that has been locked via *snd_pcm_stream_lock()*.

void **snd_pcm_stream_lock_irq**(struct snd_pcm_substream *substream)

> Lock the PCM stream

Parameters

`struct snd_pcm_substream *substream`

> PCM substream

Description

This locks the PCM stream like *snd_pcm_stream_lock()* and disables the local IRQ (only when nonatomic is false). In nonatomic case, this is identical as *snd_pcm_stream_lock()*.

void **snd_pcm_stream_unlock_irq**(struct snd_pcm_substream *substream)

> Unlock the PCM stream

Parameters

`struct snd_pcm_substream *substream`

> PCM substream

Description

This is a counter-part of *snd_pcm_stream_lock_irq()*.

void **snd_pcm_stream_unlock_irqrestore**(struct snd_pcm_substream *substream, unsigned long flags)

> Unlock the PCM stream

Parameters

`struct snd_pcm_substream *substream`

> PCM substream

`unsigned long flags`

> irq flags

Description

This is a counter-part of *snd_pcm_stream_lock_irqsave()*.

int **snd_pcm_hw_params_choose**(struct snd_pcm_substream *pcm, struct snd_pcm_hw_params *params)

>   choose a configuration defined by **params**

Parameters

struct snd_pcm_substream *pcm
>   PCM instance

struct snd_pcm_hw_params *params
>   the hw_params instance

Description

Choose one configuration from configuration space defined by **params**. The configuration chosen is that obtained fixing in this order: first access, first format, first subformat, min channels, min rate, min period time, max buffer size, min tick time

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_start**(struct snd_pcm_substream *substream)

>   start all linked streams

Parameters

struct snd_pcm_substream *substream
>   the PCM substream instance

Return

Zero if successful, or a negative error code. The stream lock must be acquired before calling this function.

int **snd_pcm_stop**(struct snd_pcm_substream *substream, snd_pcm_state_t state)

>   try to stop all running streams in the substream group

Parameters

struct snd_pcm_substream *substream
>   the PCM substream instance

snd_pcm_state_t state
>   PCM state after stopping the stream

Description

The state of each stream is then changed to the given state unconditionally.

Return

Zero if successful, or a negative error code.

int **snd_pcm_drain_done**(struct snd_pcm_substream *substream)

>   stop the DMA only when the given stream is playback

Parameters

struct snd_pcm_substream *substream
    the PCM substream

Description

After stopping, the state is changed to SETUP. Unlike *snd_pcm_stop()*, this affects only the given stream.

Return

Zero if successful, or a negative error code.

int **snd_pcm_stop_xrun**(struct snd_pcm_substream *substream)
    stop the running streams as XRUN

Parameters

struct snd_pcm_substream *substream
    the PCM substream instance

Description

This stops the given running substream (and all linked substreams) as XRUN. Unlike *snd_pcm_stop()*, this function takes the substream lock by itself.

Return

Zero if successful, or a negative error code.

int **snd_pcm_suspend_all**(struct snd_pcm *pcm)
    trigger SUSPEND to all substreams in the given pcm

Parameters

struct snd_pcm *pcm
    the PCM instance

Description

After this call, all streams are changed to SUSPENDED state.

Return

Zero if successful (or **pcm** is NULL), or a negative error code.

int **snd_pcm_prepare**(struct snd_pcm_substream *substream, struct *file* *file)
    prepare the PCM substream to be triggerable

Parameters

struct snd_pcm_substream *substream
    the PCM substream instance

struct file *file
    file to refer f_flags

Return

Zero if successful, or a negative error code.

int **snd_pcm_kernel_ioctl**(struct snd_pcm_substream *substream, unsigned int cmd, void
                            *arg)
    Execute PCM ioctl in the kernel-space

Parameters

struct snd_pcm_substream *substream
 PCM substream

unsigned int cmd
 IOCTL cmd

void *arg
 IOCTL argument

Description

The function is provided primarily for OSS layer and USB gadget drivers, and it allows only the limited set of ioctls (hw_params, sw_params, prepare, start, drain, drop, forward).

Return

zero if successful, or a negative error code

int **snd_pcm_lib_default_mmap**(struct snd_pcm_substream *substream, struct vm_area_struct *area)
 Default PCM data mmap function

Parameters

struct snd_pcm_substream *substream
 PCM substream

struct vm_area_struct *area
 VMA

Description

This is the default mmap handler for PCM data. When mmap pcm_ops is NULL, this function is invoked implicitly.

Return

zero if successful, or a negative error code

int **snd_pcm_lib_mmap_iomem**(struct snd_pcm_substream *substream, struct vm_area_struct *area)
 Default PCM data mmap function for I/O mem

Parameters

struct snd_pcm_substream *substream
 PCM substream

struct vm_area_struct *area
 VMA

Description

When your hardware uses the iomapped pages as the hardware buffer and wants to mmap it, pass this function as mmap pcm_ops. Note that this is supposed to work only on limited architectures.

Return

zero if successful, or a negative error code

int **snd_pcm_stream_linked**(struct snd_pcm_substream *substream)

> Check whether the substream is linked with others

Parameters

struct snd_pcm_substream *substream

> substream to check

Return

true if the given substream is being linked with others

snd_pcm_stream_lock_irqsave

snd_pcm_stream_lock_irqsave (substream, flags)

> Lock the PCM stream

Parameters

substream

> PCM substream

flags

> irq flags

Description

This locks the PCM stream like *snd_pcm_stream_lock()* but with the local IRQ (only when nonatomic is false). In nonatomic case, this is identical as *snd_pcm_stream_lock()*.

snd_pcm_stream_lock_irqsave_nested

snd_pcm_stream_lock_irqsave_nested (substream, flags)

> Single-nested PCM stream locking

Parameters

substream

> PCM substream

flags

> irq flags

Description

This locks the PCM stream like *snd_pcm_stream_lock_irqsave()* but with the single-depth lockdep subclass.

snd_pcm_group_for_each_entry

snd_pcm_group_for_each_entry (s, substream)

> iterate over the linked substreams

Parameters

s

> the iterator

substream

> the substream

Description

Iterate over the all linked substreams to the given **substream**. When **substream** isn't linked with any others, this gives returns **substream** itself once.

int **snd_pcm_running**(struct snd_pcm_substream *substream)
    Check whether the substream is in a running state

Parameters

struct snd_pcm_substream *substream
    substream to check

Return

true if the given substream is in the state RUNNING, or in the state DRAINING for playback.

void **__snd_pcm_set_state**(struct snd_pcm_runtime *runtime, snd_pcm_state_t state)
    Change the current PCM state

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime to set

snd_pcm_state_t state
    the current state to set

Description

Call within the stream lock

ssize_t **bytes_to_samples**(struct snd_pcm_runtime *runtime, ssize_t size)
    Unit conversion of the size from bytes to samples

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

ssize_t size
    size in bytes

Return

the size in samples

snd_pcm_sframes_t **bytes_to_frames**(struct snd_pcm_runtime *runtime, ssize_t size)
    Unit conversion of the size from bytes to frames

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

ssize_t size
    size in bytes

Return

the size in frames

ssize_t **samples_to_bytes**(struct snd_pcm_runtime *runtime, ssize_t size)

> Unit conversion of the size from samples to bytes

Parameters

struct snd_pcm_runtime *runtime

> PCM runtime instance

ssize_t size

> size in samples

Return

the byte size

ssize_t **frames_to_bytes**(struct snd_pcm_runtime *runtime, snd_pcm_sframes_t size)

> Unit conversion of the size from frames to bytes

Parameters

struct snd_pcm_runtime *runtime

> PCM runtime instance

snd_pcm_sframes_t size

> size in frames

Return

the byte size

int **frame_aligned**(struct snd_pcm_runtime *runtime, ssize_t bytes)

> Check whether the byte size is aligned to frames

Parameters

struct snd_pcm_runtime *runtime

> PCM runtime instance

ssize_t bytes

> size in bytes

Return

true if aligned, or false if not

size_t **snd_pcm_lib_buffer_bytes**(struct snd_pcm_substream *substream)

> Get the buffer size of the current PCM in bytes

Parameters

struct snd_pcm_substream *substream

> PCM substream

Return

buffer byte size

size_t **snd_pcm_lib_period_bytes**(struct snd_pcm_substream *substream)

> Get the period size of the current PCM in bytes

Parameters

struct snd_pcm_substream *substream
    PCM substream

Return

period byte size

snd_pcm_uframes_t **snd_pcm_playback_avail**(struct snd_pcm_runtime *runtime)
    Get the available (writable) space for playback

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

Description

Result is between 0 ... (boundary - 1)

Return

available frame size

snd_pcm_uframes_t **snd_pcm_capture_avail**(struct snd_pcm_runtime *runtime)
    Get the available (readable) space for capture

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

Description

Result is between 0 ... (boundary - 1)

Return

available frame size

snd_pcm_sframes_t **snd_pcm_playback_hw_avail**(struct snd_pcm_runtime *runtime)
    Get the queued space for playback

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

Return

available frame size

snd_pcm_sframes_t **snd_pcm_capture_hw_avail**(struct snd_pcm_runtime *runtime)
    Get the free space for capture

Parameters

struct snd_pcm_runtime *runtime
    PCM runtime instance

Return

available frame size

int **snd_pcm_playback_ready**(struct snd_pcm_substream *substream)

    check whether the playback buffer is available

Parameters

`struct snd_pcm_substream *substream`

    the pcm substream instance

Description

Checks whether enough free space is available on the playback buffer.

Return

Non-zero if available, or zero if not.

int **snd_pcm_capture_ready**(struct snd_pcm_substream *substream)

    check whether the capture buffer is available

Parameters

`struct snd_pcm_substream *substream`

    the pcm substream instance

Description

Checks whether enough capture data is available on the capture buffer.

Return

Non-zero if available, or zero if not.

int **snd_pcm_playback_data**(struct snd_pcm_substream *substream)

    check whether any data exists on the playback buffer

Parameters

`struct snd_pcm_substream *substream`

    the pcm substream instance

Description

Checks whether any data exists on the playback buffer.

Return

Non-zero if any data exists, or zero if not. If stop_threshold is bigger or equal to boundary, then this function returns always non-zero.

int **snd_pcm_playback_empty**(struct snd_pcm_substream *substream)

    check whether the playback buffer is empty

Parameters

`struct snd_pcm_substream *substream`

    the pcm substream instance

Description

Checks whether the playback buffer is empty.

Return

Non-zero if empty, or zero if not.

int **snd_pcm_capture_empty**(struct snd_pcm_substream *substream)

   check whether the capture buffer is empty

Parameters

struct snd_pcm_substream *substream
   the pcm substream instance

Description

Checks whether the capture buffer is empty.

Return

Non-zero if empty, or zero if not.

void **snd_pcm_trigger_done**(struct snd_pcm_substream *substream, struct
                           snd_pcm_substream *master)

   Mark the master substream

Parameters

struct snd_pcm_substream *substream
   the pcm substream instance

struct snd_pcm_substream *master
   the linked master substream

Description

When multiple substreams of the same card are linked and the hardware supports the single-shot operation, the driver calls this in the loop in *snd_pcm_group_for_each_entry()* for marking the substream as "done". Then most of trigger operations are performed only to the given master substream.

The trigger_master mark is cleared at timestamp updates at the end of trigger operations.

unsigned int **params_channels**(const struct snd_pcm_hw_params *p)

   Get the number of channels from the hw params

Parameters

const struct snd_pcm_hw_params *p
   hw params

Return

the number of channels

unsigned int **params_rate**(const struct snd_pcm_hw_params *p)

   Get the sample rate from the hw params

Parameters

const struct snd_pcm_hw_params *p
   hw params

Return

the sample rate

---

unsigned int **params_period_size**(const struct snd_pcm_hw_params *p)

    Get the period size (in frames) from the hw params

Parameters

const struct snd_pcm_hw_params *p

    hw params

Return

the period size in frames

unsigned int **params_periods**(const struct snd_pcm_hw_params *p)

    Get the number of periods from the hw params

Parameters

const struct snd_pcm_hw_params *p

    hw params

Return

the number of periods

unsigned int **params_buffer_size**(const struct snd_pcm_hw_params *p)

    Get the buffer size (in frames) from the hw params

Parameters

const struct snd_pcm_hw_params *p

    hw params

Return

the buffer size in frames

unsigned int **params_buffer_bytes**(const struct snd_pcm_hw_params *p)

    Get the buffer size (in bytes) from the hw params

Parameters

const struct snd_pcm_hw_params *p

    hw params

Return

the buffer size in bytes

int **snd_pcm_hw_constraint_single**(struct snd_pcm_runtime *runtime,
                                snd_pcm_hw_param_t var, unsigned int val)

    Constrain parameter to a single value

Parameters

struct snd_pcm_runtime *runtime

    PCM runtime instance

snd_pcm_hw_param_t var

    The hw_params variable to constrain

unsigned int val

    The value to constrain to

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_format_cpu_endian**(snd_pcm_format_t format)

> Check the PCM format is CPU-endian

Parameters

snd_pcm_format_t format
> the format to check

Return

1 if the given PCM format is CPU-endian, 0 if opposite, or a negative error code if endian not specified.

void **snd_pcm_set_runtime_buffer**(struct snd_pcm_substream *substream, struct snd_dma_buffer *bufp)

> Set the PCM runtime buffer

Parameters

struct snd_pcm_substream *substream
> PCM substream to set

struct snd_dma_buffer *bufp
> the buffer information, NULL to clear

Description

Copy the buffer information to runtime->dma_buffer when **bufp** is non-NULL. Otherwise it clears the current buffer information.

void **snd_pcm_gettime**(struct snd_pcm_runtime *runtime, struct timespec64 *tv)

> Fill the timespec64 depending on the timestamp mode

Parameters

struct snd_pcm_runtime *runtime
> PCM runtime instance

struct timespec64 *tv
> timespec64 to fill

int **snd_pcm_set_fixed_buffer**(struct snd_pcm_substream *substream, int type, struct device *data, size_t size)

> Preallocate and set up the fixed size PCM buffer

Parameters

struct snd_pcm_substream *substream
> the pcm substream instance

int type
> DMA type (SNDRV_DMA_TYPE_*)

struct device *data
> DMA type dependent data

size_t size
    the requested pre-allocation size in bytes

Description

This is a variant of *snd_pcm_set_managed_buffer()*, but this pre-allocates only the given sized buffer and doesn't allow re-allocation nor dynamic allocation of a larger buffer unlike the standard one. The function may return -ENOMEM error, hence the caller must check it.

Return

zero if successful, or a negative error code

int **snd_pcm_set_fixed_buffer_all**(struct snd_pcm *pcm, int type, struct device *data, size_t size)
    Preallocate and set up the fixed size PCM buffer

Parameters

struct snd_pcm *pcm
    the pcm instance

int type
    DMA type (SNDRV_DMA_TYPE_*)

struct device *data
    DMA type dependent data

size_t size
    the requested pre-allocation size in bytes

Description

Apply the set up of the fixed buffer via *snd_pcm_set_fixed_buffer()* for all substream. If any of allocation fails, it returns -ENOMEM, hence the caller must check the return value.

Return

zero if successful, or a negative error code

int **snd_pcm_lib_alloc_vmalloc_buffer**(struct snd_pcm_substream *substream, size_t size)
    allocate virtual DMA buffer

Parameters

struct snd_pcm_substream *substream
    the substream to allocate the buffer to

size_t size
    the requested buffer size, in bytes

Description

Allocates the PCM substream buffer using vmalloc(), i.e., the memory is contiguous in kernel virtual space, but not in physical memory. Use this if the buffer is accessed by kernel code but not by device DMA.

Return

1 if the buffer was changed, 0 if not changed, or a negative error code.

int **snd_pcm_lib_alloc_vmalloc_32_buffer**(struct snd_pcm_substream *substream, size_t size)

  allocate 32-bit-addressable buffer

Parameters

struct snd_pcm_substream *substream
  the substream to allocate the buffer to

size_t size
  the requested buffer size, in bytes

Description

This function works like *snd_pcm_lib_alloc_vmalloc_buffer()*, but uses vmalloc_32(), i.e., the pages are allocated from 32-bit-addressable memory.

Return

1 if the buffer was changed, 0 if not changed, or a negative error code.

dma_addr_t **snd_pcm_sgbuf_get_addr**(struct snd_pcm_substream *substream, unsigned int ofs)

  Get the DMA address at the corresponding offset

Parameters

struct snd_pcm_substream *substream
  PCM substream

unsigned int ofs
  byte offset

Return

DMA address

unsigned int **snd_pcm_sgbuf_get_chunk_size**(struct snd_pcm_substream *substream, unsigned int ofs, unsigned int size)

  Compute the max size that fits within the contig. page from the given size

Parameters

struct snd_pcm_substream *substream
  PCM substream

unsigned int ofs
  byte offset

unsigned int size
  byte size to examine

Return

chunk size

void **snd_pcm_mmap_data_open**(struct vm_area_struct *area)

  increase the mmap counter

Parameters

struct vm_area_struct *area
    VMA

Description

PCM mmap callback should handle this counter properly

void **snd_pcm_mmap_data_close**(struct vm_area_struct *area)
    decrease the mmap counter

Parameters

struct vm_area_struct *area
    VMA

Description

PCM mmap callback should handle this counter properly

void **snd_pcm_limit_isa_dma_size**(int dma, size_t *max)
    Get the max size fitting with ISA DMA transfer

Parameters

int dma
    DMA number

size_t *max
    pointer to store the max size

const char ***snd_pcm_direction_name**(int direction)
    Get a string naming the direction of a stream

Parameters

int direction
    Stream's direction, one of SNDRV_PCM_STREAM_XXX

Description

Returns a string naming the direction of the stream.

const char ***snd_pcm_stream_str**(struct snd_pcm_substream *substream)
    Get a string naming the direction of a stream

Parameters

struct snd_pcm_substream *substream
    the pcm substream instance

Return

A string naming the direction of the stream.

struct snd_pcm_substream ***snd_pcm_chmap_substream**(struct snd_pcm_chmap *info,
                                                    unsigned int idx)
    get the PCM substream assigned to the given chmap info

Parameters

struct snd_pcm_chmap *info
    chmap information

unsigned int idx
      the substream number index

Return

the matched PCM substream, or NULL if not found

u64 **pcm_format_to_bits**(snd_pcm_format_t pcm_format)
      Strong-typed conversion of pcm_format to bitwise

Parameters

snd_pcm_format_t pcm_format
      PCM format

Return

64bit mask corresponding to the given PCM format

pcm_for_each_format

pcm_for_each_format (f)

      helper to iterate for each format type

Parameters

f
      the iterator variable in snd_pcm_format_t type

PCM Format Helpers

int **snd_pcm_format_signed**(snd_pcm_format_t format)
      Check the PCM format is signed linear

Parameters

snd_pcm_format_t format
      the format to check

Return

1 if the given PCM format is signed linear, 0 if unsigned linear, and a negative error code for
non-linear formats.

int **snd_pcm_format_unsigned**(snd_pcm_format_t format)
      Check the PCM format is unsigned linear

Parameters

snd_pcm_format_t format
      the format to check

Return

1 if the given PCM format is unsigned linear, 0 if signed linear, and a negative error code for
non-linear formats.

int **snd_pcm_format_linear**(snd_pcm_format_t format)

   Check the PCM format is linear

Parameters

snd_pcm_format_t format

   the format to check

Return

1 if the given PCM format is linear, 0 if not.

int **snd_pcm_format_little_endian**(snd_pcm_format_t format)

   Check the PCM format is little-endian

Parameters

snd_pcm_format_t format

   the format to check

Return

1 if the given PCM format is little-endian, 0 if big-endian, or a negative error code if endian not specified.

int **snd_pcm_format_big_endian**(snd_pcm_format_t format)

   Check the PCM format is big-endian

Parameters

snd_pcm_format_t format

   the format to check

Return

1 if the given PCM format is big-endian, 0 if little-endian, or a negative error code if endian not specified.

int **snd_pcm_format_width**(snd_pcm_format_t format)

   return the bit-width of the format

Parameters

snd_pcm_format_t format

   the format to check

Return

The bit-width of the format, or a negative error code if unknown format.

int **snd_pcm_format_physical_width**(snd_pcm_format_t format)

   return the physical bit-width of the format

Parameters

snd_pcm_format_t format

   the format to check

Return

The physical bit-width of the format, or a negative error code if unknown format.

ssize_t **snd_pcm_format_size**(snd_pcm_format_t format, size_t samples)
>   return the byte size of samples on the given format

Parameters

snd_pcm_format_t format
>   the format to check

size_t samples
>   sampling rate

Return

The byte size of the given samples for the format, or a negative error code if unknown format.

const unsigned char ***snd_pcm_format_silence_64**(snd_pcm_format_t format)
>   return the silent data in 8 bytes array

Parameters

snd_pcm_format_t format
>   the format to check

Return

The format pattern to fill or NULL if error.

int **snd_pcm_format_set_silence**(snd_pcm_format_t format, void *data, unsigned int
>   samples)
>   set the silence data on the buffer

Parameters

snd_pcm_format_t format
>   the PCM format

void *data
>   the buffer pointer

unsigned int samples
>   the number of samples to set silence

Description

Sets the silence data on the buffer for the given samples.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_limit_rates**(struct snd_pcm_hardware *hw)
>   determine rate_min/rate_max fields

Parameters

struct snd_pcm_hardware *hw
>   the pcm hw instance

Description

Determines the rate_min and rate_max fields from the rates bits of the given hw.

Return

Zero if successful.

unsigned int **snd_pcm_rate_to_rate_bit**(unsigned int rate)
    converts sample rate to SNDRV_PCM_RATE_xxx bit

Parameters

unsigned int rate
    the sample rate to convert

Return

The SNDRV_PCM_RATE_xxx flag that corresponds to the given rate, or SNDRV_PCM_RATE_KNOT for an unknown rate.

unsigned int **snd_pcm_rate_bit_to_rate**(unsigned int rate_bit)
    converts SNDRV_PCM_RATE_xxx bit to sample rate

Parameters

unsigned int rate_bit
    the rate bit to convert

Return

The sample rate that corresponds to the given SNDRV_PCM_RATE_xxx flag or 0 for an unknown rate bit.

unsigned int **snd_pcm_rate_mask_intersect**(unsigned int rates_a, unsigned int rates_b)
    computes the intersection between two rate masks

Parameters

unsigned int rates_a
    The first rate mask

unsigned int rates_b
    The second rate mask

Description

This function computes the rates that are supported by both rate masks passed to the function. It will take care of the special handling of SNDRV_PCM_RATE_CONTINUOUS and SNDRV_PCM_RATE_KNOT.

Return

A rate mask containing the rates that are supported by both rates_a and rates_b.

unsigned int **snd_pcm_rate_range_to_bits**(unsigned int rate_min, unsigned int rate_max)
    converts rate range to SNDRV_PCM_RATE_xxx bit

Parameters

unsigned int rate_min
    the minimum sample rate

unsigned int rate_max
    the maximum sample rate

Description

This function has an implicit assumption: the rates in the given range have only the pre-defined rates like 44100 or 16000.

Return

The SNDRV_PCM_RATE_xxx flag that corresponds to the given rate range, or SNDRV_PCM_RATE_KNOT for an unknown range.

PCM Memory Management

void **snd_pcm_lib_preallocate_free**(struct snd_pcm_substream *substream)
> release the preallocated buffer of the specified substream.

Parameters

struct snd_pcm_substream *substream
> the pcm substream instance

Description

Releases the pre-allocated buffer of the given substream.

void **snd_pcm_lib_preallocate_free_for_all**(struct snd_pcm *pcm)
> release all pre-allocated buffers on the pcm

Parameters

struct snd_pcm *pcm
> the pcm instance

Description

Releases all the pre-allocated buffers on the given pcm.

void **snd_pcm_lib_preallocate_pages**(struct snd_pcm_substream *substream, int type,
struct device *data, size_t size, size_t max)
> pre-allocation for the given DMA type

Parameters

struct snd_pcm_substream *substream
> the pcm substream instance

int type
> DMA type (SNDRV_DMA_TYPE_*)

struct device *data
> DMA type dependent data

size_t size
> the requested pre-allocation size in bytes

size_t max
> the max. allowed pre-allocation size

Description

Do pre-allocation for the given DMA buffer type.

---

void **snd_pcm_lib_preallocate_pages_for_all**(struct snd_pcm *pcm, int type, void *data, size_t size, size_t max)

> pre-allocation for continuous memory type (all substreams)

Parameters

struct snd_pcm *pcm
> the pcm instance

int type
> DMA type (SNDRV_DMA_TYPE_*)

void *data
> DMA type dependent data

size_t size
> the requested pre-allocation size in bytes

size_t max
> the max. allowed pre-allocation size

Description

Do pre-allocation to all substreams of the given pcm for the specified DMA type.

int **snd_pcm_set_managed_buffer**(struct snd_pcm_substream *substream, int type, struct device *data, size_t size, size_t max)

> set up buffer management for a substream

Parameters

struct snd_pcm_substream *substream
> the pcm substream instance

int type
> DMA type (SNDRV_DMA_TYPE_*)

struct device *data
> DMA type dependent data

size_t size
> the requested pre-allocation size in bytes

size_t max
> the max. allowed pre-allocation size

Description

Do pre-allocation for the given DMA buffer type, and set the managed buffer allocation mode to the given substream. In this mode, PCM core will allocate a buffer automatically before PCM hw_params ops call, and release the buffer after PCM hw_free ops call as well, so that the driver doesn't need to invoke the allocation and the release explicitly in its callback. When a buffer is actually allocated before the PCM hw_params call, it turns on the runtime buffer_changed flag for drivers changing their h/w parameters accordingly.

When **size** is non-zero and **max** is zero, this tries to allocate for only the exact buffer size without fallback, and may return -ENOMEM. Otherwise, the function tries to allocate smaller chunks if the allocation fails. This is the behavior of *snd_pcm_set_fixed_buffer()*.

When both **size** and **max** are zero, the function only sets up the buffer for later dynamic allocations. It's used typically for buffers with SNDRV_DMA_TYPE_VMALLOC type.

Upon successful buffer allocation and setup, the function returns 0.

Return

zero if successful, or a negative error code

int **snd_pcm_set_managed_buffer_all**(struct snd_pcm *pcm, int type, struct device *data, size_t size, size_t max)

    set up buffer management for all substreams for all substreams

Parameters

struct snd_pcm *pcm
    the pcm instance

int type
    DMA type (SNDRV_DMA_TYPE_*)

struct device *data
    DMA type dependent data

size_t size
    the requested pre-allocation size in bytes

size_t max
    the max. allowed pre-allocation size

Description

Do pre-allocation to all substreams of the given pcm for the specified DMA type and size, and set the managed_buffer_alloc flag to each substream.

Return

zero if successful, or a negative error code

int **snd_pcm_lib_malloc_pages**(struct snd_pcm_substream *substream, size_t size)
    allocate the DMA buffer

Parameters

struct snd_pcm_substream *substream
    the substream to allocate the DMA buffer to

size_t size
    the requested buffer size in bytes

Description

Allocates the DMA buffer on the BUS type given earlier to snd_pcm_lib_preallocate_xxx_pages().

Return

1 if the buffer is changed, 0 if not changed, or a negative code on failure.

int **snd_pcm_lib_free_pages**(struct snd_pcm_substream *substream)
    release the allocated DMA buffer.

Parameters

struct snd_pcm_substream *substream
    the substream to release the DMA buffer

Description

Releases the DMA buffer allocated via *snd_pcm_lib_malloc_pages()*.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_lib_free_vmalloc_buffer**(struct snd_pcm_substream *substream)
    free vmalloc buffer

Parameters

struct snd_pcm_substream *substream
    the substream with a buffer allocated by *snd_pcm_lib_alloc_vmalloc_buffer()*

Return

Zero if successful, or a negative error code on failure.

struct page ***snd_pcm_lib_get_vmalloc_page**(struct snd_pcm_substream *substream,
                                          unsigned long offset)
    map vmalloc buffer offset to page struct

Parameters

struct snd_pcm_substream *substream
    the substream with a buffer allocated by *snd_pcm_lib_alloc_vmalloc_buffer()*

unsigned long offset
    offset in the buffer

Description

This function is to be used as the page callback in the PCM ops.

Return

The page struct, or NULL on failure.

PCM DMA Engine API

int **snd_hwparams_to_dma_slave_config**(const struct snd_pcm_substream *substream, const
                                     struct snd_pcm_hw_params *params, struct
                                     dma_slave_config *slave_config)
    Convert hw_params to dma_slave_config

Parameters

const struct snd_pcm_substream *substream
    PCM substream

const struct snd_pcm_hw_params *params
    hw_params

struct dma_slave_config *slave_config
    DMA slave config

Description

This function can be used to initialize a dma_slave_config from a substream and hw_params in a dmaengine based PCM driver implementation.

Return

zero if successful, or a negative error code

void **snd_dmaengine_pcm_set_config_from_dai_data**(const struct snd_pcm_substream *substream, const struct *snd_dmaengine_dai_dma_data* *dma_data, struct dma_slave_config *slave_config)

 Initializes a dma slave config using DAI DMA data.

Parameters

`const struct snd_pcm_substream *substream`
 PCM substream

`const struct snd_dmaengine_dai_dma_data *dma_data`
 DAI DMA data

`struct dma_slave_config *slave_config`
 DMA slave configuration

Description

Initializes the {dst,src}_addr, {dst,src}_maxburst, {dst,src}_addr_width fields of the DMA slave config from the same fields of the DAI DMA data struct. The src and dst fields will be initialized depending on the direction of the substream. If the substream is a playback stream the dst fields will be initialized, if it is a capture stream the src fields will be initialized. The {dst,src}_addr_width field will only be initialized if the SND_DMAENGINE_PCM_DAI_FLAG_PACK flag is set or if the addr_width field of the DAI DMA data struct is not equal to DMA_SLAVE_BUSWIDTH_UNDEFINED. If both conditions are met the latter takes priority.

int **snd_dmaengine_pcm_trigger**(struct snd_pcm_substream *substream, int cmd)
 dmaengine based PCM trigger implementation

Parameters

`struct snd_pcm_substream *substream`
 PCM substream

`int cmd`
 Trigger command

Description

This function can be used as the PCM trigger callback for dmaengine based PCM driver implementations.

Return

0 on success, a negative error code otherwise

snd_pcm_uframes_t **snd_dmaengine_pcm_pointer_no_residue**(struct snd_pcm_substream *substream)

dmaengine based PCM pointer implementation

Parameters

`struct snd_pcm_substream *substream`
    PCM substream

Description

This function is deprecated and should not be used by new drivers, as its results may be unreliable.

Return

PCM position in frames

snd_pcm_uframes_t **snd_dmaengine_pcm_pointer**(struct snd_pcm_substream *substream)
    dmaengine based PCM pointer implementation

Parameters

`struct snd_pcm_substream *substream`
    PCM substream

Description

This function can be used as the PCM pointer callback for dmaengine based PCM driver implementations.

Return

PCM position in frames

struct dma_chan ***snd_dmaengine_pcm_request_channel**(dma_filter_fn filter_fn, void *filter_data)
    Request channel for the dmaengine PCM

Parameters

`dma_filter_fn filter_fn`
    Filter function used to request the DMA channel

`void *filter_data`
    Data passed to the DMA filter function

Description

This function request a DMA channel for usage with dmaengine PCM.

Return

NULL or the requested DMA channel

int **snd_dmaengine_pcm_open**(struct snd_pcm_substream *substream, struct dma_chan *chan)
    Open a dmaengine based PCM substream

Parameters

`struct snd_pcm_substream *substream`
    PCM substream

struct dma_chan *chan
    DMA channel to use for data transfers

Description

The function should usually be called from the pcm open callback. Note that this function will use private_data field of the substream's runtime. So it is not available to your pcm driver implementation.

Return

0 on success, a negative error code otherwise

int **snd_dmaengine_pcm_open_request_chan**(struct snd_pcm_substream *substream, dma_filter_fn filter_fn, void *filter_data)
    Open a dmaengine based PCM substream and request channel

Parameters

struct snd_pcm_substream *substream
    PCM substream

dma_filter_fn filter_fn
    Filter function used to request the DMA channel

void *filter_data
    Data passed to the DMA filter function

Description

This function will request a DMA channel using the passed filter function and data. The function should usually be called from the pcm open callback. Note that this function will use private_data field of the substream's runtime. So it is not available to your pcm driver implementation.

Return

0 on success, a negative error code otherwise

int **snd_dmaengine_pcm_close**(struct snd_pcm_substream *substream)
    Close a dmaengine based PCM substream

Parameters

struct snd_pcm_substream *substream
    PCM substream

Return

0 on success, a negative error code otherwise

int **snd_dmaengine_pcm_close_release_chan**(struct snd_pcm_substream *substream)
    Close a dmaengine based PCM substream and release channel

Parameters

struct snd_pcm_substream *substream
    PCM substream

Description

Releases the DMA channel associated with the PCM substream.

Return

zero if successful, or a negative error code

int **snd_dmaengine_pcm_refine_runtime_hwparams**(struct snd_pcm_substream *substream,
struct *snd_dmaengine_dai_dma_data*
*dma_data, struct snd_pcm_hardware
*hw, struct dma_chan *chan)

> Refine runtime hw params

Parameters

struct snd_pcm_substream *substream
> PCM substream

struct snd_dmaengine_dai_dma_data *dma_data
> DAI DMA data

struct snd_pcm_hardware *hw
> PCM hw params

struct dma_chan *chan
> DMA channel to use for data transfers

Description

This function will query DMA capability, then refine the pcm hardware parameters.

Return

0 on success, a negative error code otherwise

enum dma_transfer_direction **snd_pcm_substream_to_dma_direction**(const struct
snd_pcm_substream
*substream)

> Get dma_transfer_direction for a PCM substream

Parameters

const struct snd_pcm_substream *substream
> PCM substream

Return

DMA transfer direction

struct **snd_dmaengine_dai_dma_data**
> DAI DMA configuration data

Definition:

```
struct snd_dmaengine_dai_dma_data {
    dma_addr_t addr;
    enum dma_slave_buswidth addr_width;
    u32 maxburst;
    void *filter_data;
    const char *chan_name;
    unsigned int fifo_size;
    unsigned int flags;
```

```
    void *peripheral_config;
    size_t peripheral_size;
};
```

Members

addr
    Address of the DAI data source or destination register.

addr_width
    Width of the DAI data source or destination register.

maxburst
    Maximum number of words(note: words, as in units of the src_addr_width member, not bytes) that can be send to or received from the DAI in one burst.

filter_data
    Custom DMA channel filter data, this will usually be used when requesting the DMA channel.

chan_name
    Custom channel name to use when requesting DMA channel.

fifo_size
    FIFO size of the DAI controller in bytes

flags
    PCM_DAI flags, only SND_DMAENGINE_PCM_DAI_FLAG_PACK for now

peripheral_config
    peripheral configuration for programming peripheral for dmaengine transfer

peripheral_size
    peripheral configuration buffer size

struct **snd_dmaengine_pcm_config**
    Configuration data for dmaengine based PCM

Definition:

```
struct snd_dmaengine_pcm_config {
    int (*prepare_slave_config)(struct snd_pcm_substream *substream,struct snd_
↪pcm_hw_params *params, struct dma_slave_config *slave_config);
    struct dma_chan *(*compat_request_channel)(struct snd_soc_pcm_runtime *rtd,
↪ struct snd_pcm_substream *substream);
    int (*process)(struct snd_pcm_substream *substream,int channel, unsigned␣
↪long hwoff, unsigned long bytes);
    dma_filter_fn compat_filter_fn;
    struct device *dma_dev;
    const char *chan_names[SNDRV_PCM_STREAM_LAST + 1];
    const struct snd_pcm_hardware *pcm_hardware;
    unsigned int prealloc_buffer_size;
};
```

Members

prepare_slave_config
> Callback used to fill in the DMA slave_config for a PCM substream. Will be called from the PCM drivers hwparams callback.

compat_request_channel
> Callback to request a DMA channel for platforms which do not use devicetree.

process
> Callback used to apply processing on samples transferred from/to user space.

compat_filter_fn
> Will be used as the filter function when requesting a channel for platforms which do not use devicetree. The filter parameter will be the DAI's DMA data.

dma_dev
> If set, request DMA channel on this device rather than the DAI device.

chan_names
> If set, these custom DMA channel names will be requested at registration time.

pcm_hardware
> snd_pcm_hardware struct to be used for the PCM.

prealloc_buffer_size
> Size of the preallocated audio buffer.

Note

If both compat_request_channel and compat_filter_fn are set compat_request_channel will be used to request the channel and compat_filter_fn will be ignored. Otherwise the channel will be requested using dma_request_channel with compat_filter_fn as the filter function.

1.1.3 Control/Mixer API

General Control Interface

void **snd_ctl_notify**(struct snd_card *card, unsigned int mask, struct snd_ctl_elem_id *id)
> Send notification to user-space for a control change

Parameters

struct snd_card *card
> the card to send notification

unsigned int mask
> the event mask, SNDRV_CTL_EVENT_*

struct snd_ctl_elem_id *id
> the ctl element id to send notification

Description

This function adds an event record with the given id and mask, appends to the list and wakes up the user-space for notification. This can be called in the atomic context.

void **snd_ctl_notify_one**(struct snd_card *card, unsigned int mask, struct snd_kcontrol *kctl, unsigned int ioff)
> Send notification to user-space for a control change

Parameters

struct snd_card *card
    the card to send notification

unsigned int mask
    the event mask, SNDRV_CTL_EVENT_*

struct snd_kcontrol *kctl
    the pointer with the control instance

unsigned int ioff
    the additional offset to the control index

Description

This function calls *snd_ctl_notify()* and does additional jobs like LED state changes.

int **snd_ctl_new**(struct snd_kcontrol **kctl, unsigned int count, unsigned int access, struct
                snd_ctl_file *file)

    create a new control instance with some elements

Parameters

struct snd_kcontrol **kctl
    the pointer to store new control instance

unsigned int count
    the number of elements in this control

unsigned int access
    the default access flags for elements in this control

struct snd_ctl_file *file
    given when locking these elements

Description

Allocates a memory object for a new control instance. The instance has elements as many as
the given number (**count**). Each element has given access permissions (**access**). Each element
is locked when **file** is given.

Return

0 on success, error code on failure

struct snd_kcontrol ***snd_ctl_new1**(const struct snd_kcontrol_new *ncontrol, void
                                    *private_data)

    create a control instance from the template

Parameters

const struct snd_kcontrol_new *ncontrol
    the initialization record

void *private_data
    the private data to set

Description

Allocates a new struct snd_kcontrol instance and initialize from the given template. When the access field of ncontrol is 0, it's assumed as READWRITE access. When the count field is 0, it's assumes as one.

Return

The pointer of the newly generated instance, or NULL on failure.

void **snd_ctl_free_one**(struct snd_kcontrol *kcontrol)
    release the control instance

Parameters

struct snd_kcontrol *kcontrol
    the control instance

Description

Releases the control instance created via *snd_ctl_new()* or *snd_ctl_new1()*. Don't call this after the control was added to the card.

int **snd_ctl_add**(struct snd_card *card, struct snd_kcontrol *kcontrol)
    add the control instance to the card

Parameters

struct snd_card *card
    the card instance

struct snd_kcontrol *kcontrol
    the control instance to add

Description

Adds the control instance created via *snd_ctl_new()* or *snd_ctl_new1()* to the given card. Assigns also an unique numid used for fast search.

It frees automatically the control which cannot be added.

Return

Zero if successful, or a negative error code on failure.

int **snd_ctl_replace**(struct snd_card *card, struct snd_kcontrol *kcontrol, bool
                        add_on_replace)
    replace the control instance of the card

Parameters

struct snd_card *card
    the card instance

struct snd_kcontrol *kcontrol
    the control instance to replace

bool add_on_replace
    add the control if not already added

Description

Replaces the given control. If the given control does not exist and the add_on_replace flag is set, the control is added. If the control exists, it is destroyed first.

It frees automatically the control which cannot be added or replaced.

Return

Zero if successful, or a negative error code on failure.

int **snd_ctl_remove**(struct snd_card *card, struct snd_kcontrol *kcontrol)
> remove the control from the card and release it

Parameters

struct snd_card *card
> the card instance

struct snd_kcontrol *kcontrol
> the control instance to remove

Description

Removes the control from the card and then releases the instance. You don't need to call *snd_ctl_free_one()*.

Note that this function takes card->controls_rwsem lock internally.

Return

0 if successful, or a negative error code on failure.

int **snd_ctl_remove_id**(struct snd_card *card, struct snd_ctl_elem_id *id)
> remove the control of the given id and release it

Parameters

struct snd_card *card
> the card instance

struct snd_ctl_elem_id *id
> the control id to remove

Description

Finds the control instance with the given id, removes it from the card list and releases it.

Return

0 if successful, or a negative error code on failure.

int **snd_ctl_remove_user_ctl**(struct snd_ctl_file *file, struct snd_ctl_elem_id *id)
> remove and release the unlocked user control

Parameters

struct snd_ctl_file * file
> active control handle

struct snd_ctl_elem_id *id
> the control id to remove

Description

Finds the control instance with the given id, removes it from the card list and releases it.

Return

0 if successful, or a negative error code on failure.

int **snd_ctl_activate_id**(struct snd_card *card, struct snd_ctl_elem_id *id, int active)
    activate/inactivate the control of the given id

Parameters

struct snd_card *card
    the card instance

struct snd_ctl_elem_id *id
    the control id to activate/inactivate

int active
    non-zero to activate

Description

Finds the control instance with the given id, and activate or inactivate the control together with notification, if changed. The given ID data is filled with full information.

Return

0 if unchanged, 1 if changed, or a negative error code on failure.

int **snd_ctl_rename_id**(struct snd_card *card, struct snd_ctl_elem_id *src_id, struct snd_ctl_elem_id *dst_id)
    replace the id of a control on the card

Parameters

struct snd_card *card
    the card instance

struct snd_ctl_elem_id *src_id
    the old id

struct snd_ctl_elem_id *dst_id
    the new id

Description

Finds the control with the old id from the card, and replaces the id with the new one.

The function tries to keep the already assigned numid while replacing the rest.

Note that this function should be used only in the card initialization phase. Calling after the card instantiation may cause issues with user-space expecting persistent numids.

Return

Zero if successful, or a negative error code on failure.

void **snd_ctl_rename**(struct snd_card *card, struct snd_kcontrol *kctl, const char *name)
    rename the control on the card

Parameters

struct snd_card *card
    the card instance

struct snd_kcontrol *kctl
    the control to rename

const char *name
    the new name

Description

Renames the specified control on the card to the new name.

Note that this function takes card->controls_rwsem lock internally.

struct snd_kcontrol ***snd_ctl_find_numid_locked**(struct snd_card *card, unsigned int
                                                           numid)
    find the control instance with the given number-id

Parameters

struct snd_card *card
    the card instance

unsigned int numid
    the number-id to search

Description

Finds the control instance with the given number-id from the card.

The caller must down card->controls_rwsem before calling this function (if the race condition can happen).

Return

The pointer of the instance if found, or NULL if not.

struct snd_kcontrol ***snd_ctl_find_numid**(struct snd_card *card, unsigned int numid)
    find the control instance with the given number-id

Parameters

struct snd_card *card
    the card instance

unsigned int numid
    the number-id to search

Description

Finds the control instance with the given number-id from the card.

Note that this function takes card->controls_rwsem lock internally.

Return

The pointer of the instance if found, or NULL if not.

struct snd_kcontrol ***snd_ctl_find_id_locked**(struct snd_card *card, const struct
                                                       snd_ctl_elem_id *id)
    find the control instance with the given id

Parameters

struct snd_card *card
    the card instance

const struct snd_ctl_elem_id *id
    the id to search

Description

Finds the control instance with the given id from the card.

The caller must down card->controls_rwsem before calling this function (if the race condition can happen).

Return

The pointer of the instance if found, or NULL if not.

struct snd_kcontrol ***snd_ctl_find_id**(struct snd_card *card, const struct snd_ctl_elem_id *id)
    find the control instance with the given id

Parameters

struct snd_card *card
    the card instance

const struct snd_ctl_elem_id *id
    the id to search

Description

Finds the control instance with the given id from the card.

Note that this function takes card->controls_rwsem lock internally.

Return

The pointer of the instance if found, or NULL if not.

int **snd_ctl_register_ioctl**(snd_kctl_ioctl_func_t fcn)
    register the device-specific control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn
    ioctl callback function

Description

called from each device manager like pcm.c, hwdep.c, etc.

Return

zero if successful, or a negative error code

int **snd_ctl_register_ioctl_compat**(snd_kctl_ioctl_func_t fcn)
    register the device-specific 32bit compat control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn
    ioctl callback function

Return

zero if successful, or a negative error code

int **snd_ctl_unregister_ioctl**(snd_kctl_ioctl_func_t fcn)

    de-register the device-specific control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn

    ioctl callback function to unregister

Return

zero if successful, or a negative error code

int **snd_ctl_unregister_ioctl_compat**(snd_kctl_ioctl_func_t fcn)

    de-register the device-specific compat 32bit control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn

    ioctl callback function to unregister

Return

zero if successful, or a negative error code

int **snd_ctl_request_layer**(const char *module_name)

    request to use the layer

Parameters

const char *module_name

    Name of the kernel module (NULL == build-in)

Return

zero if successful, or an error code when the module cannot be loaded

void **snd_ctl_register_layer**(struct snd_ctl_layer_ops *lops)

    register new control layer

Parameters

struct snd_ctl_layer_ops *lops

    operation structure

Description

The new layer can track all control elements and do additional operations on top (like audio LED handling).

void **snd_ctl_disconnect_layer**(struct snd_ctl_layer_ops *lops)

    disconnect control layer

Parameters

struct snd_ctl_layer_ops *lops

    operation structure

Description

It is expected that the information about tracked cards is freed before this call (the disconnect callback is not called here).

int **snd_ctl_boolean_mono_info**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo)

> Helper function for a standard boolean info callback with a mono channel

Parameters

struct snd_kcontrol *kcontrol
> the kcontrol instance

struct snd_ctl_elem_info *uinfo
> info to store

Description

This is a function that can be used as info callback for a standard boolean control with a single mono channel.

Return

Zero (always successful)

int **snd_ctl_boolean_stereo_info**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo)

> Helper function for a standard boolean info callback with stereo two channels

Parameters

struct snd_kcontrol *kcontrol
> the kcontrol instance

struct snd_ctl_elem_info *uinfo
> info to store

Description

This is a function that can be used as info callback for a standard boolean control with stereo two channels.

Return

Zero (always successful)

int **snd_ctl_enum_info**(struct snd_ctl_elem_info *info, unsigned int channels, unsigned int items, const char *const names[])

> fills the info structure for an enumerated control

Parameters

struct snd_ctl_elem_info *info
> the structure to be filled

unsigned int channels
> the number of the control's channels; often one

unsigned int items
> the number of control values; also the size of **names**

const char *const names[]
> an array containing the names of all control values

Description

Sets all required fields in **info** to their appropriate values. If the control's accessibility is not the default (readable and writable), the caller has to fill **info->access**.

Return

Zero (always successful)

AC97 Codec API

void **snd_ac97_write**(struct snd_ac97 *ac97, unsigned short reg, unsigned short value)
    write a value on the given register

Parameters

struct snd_ac97 *ac97
    the ac97 instance

unsigned short reg
    the register to change

unsigned short value
    the value to set

Description

Writes a value on the given register. This will invoke the write callback directly after the register check. This function doesn't change the register cache unlike #snd_ca97_write_cache(), so use this only when you don't want to reflect the change to the suspend/resume state.

unsigned short **snd_ac97_read**(struct snd_ac97 *ac97, unsigned short reg)
    read a value from the given register

Parameters

struct snd_ac97 *ac97
    the ac97 instance

unsigned short reg
    the register to read

Description

Reads a value from the given register. This will invoke the read callback directly after the register check.

Return

The read value.

void **snd_ac97_write_cache**(struct snd_ac97 *ac97, unsigned short reg, unsigned short value)
    write a value on the given register and update the cache

Parameters

struct snd_ac97 *ac97
    the ac97 instance

unsigned short reg
    the register to change

unsigned short value
    the value to set

Description

Writes a value on the given register and updates the register cache. The cached values are used for the cached-read and the suspend/resume.

int **snd_ac97_update**(struct snd_ac97 *ac97, unsigned short reg, unsigned short value)
    update the value on the given register

Parameters

struct snd_ac97 *ac97
    the ac97 instance

unsigned short reg
    the register to change

unsigned short value
    the value to set

Description

Compares the value with the register cache and updates the value only when the value is changed.

Return

1 if the value is changed, 0 if no change, or a negative code on failure.

int **snd_ac97_update_bits**(struct snd_ac97 *ac97, unsigned short reg, unsigned short mask, unsigned short value)
    update the bits on the given register

Parameters

struct snd_ac97 *ac97
    the ac97 instance

unsigned short reg
    the register to change

unsigned short mask
    the bit-mask to change

unsigned short value
    the value to set

Description

Updates the masked-bits on the given register only when the value is changed.

Return

1 if the bits are changed, 0 if no change, or a negative code on failure.

const char ***snd_ac97_get_short_name**(struct snd_ac97 *ac97)

> retrieve codec name

Parameters

struct snd_ac97 *ac97

> the codec instance

Return

The short identifying name of the codec.

int **snd_ac97_bus**(struct snd_card *card, int num, const struct snd_ac97_bus_ops *ops, void *private_data, struct *snd_ac97_bus* **rbus)

> create an AC97 bus component

Parameters

struct snd_card *card

> the card instance

int num

> the bus number

const struct snd_ac97_bus_ops *ops

> the bus callbacks table

void *private_data

> private data pointer for the new instance

struct snd_ac97_bus **rbus

> the pointer to store the new AC97 bus instance.

Description

Creates an AC97 bus component. An *struct snd_ac97_bus* instance is newly allocated and initialized.

The ops table must include valid callbacks (at least read and write). The other callbacks, wait and reset, are not mandatory.

The clock is set to 48000. If another clock is needed, set `(*rbus)->clock` manually.

The AC97 bus instance is registered as a low-level device, so you don't have to release it manually.

Return

Zero if successful, or a negative error code on failure.

int **snd_ac97_mixer**(struct *snd_ac97_bus* *bus, struct snd_ac97_template *template, struct snd_ac97 **rac97)

> create an Codec97 component

Parameters

struct snd_ac97_bus *bus

> the AC97 bus which codec is attached to

struct snd_ac97_template *template

> the template of ac97, including index, callbacks and the private data.

---

struct snd_ac97 **rac97
> the pointer to store the new ac97 instance.

Description

Creates an Codec97 component. An struct snd_ac97 instance is newly allocated and initialized from the template. The codec is then initialized by the standard procedure.

The template must include the codec number (num) and address (addr), and the private data (private_data).

The ac97 instance is registered as a low-level device, so you don't have to release it manually.

Return

Zero if successful, or a negative error code on failure.

int **snd_ac97_update_power**(struct snd_ac97 *ac97, int reg, int powerup)
> update the powerdown register

Parameters

struct snd_ac97 *ac97
> the codec instance

int reg
> the rate register, e.g. AC97_PCM_FRONT_DAC_RATE

int powerup
> non-zero when power up the part

Description

Update the AC97 powerdown register bits of the given part.

Return

Zero.

void **snd_ac97_suspend**(struct snd_ac97 *ac97)
> General suspend function for AC97 codec

Parameters

struct snd_ac97 *ac97
> the ac97 instance

Description

Suspends the codec, power down the chip.

void **snd_ac97_resume**(struct snd_ac97 *ac97)
> General resume function for AC97 codec

Parameters

struct snd_ac97 *ac97
> the ac97 instance

Description

Do the standard resume procedure, power up and restoring the old register values.

int **snd_ac97_tune_hardware**(struct snd_ac97 *ac97, const struct ac97_quirk *quirk, const char *override)

>   tune up the hardware

Parameters

struct snd_ac97 *ac97
>   the ac97 instance

const struct ac97_quirk *quirk
>   quirk list

const char *override
>   explicit quirk value (overrides the list if non-NULL)

Description

Do some workaround for each pci device, such as renaming of the headphone (true line-out) control as "Master". The quirk-list must be terminated with a zero-filled entry.

Return

Zero if successful, or a negative error code on failure.

int **snd_ac97_set_rate**(struct snd_ac97 *ac97, int reg, unsigned int rate)

>   change the rate of the given input/output.

Parameters

struct snd_ac97 *ac97
>   the ac97 instance

int reg
>   the register to change

unsigned int rate
>   the sample rate to set

Description

Changes the rate of the given input/output on the codec. If the codec doesn't support VAR, the rate must be 48000 (except for SPDIF).

The valid registers are AC97_PCM_MIC_ADC_RATE, AC97_PCM_FRONT_DAC_RATE, AC97_PCM_LR_ADC_RATE. AC97_PCM_SURR_DAC_RATE and AC97_PCM_LFE_DAC_RATE are accepted if the codec supports them. AC97_SPDIF is accepted as a pseudo register to modify the SPDIF status bits.

Return

Zero if successful, or a negative error code on failure.

int **snd_ac97_pcm_assign**(struct *snd_ac97_bus* *bus, unsigned short pcms_count, const struct ac97_pcm *pcms)

>   assign AC97 slots to given PCM streams

Parameters

struct snd_ac97_bus *bus
>   the ac97 bus instance

---

unsigned short pcms_count
>   count of PCMs to be assigned

const struct ac97_pcm *pcms
>   PCMs to be assigned

Description

It assigns available AC97 slots for given PCMs. If none or only some slots are available, pcm->xxx.slots and pcm->xxx.rslots[] members are reduced and might be zero.

Return

Zero if successful, or a negative error code on failure.

int **snd_ac97_pcm_open**(struct ac97_pcm *pcm, unsigned int rate, enum ac97_pcm_cfg cfg, unsigned short slots)
>   opens the given AC97 pcm

Parameters

struct ac97_pcm *pcm
>   the ac97 pcm instance

unsigned int rate
>   rate in Hz, if codec does not support VRA, this value must be 48000Hz

enum ac97_pcm_cfg cfg
>   output stream characteristics

unsigned short slots
>   a subset of allocated slots (snd_ac97_pcm_assign) for this pcm

Description

It locks the specified slots and sets the given rate to AC97 registers.

Return

Zero if successful, or a negative error code on failure.

int **snd_ac97_pcm_close**(struct ac97_pcm *pcm)
>   closes the given AC97 pcm

Parameters

struct ac97_pcm *pcm
>   the ac97 pcm instance

Description

It frees the locked AC97 slots.

Return

Zero.

int **snd_ac97_pcm_double_rate_rules**(struct snd_pcm_runtime *runtime)
>   set double rate constraints

Parameters

```
struct snd_pcm_runtime *runtime
```
the runtime of the ac97 front playback pcm

Description

Installs the hardware constraint rules to prevent using double rates and more than two channels at the same time.

Return

Zero if successful, or a negative error code on failure.

Virtual Master Control API

int **snd_ctl_add_followers**(struct snd_card *card, struct snd_kcontrol *master, const char *const *list)
   add multiple followers to vmaster

Parameters

```
struct snd_card *card
```
card instance

```
struct snd_kcontrol *master
```
the target vmaster kcontrol object

```
const char * const *list
```
NULL-terminated list of name strings of followers to be added

Description

Adds the multiple follower kcontrols with the given names. Returns 0 for success or a negative error code.

struct snd_kcontrol ***snd_ctl_make_virtual_master**(char *name, const unsigned int *tlv)
   Create a virtual master control

Parameters

```
char *name
```
name string of the control element to create

```
const unsigned int *tlv
```
optional TLV int array for dB information

Description

Creates a virtual master control with the given name string.

After creating a vmaster element, you can add the follower controls via *snd_ctl_add_follower()* or *snd_ctl_add_follower_uncached()*.

The optional argument **tlv** can be used to specify the TLV information for dB scale of the master control. It should be a single element with #SNDRV_CTL_TLVT_DB_SCALE, #SNDRV_CTL_TLV_DB_MINMAX or #SNDRV_CTL_TLVT_DB_MINMAX_MUTE type, and should be the max 0dB.

Return

The created control element, or NULL for errors (ENOMEM).

int **snd_ctl_add_vmaster_hook**(struct snd_kcontrol *kcontrol, void (*hook)(void
          *private_data, int), void *private_data)

 Add a hook to a vmaster control

Parameters

struct snd_kcontrol *kcontrol
 vmaster kctl element

void (*hook)(void *private_data, int)
 the hook function

void *private_data
 the private_data pointer to be saved

Description

Adds the given hook to the vmaster control element so that it's called at each time when the
value is changed.

Return

Zero.

void **snd_ctl_sync_vmaster**(struct snd_kcontrol *kcontrol, bool hook_only)

 Sync the vmaster followers and hook

Parameters

struct snd_kcontrol *kcontrol
 vmaster kctl element

bool hook_only
 sync only the hook

Description

Forcibly call the put callback of each follower and call the hook function to synchronize with
the current value of the given vmaster element. NOP when NULL is passed to **kcontrol**.

int **snd_ctl_apply_vmaster_followers**(struct snd_kcontrol *kctl, int (*func)(struct
          snd_kcontrol *vfollower, struct snd_kcontrol
          *follower, void *arg), void *arg)

 Apply function to each vmaster follower

Parameters

struct snd_kcontrol *kctl
 vmaster kctl element

**int (*func)(struct snd_kcontrol *vfollower, struct snd_kcontrol *follower, void
*arg)**
 function to apply

void *arg
 optional function argument

Description

Apply the function **func** to each follower kctl of the given vmaster kctl.

Return

0 if successful, or a negative error code

struct snd_kcontrol ***snd_ctl_find_id_mixer**(struct snd_card *card, const char *name)
    find the control instance with the given name string

Parameters

struct snd_card *card
    the card instance

const char *name
    the name string

Description

Finds the control instance with the given name and **SNDRV_CTL_ELEM_IFACE_MIXER**. Other fields are set to zero.

This is merely a wrapper to *snd_ctl_find_id()*.

Return

The pointer of the instance if found, or NULL if not.

int **snd_ctl_add_follower**(struct snd_kcontrol *master, struct snd_kcontrol *follower)
    Add a virtual follower control

Parameters

struct snd_kcontrol *master
    vmaster element

struct snd_kcontrol *follower
    follower element to add

Description

Add a virtual follower control to the given master element created via snd_ctl_create_virtual_master() beforehand.

All followers must be the same type (returning the same information via info callback). The function doesn't check it, so it's your responsibility.

Also, some additional limitations: at most two channels, logarithmic volume control (dB level) thus no linear volume, master can only attenuate the volume without gain

Return

Zero if successful or a negative error code.

int **snd_ctl_add_follower_uncached**(struct snd_kcontrol *master, struct snd_kcontrol *follower)
    Add a virtual follower control

Parameters

struct snd_kcontrol *master
    vmaster element

struct snd_kcontrol *follower
    follower element to add

1.1. The ALSA Driver API 73

Description

Add a virtual follower control to the given master. Unlike *snd_ctl_add_follower()*, the element added via this function is supposed to have volatile values, and get callback is called at each time queried from the master.

When the control peeks the hardware values directly and the value can be changed by other means than the put callback of the element, this function should be used to keep the value always up-to-date.

Return

Zero if successful or a negative error code.

1.1.4 MIDI API

Raw MIDI API

int **snd_rawmidi_receive**(struct snd_rawmidi_substream *substream, const unsigned char
                               *buffer, int count)

   receive the input data from the device

Parameters

struct snd_rawmidi_substream *substream
      the rawmidi substream

const unsigned char *buffer
      the buffer pointer

int count
      the data size to read

Description

Reads the data from the internal buffer.

Return

The size of read data, or a negative error code on failure.

int **snd_rawmidi_transmit_empty**(struct snd_rawmidi_substream *substream)
      check whether the output buffer is empty

Parameters

struct snd_rawmidi_substream *substream
      the rawmidi substream

Return

1 if the internal output buffer is empty, 0 if not.

int **snd_rawmidi_transmit_peek**(struct snd_rawmidi_substream *substream, unsigned char
                                     *buffer, int count)
      copy data from the internal buffer

Parameters

struct snd_rawmidi_substream *substream
    the rawmidi substream

unsigned char *buffer
    the buffer pointer

int count
    data size to transfer

Description

Copies data from the internal output buffer to the given buffer.

Call this in the interrupt handler when the midi output is ready, and call
snd_rawmidi_transmit_ack() after the transmission is finished.

Return

The size of copied data, or a negative error code on failure.

int **snd_rawmidi_transmit_ack**(struct snd_rawmidi_substream *substream, int count)
    acknowledge the transmission

Parameters

struct snd_rawmidi_substream *substream
    the rawmidi substream

int count
    the transferred count

Description

Advances the hardware pointer for the internal output buffer with the given size and updates
the condition. Call after the transmission is finished.

Return

The advanced size if successful, or a negative error code on failure.

int **snd_rawmidi_transmit**(struct snd_rawmidi_substream *substream, unsigned char *buffer,
                          int count)
    copy from the buffer to the device

Parameters

struct snd_rawmidi_substream *substream
    the rawmidi substream

unsigned char *buffer
    the buffer pointer

int count
    the data size to transfer

Description

Copies data from the buffer to the device and advances the pointer.

Return

The copied size if successful, or a negative error code on failure.

---

int **snd_rawmidi_proceed**(struct snd_rawmidi_substream *substream)

    Discard the all pending bytes and proceed

Parameters

struct snd_rawmidi_substream *substream

    rawmidi substream

Return

the number of discarded bytes

int **snd_rawmidi_new**(struct snd_card *card, char *id, int device, int output_count, int input_count, struct snd_rawmidi **rrawmidi)

    create a rawmidi instance

Parameters

struct snd_card *card

    the card instance

char *id

    the id string

int device

    the device index

int output_count

    the number of output streams

int input_count

    the number of input streams

struct snd_rawmidi **rrawmidi

    the pointer to store the new rawmidi instance

Description

Creates a new rawmidi instance. Use *snd_rawmidi_set_ops()* to set the operators to the new instance.

Return

Zero if successful, or a negative error code on failure.

void **snd_rawmidi_set_ops**(struct snd_rawmidi *rmidi, int stream, const struct snd_rawmidi_ops *ops)

    set the rawmidi operators

Parameters

struct snd_rawmidi *rmidi

    the rawmidi instance

int stream

    the stream direction, SNDRV_RAWMIDI_STREAM_XXX

const struct snd_rawmidi_ops *ops

    the operator table

Description

Sets the rawmidi operators for the given stream direction.

MPU401-UART API

irqreturn_t **snd_mpu401_uart_interrupt**(int irq, void *dev_id)
 generic MPU401-UART interrupt handler

Parameters

`int irq`
 the irq number

`void *dev_id`
 mpu401 instance

Description

Processes the interrupt for MPU401-UART i/o.

Return

`IRQ_HANDLED` if the interrupt was handled. `IRQ_NONE` otherwise.

irqreturn_t **snd_mpu401_uart_interrupt_tx**(int irq, void *dev_id)
 generic MPU401-UART transmit irq handler

Parameters

`int irq`
 the irq number

`void *dev_id`
 mpu401 instance

Description

Processes the interrupt for MPU401-UART output.

Return

`IRQ_HANDLED` if the interrupt was handled. `IRQ_NONE` otherwise.

int **snd_mpu401_uart_new**(struct snd_card *card, int device, unsigned short hardware,
         unsigned long port, unsigned int info_flags, int irq, struct
         snd_rawmidi **rrawmidi)
 create an MPU401-UART instance

Parameters

`struct snd_card *card`
 the card instance

`int device`
 the device index, zero-based

`unsigned short hardware`
 the hardware type, MPU401_HW_XXXX

unsigned long port
    the base address of MPU401 port

unsigned int info_flags
    bitflags MPU401_INFO_XXX

int irq
    the ISA irq number, -1 if not to be allocated

struct snd_rawmidi ** rrawmidi
    the pointer to store the new rawmidi instance

Description

Creates a new MPU-401 instance.

Note that the rawmidi instance is returned on the rrawmidi argument, not the mpu401 instance itself. To access to the mpu401 instance, cast from rawmidi->private_data (with struct snd_mpu401 magic-cast).

Return

Zero if successful, or a negative error code.

1.1.5 Proc Info API

Proc Info Interface

int **snd_info_get_line**(struct snd_info_buffer *buffer, char *line, int len)
    read one line from the procfs buffer

Parameters

struct snd_info_buffer *buffer
    the procfs buffer

char *line
    the buffer to store

int len
    the max. buffer size

Description

Reads one line from the buffer and stores the string.

Return

Zero if successful, or 1 if error or EOF.

const char ***snd_info_get_str**(char *dest, const char *src, int len)
    parse a string token

Parameters

char *dest
    the buffer to store the string token

const char *src
    the original string

int len
    the max. length of token - 1

Description

Parses the original string and copy a token to the given string buffer.

Return

The updated pointer of the original string so that it can be used for the next call.

struct snd_info_entry *__snd_info_create_module_entry__(struct *module* *module, const char *name, struct snd_info_entry *parent)

    create an info entry for the given module

Parameters

struct module * module
    the module pointer

const char *name
    the file name

struct snd_info_entry *parent
    the parent directory

Description

Creates a new info entry and assigns it to the given module.

Return

The pointer of the new instance, or NULL on failure.

struct snd_info_entry *__snd_info_create_card_entry__(struct snd_card *card, const char *name, struct snd_info_entry *parent)

    create an info entry for the given card

Parameters

struct snd_card *card
    the card instance

const char *name
    the file name

struct snd_info_entry * parent
    the parent directory

Description

Creates a new info entry and assigns it to the given card.

Return

The pointer of the new instance, or NULL on failure.

void __snd_info_free_entry__(struct snd_info_entry *entry)
    release the info entry

Parameters

struct snd_info_entry * entry
    the info entry

Description

Releases the info entry.

int **snd_info_register**(struct snd_info_entry *entry)
    register the info entry

Parameters

struct snd_info_entry *entry
    the info entry

Description

Registers the proc info entry. The all children entries are registered recursively.

Return

Zero if successful, or a negative error code on failure.

int **snd_card_rw_proc_new**(struct snd_card *card, const char *name, void *private_data, void
                            (*read)(struct snd_info_entry*, struct snd_info_buffer*), void
                            (*write)(struct snd_info_entry *entry, struct snd_info_buffer
                            *buffer))
    Create a read/write text proc file entry for the card

Parameters

struct snd_card *card
    the card instance

const char *name
    the file name

void *private_data
    the arbitrary private data

void (*read)(struct snd_info_entry *, struct snd_info_buffer *)
    the read callback

void (*write)(struct snd_info_entry *entry, struct snd_info_buffer *buffer)
    the write callback, NULL for read-only

Description

This proc file entry will be registered via *snd_card_register()* call, and it will be removed automatically at the card removal, too.

Return

zero if successful, or a negative error code

1.1.6 Compress Offload

Compress Offload API

int **snd_compr_stop_error**(struct *snd_compr_stream* *stream, snd_pcm_state_t state)
    Report a fatal error on a stream

Parameters

struct snd_compr_stream *stream
    pointer to stream

snd_pcm_state_t state
    state to transition the stream to

Description

Stop the stream and set its state.

Should be called with compressed device lock held.

Return

zero if successful, or a negative error code

int **snd_compress_new**(struct snd_card *card, int device, int dirn, const char *id, struct *snd_compr* *compr)
    create new compress device

Parameters

struct snd_card *card
    sound card pointer

int device
    device number

int dirn
    device direction, should be of type enum snd_compr_direction

const char *id
    ID string

struct snd_compr *compr
    compress device pointer

Return

zero if successful, or a negative error code

struct **snd_compressed_buffer**
    compressed buffer

Definition:

```
struct snd_compressed_buffer {
    __u32 fragment_size;
    __u32 fragments;
};
```

Members

fragment_size
   size of buffer fragment in bytes

fragments
   number of such fragments

struct **snd_compr_params**
   compressed stream params

Definition:

```
struct snd_compr_params {
    struct snd_compressed_buffer buffer;
    struct snd_codec codec;
    __u8 no_wake_mode;
};
```

Members

buffer
   buffer description

codec
   codec parameters

no_wake_mode
   dont wake on fragment elapsed

struct **snd_compr_tstamp**
   timestamp descriptor

Definition:

```
struct snd_compr_tstamp {
    __u32 byte_offset;
    __u32 copied_total;
    __u32 pcm_frames;
    __u32 pcm_io_frames;
    __u32 sampling_rate;
};
```

Members

byte_offset
   Byte offset in ring buffer to DSP

copied_total
   Total number of bytes copied from/to ring buffer to/by DSP

pcm_frames
   Frames decoded or encoded by DSP. This field will evolve by large steps and should only
   be used to monitor encoding/decoding progress. It shall not be used for timing estimates.

pcm_io_frames
   Frames rendered or received by DSP into a mixer or an audio output/input. This field
   should be used for A/V sync or time estimates.

sampling_rate
　　sampling rate of audio

struct **snd_compr_avail**
　　avail descriptor

Definition:

```
struct snd_compr_avail {
    __u64 avail;
    struct snd_compr_tstamp tstamp;
};
```

Members

avail
　　Number of bytes available in ring buffer for writing/reading

tstamp
　　timestamp information

struct **snd_compr_caps**
　　caps descriptor

Definition:

```
struct snd_compr_caps {
    __u32 num_codecs;
    __u32 direction;
    __u32 min_fragment_size;
    __u32 max_fragment_size;
    __u32 min_fragments;
    __u32 max_fragments;
    __u32 codecs[MAX_NUM_CODECS];
    __u32 reserved[11];
};
```

Members

num_codecs
　　number of codecs supported

direction
　　direction supported. Of type snd_compr_direction

min_fragment_size
　　minimum fragment supported by DSP

max_fragment_size
　　maximum fragment supported by DSP

min_fragments
　　min fragments supported by DSP

max_fragments
　　max fragments supported by DSP

---

codecs
    pointer to array of codecs

reserved
    reserved field

struct **snd_compr_codec_caps**
    query capability of codec

Definition:

```
struct snd_compr_codec_caps {
    __u32 codec;
    __u32 num_descriptors;
    struct snd_codec_desc descriptor[MAX_NUM_CODEC_DESCRIPTORS];
};
```

Members

codec
    codec for which capability is queried

num_descriptors
    number of codec descriptors

descriptor
    array of codec capability descriptor

enum **sndrv_compress_encoder**
    encoder metadata key

Constants

SNDRV_COMPRESS_ENCODER_PADDING
    no of samples appended by the encoder at the end of the track

SNDRV_COMPRESS_ENCODER_DELAY
    no of samples inserted by the encoder at the beginning of the track

struct **snd_compr_metadata**
    compressed stream metadata

Definition:

```
struct snd_compr_metadata {
    __u32 key;
    __u32 value[8];
};
```

Members

key
    key id

value
    key value

struct **snd_enc_vorbis**

    Vorbis encoder parameters

Definition:

```
struct snd_enc_vorbis {
    __s32 quality;
    __u32 managed;
    __u32 max_bit_rate;
    __u32 min_bit_rate;
    __u32 downmix;
};
```

Members

quality

    Sets encoding quality to n, between -1 (low) and 10 (high). In the default mode of operation, the quality level is 3. Normal quality range is 0 - 10.

managed

    Boolean. Set bitrate management mode. This turns off the normal VBR encoding, but allows hard or soft bitrate constraints to be enforced by the encoder. This mode can be slower, and may also be lower quality. It is primarily useful for streaming.

max_bit_rate

    Enabled only if managed is TRUE

min_bit_rate

    Enabled only if managed is TRUE

downmix

    Boolean. Downmix input from stereo to mono (has no effect on non-stereo streams). Useful for lower-bitrate encoding.

Description

These options were extracted from the OpenMAX IL spec and Gstreamer vorbisenc properties

For best quality users should specify VBR mode and set quality levels.

struct **snd_enc_real**

    RealAudio encoder parameters

Definition:

```
struct snd_enc_real {
    __u32 quant_bits;
    __u32 start_region;
    __u32 num_regions;
};
```

Members

quant_bits

    number of coupling quantization bits in the stream

start_region

    coupling start region in the stream

num_regions
   number of regions value

Description

These options were extracted from the OpenMAX IL spec

struct **snd_enc_flac**
   FLAC encoder parameters

Definition:

```
struct snd_enc_flac {
    __u32 num;
    __u32 gain;
};
```

Members

num
   serial number, valid only for OGG formats needs to be set by application

gain
   Add replay gain tags

Description

These options were extracted from the FLAC online documentation at http://flac.sourceforge.net/documentation_tools_flac.html

To make the API simpler, it is assumed that the user will select quality profiles. Additional options that affect encoding quality and speed can be added at a later stage if needed.

By default the Subset format is used by encoders.

TAGS such as pictures, etc, cannot be handled by an offloaded encoder and are not supported in this API.

struct **snd_compr_runtime**
   runtime stream description

Definition:

```
struct snd_compr_runtime {
    snd_pcm_state_t state;
    struct snd_compr_ops *ops;
    void *buffer;
    u64 buffer_size;
    u32 fragment_size;
    u32 fragments;
    u64 total_bytes_available;
    u64 total_bytes_transferred;
    wait_queue_head_t sleep;
    void *private_data;
    unsigned char *dma_area;
    dma_addr_t dma_addr;
    size_t dma_bytes;
```

```
        struct snd_dma_buffer *dma_buffer_p;
};
```

Members

state
    stream state

ops
    pointer to DSP callbacks

buffer
    pointer to kernel buffer, valid only when not in mmap mode or DSP doesn't implement copy

buffer_size
    size of the above buffer

fragment_size
    size of buffer fragment in bytes

fragments
    number of such fragments

total_bytes_available
    cumulative number of bytes made available in the ring buffer

total_bytes_transferred
    cumulative bytes transferred by offload DSP

sleep
    poll sleep

private_data
    driver private data pointer

dma_area
    virtual buffer address

dma_addr
    physical buffer address (not accessible from main CPU)

dma_bytes
    size of DMA area

dma_buffer_p
    runtime dma buffer pointer

struct **snd_compr_stream**
    compressed stream

Definition:

```
struct snd_compr_stream {
    const char *name;
    struct snd_compr_ops *ops;
    struct snd_compr_runtime *runtime;
    struct snd_compr *device;
    struct delayed_work error_work;
```

```
    enum snd_compr_direction direction;
    bool metadata_set;
    bool next_track;
    bool partial_drain;
    bool pause_in_draining;
    void *private_data;
    struct snd_dma_buffer dma_buffer;
};
```

Members

name
    device name

ops
    pointer to DSP callbacks

runtime
    pointer to runtime structure

device
    device pointer

error_work
    delayed work used when closing the stream due to an error

direction
    stream direction, playback/recording

metadata_set
    metadata set flag, true when set

next_track
    has userspace signal next track transition, true when set

partial_drain
    undergoing partial_drain for stream, true when set

pause_in_draining
    paused during draining state, true when set

private_data
    pointer to DSP private data

dma_buffer
    allocated buffer if any

struct **snd_compr_ops**
    compressed path DSP operations

Definition:

```
struct snd_compr_ops {
    int (*open)(struct snd_compr_stream *stream);
    int (*free)(struct snd_compr_stream *stream);
    int (*set_params)(struct snd_compr_stream *stream, struct snd_compr_params
↪*params);
    int (*get_params)(struct snd_compr_stream *stream, struct snd_codec
↪*params);
```

```
    int (*set_metadata)(struct snd_compr_stream *stream, struct snd_compr_
↪metadata *metadata);
    int (*get_metadata)(struct snd_compr_stream *stream, struct snd_compr_
↪metadata *metadata);
    int (*trigger)(struct snd_compr_stream *stream, int cmd);
    int (*pointer)(struct snd_compr_stream *stream, struct snd_compr_tstamp
↪*tstamp);
    int (*copy)(struct snd_compr_stream *stream, char __user *buf, size_t
↪count);
    int (*mmap)(struct snd_compr_stream *stream, struct vm_area_struct *vma);
    int (*ack)(struct snd_compr_stream *stream, size_t bytes);
    int (*get_caps) (struct snd_compr_stream *stream, struct snd_compr_caps
↪*caps);
    int (*get_codec_caps) (struct snd_compr_stream *stream, struct snd_compr_
↪codec_caps *codec);
};
```

Members

open
> Open the compressed stream This callback is mandatory and shall keep dsp ready to receive the stream parameter

free
> Close the compressed stream, mandatory

set_params
> Sets the compressed stream parameters, mandatory This can be called in during stream creation only to set codec params and the stream properties

get_params
> retrieve the codec parameters, mandatory

set_metadata
> Set the metadata values for a stream

get_metadata
> retrieves the requested metadata values from stream

trigger
> Trigger operations like start, pause, resume, drain, stop. This callback is mandatory

pointer
> Retrieve current h/w pointer information. Mandatory

copy
> Copy the compressed data to/from userspace, Optional Can't be implemented if DSP supports mmap

mmap
> DSP mmap method to mmap DSP memory

ack
> Ack for DSP when data is written to audio buffer, Optional Not valid if copy is implemented

get_caps
> Retrieve DSP capabilities, mandatory

1.1. The ALSA Driver API

get_codec_caps
    Retrieve capabilities for a specific codec, mandatory

struct **snd_compr**
    Compressed device

Definition:

```
struct snd_compr {
    const char *name;
    struct device *dev;
    struct snd_compr_ops *ops;
    void *private_data;
    struct snd_card *card;
    unsigned int direction;
    struct mutex lock;
    int device;
    bool use_pause_in_draining;
#ifdef CONFIG_SND_VERBOSE_PROCFS;
};
```

Members

name
    DSP device name

dev
    associated device instance

ops
    pointer to DSP callbacks

private_data
    pointer to DSP pvt data

card
    sound card pointer

direction
    Playback or capture direction

lock
    device lock

device
    device id

use_pause_in_draining
    allow pause in draining, true when set

void **snd_compr_use_pause_in_draining**(struct *snd_compr_stream* *substream)
    Allow pause and resume in draining state

Parameters

struct snd_compr_stream *substream
    compress substream to set

Description

Allow pause and resume in draining state. Only HW driver supports this transition can call this API.

void **snd_compr_set_runtime_buffer**(struct *snd_compr_stream* *stream, struct
snd_dma_buffer *bufp)

Set the Compress runtime buffer

Parameters

struct snd_compr_stream *stream
compress stream to set

struct snd_dma_buffer *bufp
the buffer information, NULL to clear

Description

Copy the buffer information to runtime buffer when **bufp** is non-NULL. Otherwise it clears the current buffer information.

1.1.7 ASoC

ASoC Core API

struct snd_soc_component ***snd_soc_kcontrol_component**(struct snd_kcontrol *kcontrol)

Returns the component that registered the control

Parameters

struct snd_kcontrol *kcontrol
The control for which to get the component

Note

This function will work correctly if the control has been registered for a component. With snd_soc_add_codec_controls() or via table based setup for either a CODEC or component driver. Otherwise the behavior is undefined.

struct snd_soc_dai ***snd_soc_find_dai**(const struct snd_soc_dai_link_component *dlc)

Find a registered DAI

Parameters

const struct snd_soc_dai_link_component *dlc
name of the DAI or the DAI driver and optional component info to match

Description

This function will search all registered components and their DAIs to find the DAI of the same name. The component's of_node and name should also match if being specified.

Return

pointer of DAI, or NULL if not found.

void **snd_soc_remove_pcm_runtime**(struct snd_soc_card *card, struct snd_soc_pcm_runtime
*rtd)

> Remove a pcm_runtime from card

Parameters

struct snd_soc_card *card
> The ASoC card to which the pcm_runtime has

struct snd_soc_pcm_runtime *rtd
> The pcm_runtime to remove

Description

This function removes a pcm_runtime from the ASoC card.

int **snd_soc_add_pcm_runtime**(struct snd_soc_card *card, struct snd_soc_dai_link *dai_link)

> Add a pcm_runtime dynamically via dai_link

Parameters

struct snd_soc_card *card
> The ASoC card to which the pcm_runtime is added

struct snd_soc_dai_link *dai_link
> The DAI link to find pcm_runtime

Description

This function adds a pcm_runtime ASoC card by using dai_link.

Note

Topology can use this API to add pcm_runtime when probing the topology component. And
machine drivers can still define static DAI links in dai_link array.

int **snd_soc_runtime_set_dai_fmt**(struct snd_soc_pcm_runtime *rtd, unsigned int dai_fmt)

> Change DAI link format for a ASoC runtime

Parameters

struct snd_soc_pcm_runtime *rtd
> The runtime for which the DAI link format should be changed

unsigned int dai_fmt
> The new DAI link format

Description

This function updates the DAI link format for all DAIs connected to the DAI link for the specified
runtime.

Returns 0 on success, otherwise a negative error code.

Note

For setups with a static format set the dai_fmt field in the corresponding snd_dai_link struct
instead of using this function.

int **snd_soc_set_dmi_name**(struct snd_soc_card *card, const char *flavour)

> Register DMI names to card

Parameters

struct snd_soc_card *card
  The card to register DMI names

const char *flavour
  The flavour "differentiator" for the card amongst its peers.

Description

An Intel machine driver may be used by many different devices but are difficult for userspace to differentiate, since machine drivers usually use their own name as the card short name and leave the card long name blank. To differentiate such devices and fix bugs due to lack of device-specific configurations, this function allows DMI info to be used as the sound card long name, in the format of "vendor-product-version-board" (Character '-' is used to separate different DMI fields here). This will help the user space to load the device-specific Use Case Manager (UCM) configurations for the card.

Possible card long names may be: DellInc.-XPS139343-01-0310JH ASUSTeKCOMPUTERINC.-T100TA-1.0-T100TA Circuitco-MinnowboardMaxD0PLATFORM-D0-MinnowBoardMAX

This function also supports flavoring the card longname to provide the extra differentiation, like "vendor-product-version-board-flavor".

We only keep number and alphabet characters and a few separator characters in the card long name since UCM in the user space uses the card long names as card configuration directory names and AudoConf cannot support special characters like SPACE.

Returns 0 on success, otherwise a negative error code.

struct snd_kcontrol ***snd_soc_cnew**(const struct snd_kcontrol_new *_template, void *data, const char *long_name, const char *prefix)

  create new control

Parameters

const struct snd_kcontrol_new *_template
  control template

void *data
  control private data

const char *long_name
  control long name

const char *prefix
  control name prefix

Description

Create a new mixer control from a template control.

Returns 0 for success, else error.

int **snd_soc_add_component_controls**(struct snd_soc_component *component, const struct snd_kcontrol_new *controls, unsigned int num_controls)

  Add an array of controls to a component.

Parameters

struct snd_soc_component *component
> Component to add controls to

const struct snd_kcontrol_new *controls
> Array of controls to add

unsigned int num_controls
> Number of elements in the array

Return

0 for success, else error.

int **snd_soc_add_card_controls**(struct snd_soc_card *soc_card, const struct snd_kcontrol_new *controls, int num_controls)
> add an array of controls to a SoC card. Convenience function to add a list of controls.

Parameters

struct snd_soc_card *soc_card
> SoC card to add controls to

const struct snd_kcontrol_new *controls
> array of controls to add

int num_controls
> number of elements in the array

Description

Return 0 for success, else error.

int **snd_soc_add_dai_controls**(struct snd_soc_dai *dai, const struct snd_kcontrol_new *controls, int num_controls)
> add an array of controls to a DAI. Convenience function to add a list of controls.

Parameters

struct snd_soc_dai *dai
> DAI to add controls to

const struct snd_kcontrol_new *controls
> array of controls to add

int num_controls
> number of elements in the array

Description

Return 0 for success, else error.

int **snd_soc_register_card**(struct snd_soc_card *card)
> Register a card with the ASoC core

Parameters

struct snd_soc_card *card
> Card to register

void **snd_soc_unregister_card**(struct snd_soc_card *card)

> Unregister a card with the ASoC core

Parameters

struct snd_soc_card *card

> Card to unregister

struct snd_soc_dai ***snd_soc_register_dai**(struct snd_soc_component *component, struct snd_soc_dai_driver *dai_drv, bool legacy_dai_naming)

> Register a DAI dynamically & create its widgets

Parameters

struct snd_soc_component *component

> The component the DAIs are registered for

struct snd_soc_dai_driver *dai_drv

> DAI driver to use for the DAI

bool legacy_dai_naming

> if `true`, use legacy single-name format; if `false`, use multiple-name format;

Description

Topology can use this API to register DAIs when probing a component. These DAIs's widgets will be freed in the card cleanup and the DAIs will be freed in the component cleanup.

void **snd_soc_unregister_dais**(struct snd_soc_component *component)

> Unregister DAIs from the ASoC core

Parameters

struct snd_soc_component *component

> The component for which the DAIs should be unregistered

int **snd_soc_register_dais**(struct snd_soc_component *component, struct snd_soc_dai_driver *dai_drv, size_t count)

> Register a DAI with the ASoC core

Parameters

struct snd_soc_component *component

> The component the DAIs are registered for

struct snd_soc_dai_driver *dai_drv

> DAI driver to use for the DAIs

size_t count

> Number of DAIs

void **snd_soc_unregister_component_by_driver**(struct device *dev, const struct snd_soc_component_driver *component_driver)

> Unregister component using a given driver from the ASoC core

Parameters

---

struct device *dev
    The device to unregister

const struct snd_soc_component_driver *component_driver
    The component driver to unregister

void **snd_soc_unregister_component**(struct device *dev)
    Unregister all related component from the ASoC core

Parameters

struct device *dev
    The device to unregister

struct snd_soc_dai ***devm_snd_soc_register_dai**(struct device *dev, struct snd_soc_component *component, struct snd_soc_dai_driver *dai_drv, bool legacy_dai_naming)
    resource-managed dai registration

Parameters

struct device *dev
    Device used to manage component

struct snd_soc_component *component
    The component the DAIs are registered for

struct snd_soc_dai_driver *dai_drv
    DAI driver to use for the DAI

bool legacy_dai_naming
    if `true`, use legacy single-name format; if `false`, use multiple-name format;

int **devm_snd_soc_register_component**(struct device *dev, const struct snd_soc_component_driver *cmpnt_drv, struct snd_soc_dai_driver *dai_drv, int num_dai)
    resource managed component registration

Parameters

struct device *dev
    Device used to manage component

const struct snd_soc_component_driver *cmpnt_drv
    Component driver

struct snd_soc_dai_driver *dai_drv
    DAI driver

int num_dai
    Number of DAIs to register

Description

Register a component with automatic unregistration when the device is unregistered.

int **devm_snd_soc_register_card**(struct device *dev, struct snd_soc_card *card)
    resource managed card registration

Parameters

struct device *dev
    Device used to manage card

struct snd_soc_card *card
    Card to register

Description

Register a card with automatic unregistration when the device is unregistered.

int **devm_snd_dmaengine_pcm_register**(struct device *dev, const struct *snd_dmaengine_pcm_config* *config, unsigned int flags)
    resource managed dmaengine PCM registration

Parameters

struct device *dev
    The parent device for the PCM device

const struct snd_dmaengine_pcm_config *config
    Platform specific PCM configuration

unsigned int flags
    Platform specific quirks

Description

Register a dmaengine based PCM device with automatic unregistration when the device is unregistered.

int **snd_soc_component_set_sysclk**(struct snd_soc_component *component, int clk_id, int source, unsigned int freq, int dir)
    configure COMPONENT system or master clock.

Parameters

struct snd_soc_component *component
    COMPONENT

int clk_id
    DAI specific clock ID

int source
    Source for the clock

unsigned int freq
    new clock frequency in Hz

int dir
    new clock direction - input/output.

Description

Configures the CODEC master (MCLK) or system (SYSCLK) clocking.

int **snd_soc_component_set_jack**(struct snd_soc_component *component, struct snd_soc_jack *jack, void *data)
    configure component jack.

Parameters

struct snd_soc_component *component
    COMPONENTs

struct snd_soc_jack *jack
    structure to use for the jack

void *data
    can be used if codec driver need extra data for configuring jack

Description

Configures and enables jack detection function.

int **snd_soc_component_get_jack_type**(struct snd_soc_component *component)

Parameters

struct snd_soc_component *component
    COMPONENTs

Description

Returns the jack type of the component This can either be the supported type or one read from devicetree with the property: jack-type.

void **snd_soc_component_init_regmap**(struct snd_soc_component *component, struct
                                        regmap *regmap)

    Initialize regmap instance for the component

Parameters

struct snd_soc_component *component
    The component for which to initialize the regmap instance

struct regmap *regmap
    The regmap instance that should be used by the component

Description

This function allows deferred assignment of the regmap instance that is associated with the component. Only use this if the regmap instance is not yet ready when the component is registered. The function must also be called before the first IO attempt of the component.

void **snd_soc_component_exit_regmap**(struct snd_soc_component *component)

    De-initialize regmap instance for the component

Parameters

struct snd_soc_component *component
    The component for which to de-initialize the regmap instance

Description

Calls regmap_exit() on the regmap instance associated to the component and removes the regmap instance from the component.

This function should only be used if *snd_soc_component_init_regmap()* was used to initialize the regmap instance.

unsigned int **snd_soc_component_read**(struct snd_soc_component *component, unsigned int reg)

   Read register value

Parameters

struct snd_soc_component *component
   Component to read from

unsigned int reg
   Register to read

Return

read value

int **snd_soc_component_write**(struct snd_soc_component *component, unsigned int reg, unsigned int val)

   Write register value

Parameters

struct snd_soc_component *component
   Component to write to

unsigned int reg
   Register to write

unsigned int val
   Value to write to the register

Return

0 on success, a negative error code otherwise.

int **snd_soc_component_update_bits**(struct snd_soc_component *component, unsigned int reg, unsigned int mask, unsigned int val)

   Perform read/modify/write cycle

Parameters

struct snd_soc_component *component
   Component to update

unsigned int reg
   Register to update

unsigned int mask
   Mask that specifies which bits to update

unsigned int val
   New value for the bits specified by mask

Return

1 if the operation was successful and the value of the register changed, 0 if the operation was successful, but the value did not change. Returns a negative error code otherwise.

int **snd_soc_component_update_bits_async**(struct snd_soc_component *component, unsigned int reg, unsigned int mask, unsigned int val)

Perform asynchronous read/modify/write cycle

Parameters

struct snd_soc_component *component
    Component to update

unsigned int reg
    Register to update

unsigned int mask
    Mask that specifies which bits to update

unsigned int val
    New value for the bits specified by mask

Description

This function is similar to *snd_soc_component_update_bits()*, but the update operation is scheduled asynchronously. This means it may not be completed when the function returns. To make sure that all scheduled updates have been completed *snd_soc_component_async_complete()* must be called.

Return

1 if the operation was successful and the value of the register changed, 0 if the operation was successful, but the value did not change. Returns a negative error code otherwise.

unsigned int **snd_soc_component_read_field**(struct snd_soc_component *component, unsigned int reg, unsigned int mask)

    Read register field value

Parameters

struct snd_soc_component *component
    Component to read from

unsigned int reg
    Register to read

unsigned int mask
    mask of the register field

Return

read value of register field.

int **snd_soc_component_write_field**(struct snd_soc_component *component, unsigned int reg, unsigned int mask, unsigned int val)

    write to register field

Parameters

struct snd_soc_component *component
    Component to write to

unsigned int reg
    Register to write

unsigned int mask
    mask of the register field to update

unsigned int val
    value of the field to write

Return

1 for change, otherwise 0.

void **snd_soc_component_async_complete**(struct snd_soc_component *component)
    Ensure asynchronous I/O has completed

Parameters

struct snd_soc_component *component
    Component for which to wait

Description

This function blocks until all asynchronous I/O which has previously been scheduled using *snd_soc_component_update_bits_async()* has completed.

int **snd_soc_component_test_bits**(struct snd_soc_component *component, unsigned int reg, unsigned int mask, unsigned int value)
    Test register for change

Parameters

struct snd_soc_component *component
    component

unsigned int reg
    Register to test

unsigned int mask
    Mask that specifies which bits to test

unsigned int value
    Value to test against

Description

Tests a register with a new value and checks if the new value is different from the old value.

Return

1 for change, otherwise 0.

void **snd_soc_runtime_action**(struct snd_soc_pcm_runtime *rtd, int stream, int action)
    Increment/Decrement active count for PCM runtime components

Parameters

struct snd_soc_pcm_runtime *rtd
    ASoC PCM runtime that is activated

int stream
    Direction of the PCM stream

int action
    Activate stream if 1. Deactivate if -1.

Description

Increments/Decrements the active count for all the DAIs and components attached to a PCM runtime. Should typically be called when a stream is opened.

Must be called with the rtd->card->pcm_mutex being held

bool **snd_soc_runtime_ignore_pmdown_time**(struct snd_soc_pcm_runtime *rtd)
    Check whether to ignore the power down delay

Parameters

struct snd_soc_pcm_runtime *rtd
    The ASoC PCM runtime that should be checked.

Description

This function checks whether the power down delay should be ignored for a specific PCM runtime. Returns true if the delay is 0, if it the DAI link has been configured to ignore the delay, or if none of the components benefits from having the delay.

int **snd_soc_set_runtime_hwparams**(struct snd_pcm_substream *substream, const struct
                                     snd_pcm_hardware *hw)
    set the runtime hardware parameters

Parameters

struct snd_pcm_substream *substream
    the pcm substream

const struct snd_pcm_hardware *hw
    the hardware parameters

Description

Sets the substream runtime hardware parameters.

int **snd_soc_runtime_calc_hw**(struct snd_soc_pcm_runtime *rtd, struct snd_pcm_hardware
                               *hw, int stream)
    Calculate hw limits for a PCM stream

Parameters

struct snd_soc_pcm_runtime *rtd
    ASoC PCM runtime

struct snd_pcm_hardware *hw
    PCM hardware parameters (output)

int stream
    Direction of the PCM stream

Description

Calculates the subset of stream parameters supported by all DAIs associated with the PCM stream.

int **snd_soc_info_enum_double**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info
                                *uinfo)
    enumerated double mixer info callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_info *uinfo
    control element information

Description

Callback to provide information about a double enumerated mixer control.

Returns 0 for success.

int **snd_soc_get_enum_double**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
                              *ucontrol)
    enumerated double mixer get callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to get the value of a double enumerated mixer.

Returns 0 for success.

int **snd_soc_put_enum_double**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
                              *ucontrol)
    enumerated double mixer put callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to set the value of a double enumerated mixer.

Returns 0 for success.

int **snd_soc_read_signed**(struct snd_soc_component *component, unsigned int reg, unsigned
                          int mask, unsigned int shift, unsigned int sign_bit, int *signed_val)
    Read a codec register and interpret as signed value

Parameters

struct snd_soc_component *component
    component

unsigned int reg
    Register to read

unsigned int mask
    Mask to use after shifting the register value

unsigned int shift
    Right shift of register value

unsigned int sign_bit
    Bit that describes if a number is negative or not.

int *signed_val
    Pointer to where the read value should be stored

Description

This functions reads a codec register. The register value is shifted right by 'shift' bits and masked with the given 'mask'. Afterwards it translates the given registervalue into a signed integer if sign_bit is non-zero.

Returns 0 on success, otherwise an error value

int **snd_soc_info_volsw**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo)
    single mixer info callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_info *uinfo
    control element information

Description

Callback to provide information about a single mixer control, or a double mixer control that spans 2 registers.

Returns 0 for success.

int **snd_soc_info_volsw_sx**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo)
    Mixer info callback for SX TLV controls

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_info *uinfo
    control element information

Description

Callback to provide information about a single mixer control, or a double mixer control that spans 2 registers of the SX TLV type. SX TLV controls have a range that represents both positive and negative values either side of zero but without a sign bit. min is the minimum register value, max is the number of steps.

Returns 0 for success.

int **snd_soc_get_volsw**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)
    single mixer get callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to get the value of a single mixer control, or a double mixer control that spans 2 registers.

Returns 0 for success.

int **snd_soc_put_volsw**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)
    single mixer put callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to set the value of a single mixer control, or a double mixer control that spans 2 registers.

Returns 0 for success.

int **snd_soc_get_volsw_sx**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
                              *ucontrol)
    single mixer get callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to get the value of a single mixer control, or a double mixer control that spans 2 registers.

Returns 0 for success.

int **snd_soc_put_volsw_sx**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
                              *ucontrol)
    double mixer set callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to set the value of a double mixer control that spans 2 registers.

Returns 0 for success.

int **snd_soc_info_volsw_range**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info
*uinfo)

single mixer info callback with range.

Parameters

struct snd_kcontrol *kcontrol
mixer control

struct snd_ctl_elem_info *uinfo
control element information

Description

Callback to provide information, within a range, about a single mixer control.

returns 0 for success.

int **snd_soc_put_volsw_range**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
*ucontrol)

single mixer put value callback with range.

Parameters

struct snd_kcontrol *kcontrol
mixer control

struct snd_ctl_elem_value *ucontrol
control element information

Description

Callback to set the value, within a range, for a single mixer control.

Returns 0 for success.

int **snd_soc_get_volsw_range**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
*ucontrol)

single mixer get callback with range

Parameters

struct snd_kcontrol *kcontrol
mixer control

struct snd_ctl_elem_value *ucontrol
control element information

Description

Callback to get the value, within a range, of a single mixer control.

Returns 0 for success.

int **snd_soc_limit_volume**(struct snd_soc_card *card, const char *name, int max)

Set new limit to an existing volume control.

Parameters

struct snd_soc_card *card

where to look for the control

const char *name

Name of the control

int max

new maximum limit

Description

Return 0 for success, else error.

int **snd_soc_info_xr_sx**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo)

signed multi register info callback

Parameters

struct snd_kcontrol *kcontrol

mreg control

struct snd_ctl_elem_info *uinfo

control element information

Description

Callback to provide information of a control that can span multiple codec registers which together forms a single signed value in a MSB/LSB manner.

Returns 0 for success.

int **snd_soc_get_xr_sx**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)

signed multi register get callback

Parameters

struct snd_kcontrol *kcontrol

mreg control

struct snd_ctl_elem_value *ucontrol

control element information

Description

Callback to get the value of a control that can span multiple codec registers which together forms a single signed value in a MSB/LSB manner. The control supports specifying total no of bits used to allow for bitfields across the multiple codec registers.

Returns 0 for success.

int **snd_soc_put_xr_sx**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)

signed multi register get callback

Parameters

struct snd_kcontrol *kcontrol

mreg control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to set the value of a control that can span multiple codec registers which together forms a single signed value in a MSB/LSB manner. The control supports specifying total no of bits used to allow for bitfields across the multiple codec registers.

Returns 0 for success.

int **snd_soc_get_strobe**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)
    strobe get callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback get the value of a strobe mixer control.

Returns 0 for success.

int **snd_soc_put_strobe**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)
    strobe put callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback strobe a register bit to high then low (or the inverse) in one pass of a single mixer enum control.

Returns 1 for success.

int **snd_soc_new_compress**(struct snd_soc_pcm_runtime *rtd, int num)
    create a new compress.

Parameters

struct snd_soc_pcm_runtime *rtd
    The runtime for which we will create compress

int num
    the device index number (zero based - shared with normal PCMs)

Return

0 for success, else error.

ASoC DAPM API

struct snd_soc_dapm_widget ***snd_soc_dapm_kcontrol_widget**(struct snd_kcontrol
*kcontrol)

   Returns the widget associated to a kcontrol

Parameters

struct snd_kcontrol *kcontrol
   The kcontrol

struct snd_soc_dapm_context ***snd_soc_dapm_kcontrol_dapm**(struct snd_kcontrol *kcontrol)
   Returns the dapm context associated to a kcontrol

Parameters

struct snd_kcontrol *kcontrol
   The kcontrol

Note

This function must only be used on kcontrols that are known to have been registered for a
CODEC. Otherwise the behaviour is undefined.

int **snd_soc_dapm_force_bias_level**(struct snd_soc_dapm_context *dapm, enum
snd_soc_bias_level level)

   Sets the DAPM bias level

Parameters

struct snd_soc_dapm_context *dapm
   The DAPM context for which to set the level

enum snd_soc_bias_level level
   The level to set

Description

Forces the DAPM bias level to a specific state. It will call the bias level callback of DAPM
context with the specified level. This will even happen if the context is already at the same level.
Furthermore it will not go through the normal bias level sequencing, meaning any intermediate
states between the current and the target state will not be entered.

Note that the change in bias level is only temporary and the next time *snd_soc_dapm_sync()* is
called the state will be set to the level as determined by the DAPM core. The function is mainly
intended to be used to used during probe or resume from suspend to power up the device so
initialization can be done, before the DAPM core takes over.

int **snd_soc_dapm_set_bias_level**(struct snd_soc_dapm_context *dapm, enum
snd_soc_bias_level level)

   set the bias level for the system

Parameters

struct snd_soc_dapm_context *dapm
   DAPM context

enum snd_soc_bias_level level
   level to configure

Description

Configure the bias (power) levels for the SoC audio device.

Returns 0 for success else error.

int **snd_soc_dapm_dai_get_connected_widgets**(struct snd_soc_dai *dai, int stream, struct snd_soc_dapm_widget_list **list, bool (*custom_stop_condition)(struct snd_soc_dapm_widget*, enum snd_soc_dapm_direction))

> query audio path and it's widgets.

Parameters

struct snd_soc_dai *dai
> the soc DAI.

int stream
> stream direction.

struct snd_soc_dapm_widget_list **list
> list of active widgets for this stream.

bool (*custom_stop_condition)(struct snd_soc_dapm_widget *, enum snd_soc_dapm_direction)
> (optional) a function meant to stop the widget graph walk based on custom logic.

Description

Queries DAPM graph as to whether a valid audio stream path exists for the initial stream specified by name. This takes into account current mixer and mux kcontrol settings. Creates list of valid widgets.

Optionally, can be supplied with a function acting as a stopping condition. This function takes the dapm widget currently being examined and the walk direction as an arguments, it should return true if the walk should be stopped and false otherwise.

Returns the number of valid paths or negative error.

void **snd_soc_dapm_free_widget**(struct snd_soc_dapm_widget *w)
> Free specified widget

Parameters

struct snd_soc_dapm_widget *w
> widget to free

Description

Removes widget from all paths and frees memory occupied by it.

int **snd_soc_dapm_sync_unlocked**(struct snd_soc_dapm_context *dapm)
> scan and power dapm paths

Parameters

struct snd_soc_dapm_context *dapm
> DAPM context

Description

Walks all dapm audio paths and powers widgets according to their stream or path usage.

Requires external locking.

Returns 0 for success.

int **snd_soc_dapm_sync**(struct snd_soc_dapm_context *dapm)
    scan and power dapm paths

Parameters

struct snd_soc_dapm_context *dapm
    DAPM context

Description

Walks all dapm audio paths and powers widgets according to their stream or path usage.

Returns 0 for success.

int **snd_soc_dapm_add_routes**(struct snd_soc_dapm_context *dapm, const struct
                              snd_soc_dapm_route *route, int num)
    Add routes between DAPM widgets

Parameters

struct snd_soc_dapm_context *dapm
    DAPM context

const struct snd_soc_dapm_route *route
    audio routes

int num
    number of routes

Description

Connects 2 dapm widgets together via a named audio path. The sink is the widget receiving the audio signal, whilst the source is the sender of the audio signal.

Returns 0 for success else error. On error all resources can be freed with a call to snd_soc_card_free().

int **snd_soc_dapm_del_routes**(struct snd_soc_dapm_context *dapm, const struct
                              snd_soc_dapm_route *route, int num)
    Remove routes between DAPM widgets

Parameters

struct snd_soc_dapm_context *dapm
    DAPM context

const struct snd_soc_dapm_route *route
    audio routes

int num
    number of routes

Description

Removes routes from the DAPM context.

int **snd_soc_dapm_weak_routes**(struct snd_soc_dapm_context *dapm, const struct
snd_soc_dapm_route *route, int num)

>   Mark routes between DAPM widgets as weak

Parameters

struct snd_soc_dapm_context *dapm
>   DAPM context

const struct snd_soc_dapm_route *route
>   audio routes

int num
>   number of routes

Description

Mark existing routes matching those specified in the passed array as being weak, meaning that
they are ignored for the purpose of power decisions. The main intended use case is for sidetone
paths which couple audio between other independent paths if they are both active in order to
make the combination work better at the user level but which aren't intended to be "used".

Note that CODEC drivers should not use this as sidetone type paths can frequently also be used
as bypass paths.

int **snd_soc_dapm_new_widgets**(struct snd_soc_card *card)

>   add new dapm widgets

Parameters

struct snd_soc_card *card
>   card to be checked for new dapm widgets

Description

Checks the codec for any new dapm widgets and creates them if found.

Returns 0 for success.

int **snd_soc_dapm_get_volsw**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
*ucontrol)

>   dapm mixer get callback

Parameters

struct snd_kcontrol *kcontrol
>   mixer control

struct snd_ctl_elem_value *ucontrol
>   control element information

Description

Callback to get the value of a dapm mixer control.

Returns 0 for success.

int **snd_soc_dapm_put_volsw**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
*ucontrol)

>   dapm mixer set callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to set the value of a dapm mixer control.

Returns 0 for success.

int **snd_soc_dapm_get_enum_double**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)
    dapm enumerated double mixer get callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to get the value of a dapm enumerated double mixer control.

Returns 0 for success.

int **snd_soc_dapm_put_enum_double**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value *ucontrol)
    dapm enumerated double mixer set callback

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_value *ucontrol
    control element information

Description

Callback to set the value of a dapm enumerated double mixer control.

Returns 0 for success.

int **snd_soc_dapm_info_pin_switch**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo)
    Info for a pin switch

Parameters

struct snd_kcontrol *kcontrol
    mixer control

struct snd_ctl_elem_info *uinfo
    control element information

Description

Callback to provide information about a pin switch control.

int **snd_soc_dapm_get_pin_switch**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
*ucontrol)

Get information for a pin switch

Parameters

struct snd_kcontrol *kcontrol
mixer control

struct snd_ctl_elem_value *ucontrol
Value

int **snd_soc_dapm_put_pin_switch**(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_value
*ucontrol)

Set information for a pin switch

Parameters

struct snd_kcontrol *kcontrol
mixer control

struct snd_ctl_elem_value *ucontrol
Value

struct snd_soc_dapm_widget ***snd_soc_dapm_new_control**(struct snd_soc_dapm_context
*dapm, const struct
snd_soc_dapm_widget *widget)

create new dapm control

Parameters

struct snd_soc_dapm_context *dapm
DAPM context

const struct snd_soc_dapm_widget *widget
widget template

Description

Creates new DAPM control based upon a template.

Returns a widget pointer on success or an error pointer on failure

int **snd_soc_dapm_new_controls**(struct snd_soc_dapm_context *dapm, const struct
snd_soc_dapm_widget *widget, int num)

create new dapm controls

Parameters

struct snd_soc_dapm_context *dapm
DAPM context

const struct snd_soc_dapm_widget *widget
widget array

int num
number of widgets

Description

Creates new DAPM controls based upon the templates.

Returns 0 for success else error.

int **snd_soc_dapm_new_dai_widgets**(struct snd_soc_dapm_context *dapm, struct snd_soc_dai *dai)

> Create new DAPM widgets

Parameters

struct snd_soc_dapm_context *dapm
> DAPM context

struct snd_soc_dai *dai
> parent DAI

Description

Returns 0 on success, error code otherwise.

void **snd_soc_dapm_stream_event**(struct snd_soc_pcm_runtime *rtd, int stream, int event)
> send a stream event to the dapm core

Parameters

struct snd_soc_pcm_runtime *rtd
> PCM runtime data

int stream
> stream name

int event
> stream event

Description

Sends a stream event to the dapm core. The core then makes any necessary widget power changes.

Returns 0 for success else error.

int **snd_soc_dapm_enable_pin_unlocked**(struct snd_soc_dapm_context *dapm, const char *pin)

> enable pin.

Parameters

struct snd_soc_dapm_context *dapm
> DAPM context

const char *pin
> pin name

Description

Enables input/output pin and its parents or children widgets iff there is a valid audio route and active audio stream.

Requires external locking.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_enable_pin**(struct snd_soc_dapm_context *dapm, const char *pin)

   enable pin.

Parameters

struct snd_soc_dapm_context *dapm
   DAPM context

const char *pin
   pin name

Description

Enables input/output pin and its parents or children widgets iff there is a valid audio route and active audio stream.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_force_enable_pin_unlocked**(struct snd_soc_dapm_context *dapm, const char *pin)

   force a pin to be enabled

Parameters

struct snd_soc_dapm_context *dapm
   DAPM context

const char *pin
   pin name

Description

Enables input/output pin regardless of any other state. This is intended for use with microphone bias supplies used in microphone jack detection.

Requires external locking.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_force_enable_pin**(struct snd_soc_dapm_context *dapm, const char *pin)

   force a pin to be enabled

Parameters

struct snd_soc_dapm_context *dapm
   DAPM context

const char *pin
   pin name

Description

Enables input/output pin regardless of any other state. This is intended for use with microphone bias supplies used in microphone jack detection.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_disable_pin_unlocked**(struct snd_soc_dapm_context *dapm, const char *pin)

> disable pin.

Parameters

struct snd_soc_dapm_context *dapm
> DAPM context

const char *pin
> pin name

Description

Disables input/output pin and its parents or children widgets.

Requires external locking.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_disable_pin**(struct snd_soc_dapm_context *dapm, const char *pin)

> disable pin.

Parameters

struct snd_soc_dapm_context *dapm
> DAPM context

const char *pin
> pin name

Description

Disables input/output pin and its parents or children widgets.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_nc_pin_unlocked**(struct snd_soc_dapm_context *dapm, const char *pin)

> permanently disable pin.

Parameters

struct snd_soc_dapm_context *dapm
> DAPM context

const char *pin
> pin name

Description

Marks the specified pin as being not connected, disabling it along any parent or child widgets.
At present this is identical to *snd_soc_dapm_disable_pin()* but in future it will be extended to
do additional things such as disabling controls which only affect paths through the pin.

Requires external locking.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_nc_pin**(struct snd_soc_dapm_context *dapm, const char *pin)

    permanently disable pin.

Parameters

struct snd_soc_dapm_context *dapm
    DAPM context

const char *pin
    pin name

Description

Marks the specified pin as being not connected, disabling it along any parent or child widgets. At present this is identical to *snd_soc_dapm_disable_pin()* but in future it will be extended to do additional things such as disabling controls which only affect paths through the pin.

NOTE

snd_soc_dapm_sync() needs to be called after this for DAPM to do any widget power switching.

int **snd_soc_dapm_get_pin_status**(struct snd_soc_dapm_context *dapm, const char *pin)

    get audio pin status

Parameters

struct snd_soc_dapm_context *dapm
    DAPM context

const char *pin
    audio signal pin endpoint (or start point)

Description

Get audio pin status - connected or disconnected.

Returns 1 for connected otherwise 0.

int **snd_soc_dapm_ignore_suspend**(struct snd_soc_dapm_context *dapm, const char *pin)

    ignore suspend status for DAPM endpoint

Parameters

struct snd_soc_dapm_context *dapm
    DAPM context

const char *pin
    audio signal pin endpoint (or start point)

Description

Mark the given endpoint or pin as ignoring suspend. When the system is disabled a path between two endpoints flagged as ignoring suspend will not be disabled. The path must already be enabled via normal means at suspend time, it will not be turned on if it was not already enabled.

void **snd_soc_dapm_free**(struct snd_soc_dapm_context *dapm)

> free dapm resources

Parameters

struct snd_soc_dapm_context *dapm
> DAPM context

Description

Free all dapm widgets and resources.

ASoC DMA Engine API

int **snd_dmaengine_pcm_prepare_slave_config**(struct snd_pcm_substream *substream,
                                               struct snd_pcm_hw_params *params, struct
                                               dma_slave_config *slave_config)

> Generic prepare_slave_config callback

Parameters

struct snd_pcm_substream *substream
> PCM substream

struct snd_pcm_hw_params *params
> hw_params

struct dma_slave_config *slave_config
> DMA slave config to prepare

Description

This function can be used as a generic prepare_slave_config callback for platforms which make use of the snd_dmaengine_dai_dma_data struct for their DAI DMA data. Internally the function will first call snd_hwparams_to_dma_slave_config to fill in the slave config based on the hw_params, followed by snd_dmaengine_pcm_set_config_from_dai_data to fill in the remaining fields based on the DAI DMA data.

int **snd_dmaengine_pcm_register**(struct device *dev, const struct
                                   snd_dmaengine_pcm_config *config, unsigned int flags)

> Register a dmaengine based PCM device

Parameters

struct device *dev
> The parent device for the PCM device

const struct snd_dmaengine_pcm_config *config
> Platform specific PCM configuration

unsigned int flags
> Platform specific quirks

void **snd_dmaengine_pcm_unregister**(struct device *dev)

> Removes a dmaengine based PCM device

Parameters

struct device *dev
    Parent device the PCM was register with

Description

Removes a dmaengine based PCM device previously registered with snd_dmaengine_pcm_register.

1.1.8 Miscellaneous Functions

Hardware-Dependent Devices API

int **snd_hwdep_new**(struct snd_card *card, char *id, int device, struct snd_hwdep **rhwdep)
    create a new hwdep instance

Parameters

struct snd_card *card
    the card instance

char *id
    the id string

int device
    the device index (zero-based)

struct snd_hwdep **rhwdep
    the pointer to store the new hwdep instance

Description

Creates a new hwdep instance with the given index on the card. The callbacks (hwdep->ops) must be set on the returned instance after this call manually by the caller.

Return

Zero if successful, or a negative error code on failure.

Jack Abstraction Layer API

enum **snd_jack_types**
    Jack types which can be reported

Constants

SND_JACK_HEADPHONE
    Headphone

SND_JACK_MICROPHONE
    Microphone

SND_JACK_HEADSET
    Headset

SND_JACK_LINEOUT
    Line out

SND_JACK_MECHANICAL
    Mechanical switch

SND_JACK_VIDEOOUT
    Video out

SND_JACK_AVOUT
    AV (Audio Video) out

SND_JACK_LINEIN
    Line in

SND_JACK_BTN_0
    Button 0

SND_JACK_BTN_1
    Button 1

SND_JACK_BTN_2
    Button 2

SND_JACK_BTN_3
    Button 3

SND_JACK_BTN_4
    Button 4

SND_JACK_BTN_5
    Button 5

Description

These values are used as a bitmask.

Note that this must be kept in sync with the lookup table in sound/core/jack.c.

int **snd_jack_add_new_kctl**(struct snd_jack *jack, const char *name, int mask)
    Create a new snd_jack_kctl and add it to jack

Parameters

struct snd_jack *jack
    the jack instance which the kctl will attaching to

const char * name
    the name for the snd_kcontrol object

int mask
    a bitmask of enum snd_jack_type values that can be detected by this snd_jack_kctl object.

Description

Creates a new snd_kcontrol object and adds it to the jack kctl_list.

Return

Zero if successful, or a negative error code on failure.

int **snd_jack_new**(struct snd_card *card, const char *id, int type, struct snd_jack **jjack, bool
                initial_kctl, bool phantom_jack)
    Create a new jack

Parameters

struct snd_card *card
    the card instance

const char *id
    an identifying string for this jack

int type
    a bitmask of enum snd_jack_type values that can be detected by this jack

struct snd_jack **jjack
    Used to provide the allocated jack object to the caller.

bool initial_kctl
    if true, create a kcontrol and add it to the jack list.

bool phantom_jack
    Don't create a input device for phantom jacks.

Description

Creates a new jack object.

Return

Zero if successful, or a negative error code on failure. On success **jjack** will be initialised.

void **snd_jack_set_parent**(struct snd_jack *jack, struct device *parent)
    Set the parent device for a jack

Parameters

struct snd_jack *jack
    The jack to configure

struct device *parent
    The device to set as parent for the jack.

Description

Set the parent for the jack devices in the device tree. This function is only valid prior to registration of the jack. If no parent is configured then the parent device will be the sound card.

int **snd_jack_set_key**(struct snd_jack *jack, enum *snd_jack_types* type, int keytype)
    Set a key mapping on a jack

Parameters

struct snd_jack *jack
    The jack to configure

enum snd_jack_types type
    Jack report type for this key

int keytype
    Input layer key type to be reported

Description

Map a SND_JACK_BTN_* button type to an input layer key, allowing reporting of keys on accessories via the jack abstraction. If no mapping is provided but keys are enabled in the jack type then BTN_n numeric buttons will be reported.

If jacks are not reporting via the input API this call will have no effect.

Note that this is intended to be use by simple devices with small numbers of keys that can be reported. It is also possible to access the input device directly - devices with complex input capabilities on accessories should consider doing this rather than using this abstraction.

This function may only be called prior to registration of the jack.

Return

Zero if successful, or a negative error code on failure.

void **snd_jack_report**(struct snd_jack *jack, int status)
>   Report the current status of a jack

Parameters

struct snd_jack *jack
>   The jack to report status for

int status
>   The current status of the jack

Note

This function uses mutexes and should be called from a context which can sleep (such as a workqueue).

void **snd_soc_jack_report**(struct snd_soc_jack *jack, int status, int mask)
>   Report the current status for a jack

Parameters

struct snd_soc_jack *jack
>   the jack

int status
>   a bitmask of enum snd_jack_type values that are currently detected.

int mask
>   a bitmask of enum snd_jack_type values that being reported.

Description

If configured using *snd_soc_jack_add_pins()* then the associated DAPM pins will be enabled or disabled as appropriate and DAPM synchronised.

Note

This function uses mutexes and should be called from a context which can sleep (such as a workqueue).

int **snd_soc_jack_add_zones**(struct snd_soc_jack *jack, int count, struct snd_soc_jack_zone *zones)
>   Associate voltage zones with jack

Parameters

struct snd_soc_jack *jack
    ASoC jack

int count
    Number of zones

struct snd_soc_jack_zone *zones
    Array of zones

Description

After this function has been called the zones specified in the array will be associated with the jack.

int **snd_soc_jack_get_type**(struct snd_soc_jack *jack, int micbias_voltage)
    Based on the mic bias value, this function returns the type of jack from the zones declared in the jack type

Parameters

struct snd_soc_jack *jack
    ASoC jack

int micbias_voltage
    mic bias voltage at adc channel when jack is plugged in

Description

Based on the mic bias value passed, this function helps identify the type of jack from the already declared jack zones

int **snd_soc_jack_add_pins**(struct snd_soc_jack *jack, int count, struct snd_soc_jack_pin *pins)
    Associate DAPM pins with an ASoC jack

Parameters

struct snd_soc_jack *jack
    ASoC jack created with snd_soc_card_jack_new_pins()

int count
    Number of pins

struct snd_soc_jack_pin *pins
    Array of pins

Description

After this function has been called the DAPM pins specified in the pins array will have their status updated to reflect the current state of the jack whenever the jack status is updated.

void **snd_soc_jack_notifier_register**(struct snd_soc_jack *jack, struct notifier_block *nb)
    Register a notifier for jack status

Parameters

struct snd_soc_jack *jack
    ASoC jack

struct notifier_block *nb
    Notifier block to register

Description

Register for notification of the current status of the jack. Note that it is not possible to report additional jack events in the callback from the notifier, this is intended to support applications such as enabling electrical detection only when a mechanical detection event has occurred.

void **snd_soc_jack_notifier_unregister**(struct snd_soc_jack *jack, struct notifier_block *nb)

>   Unregister a notifier for jack status

Parameters

struct snd_soc_jack *jack
>   ASoC jack

struct notifier_block *nb
>   Notifier block to unregister

Description

Stop notifying for status changes.

int **snd_soc_jack_add_gpios**(struct snd_soc_jack *jack, int count, struct snd_soc_jack_gpio *gpios)

>   Associate GPIO pins with an ASoC jack

Parameters

struct snd_soc_jack *jack
>   ASoC jack

int count
>   number of pins

struct snd_soc_jack_gpio *gpios
>   array of gpio pins

Description

This function will request gpio, set data direction and request irq for each gpio in the array.

int **snd_soc_jack_add_gpiods**(struct device *gpiod_dev, struct snd_soc_jack *jack, int count, struct snd_soc_jack_gpio *gpios)

>   Associate GPIO descriptor pins with an ASoC jack

Parameters

struct device *gpiod_dev
>   GPIO consumer device

struct snd_soc_jack *jack
>   ASoC jack

int count
>   number of pins

struct snd_soc_jack_gpio *gpios
>   array of gpio pins

Description

This function will request gpio, set data direction and request irq for each gpio in the array.

void **snd_soc_jack_free_gpios**(struct snd_soc_jack *jack, int count, struct snd_soc_jack_gpio *gpios)

> Release GPIO pins' resources of an ASoC jack

Parameters

struct snd_soc_jack *jack
> ASoC jack

int count
> number of pins

struct snd_soc_jack_gpio *gpios
> array of gpio pins

Description

Release gpio and irq resources for gpio pins associated with an ASoC jack.

ISA DMA Helpers

void **snd_dma_program**(unsigned long dma, unsigned long addr, unsigned int size, unsigned short mode)

> program an ISA DMA transfer

Parameters

unsigned long dma
> the dma number

unsigned long addr
> the physical address of the buffer

unsigned int size
> the DMA transfer size

unsigned short mode
> the DMA transfer mode, DMA_MODE_XXX

Description

Programs an ISA DMA transfer for the given buffer.

void **snd_dma_disable**(unsigned long dma)

> stop the ISA DMA transfer

Parameters

unsigned long dma
> the dma number

Description

Stops the ISA DMA transfer.

unsigned int **snd_dma_pointer**(unsigned long dma, unsigned int size)

> return the current pointer to DMA transfer buffer in bytes

Parameters

unsigned long dma
    the dma number

unsigned int size
    the dma transfer size

Return

The current pointer in DMA transfer buffer in bytes.

int **snd_devm_request_dma**(struct device *dev, int dma, const char *name)
    the managed version of request_dma()

Parameters

struct device *dev
    the device pointer

int dma
    the dma number

const char *name
    the name string of the requester

Description

The requested DMA will be automatically released at unbinding via devres.

Return

zero on success, or a negative error code

Other Helper Macros

void **snd_power_ref**(struct snd_card *card)
    Take the reference count for power control

Parameters

struct snd_card *card
    sound card object

Description

The power_ref reference of the card is used for managing to block the *snd_power_sync_ref()* operation. This function increments the reference. The counterpart *snd_power_unref()* has to be called appropriately later.

void **snd_power_unref**(struct snd_card *card)
    Release the reference count for power control

Parameters

struct snd_card *card
    sound card object

void **snd_power_sync_ref**(struct snd_card *card)
    wait until the card power_ref is freed

Parameters

struct snd_card *card
    sound card object

Description

This function is used to synchronize with the pending power_ref being released.

void **snd_card_unref**(struct snd_card *card)
    Unreference the card object

Parameters

struct snd_card *card
    the card object to unreference

Description

Call this function for the card object that was obtained via *snd_card_ref()* or *snd_lookup_minor_data()*.

snd_printk

snd_printk (fmt, ...)
    printk wrapper

Parameters

fmt
    format string

...
    variable arguments

Description

Works like printk() but prints the file and the line of the caller when configured with CONFIG_SND_VERBOSE_PRINTK.

snd_printd

snd_printd (fmt, ...)
    debug printk

Parameters

fmt
    format string

...
    variable arguments

Description

Works like *snd_printk()* for debugging purposes. Ignored when CONFIG_SND_DEBUG is not set.

snd_BUG

snd_BUG ()
    give a BUG warning message and stack trace

Parameters

Description

Calls WARN() if CONFIG_SND_DEBUG is set. Ignored when CONFIG_SND_DEBUG is not set.

snd_printd_ratelimit

snd_printd_ratelimit ()

>   Suppress high rates of output when CONFIG_SND_DEBUG is enabled.

Parameters

snd_BUG_ON

snd_BUG_ON (cond)

>   debugging check macro

Parameters

cond
>   condition to evaluate

Description

Has the same behavior as WARN_ON when CONFIG_SND_DEBUG is set, otherwise just evaluates the conditional and returns the value.

snd_printdd

snd_printdd (format, ...)

>   debug printk

Parameters

format
>   format string

...
>   variable arguments

Description

Works like *snd_printk()* for debugging purposes. Ignored when CONFIG_SND_DEBUG_VERBOSE is not set.

int **register_sound_special_device**(const struct file_operations *fops, int unit, struct device *dev)

>   register a special sound node

Parameters

const struct file_operations *fops
>   File operations for the driver

int unit
>   Unit number to allocate

struct device *dev
>   device pointer

Allocate a special sound device by minor number from the sound subsystem.

Return

The allocated number is returned on success. On failure,
     a negative error code is returned.

int **register_sound_mixer**(const struct file_operations *fops, int dev)
     register a mixer device

Parameters

const struct file_operations *fops
     File operations for the driver

int dev
     Unit number to allocate

     Allocate a mixer device. Unit is the number of the mixer requested. Pass -1 to request the next free mixer unit.

Return

On success, the allocated number is returned. On failure,
     a negative error code is returned.

int **register_sound_dsp**(const struct file_operations *fops, int dev)
     register a DSP device

Parameters

const struct file_operations *fops
     File operations for the driver

int dev
     Unit number to allocate

     Allocate a DSP device. Unit is the number of the DSP requested. Pass -1 to request the next free DSP unit.

     This function allocates both the audio and dsp device entries together and will always allocate them as a matching pair - eg dsp3/audio3

Return

On success, the allocated number is returned. On failure,
     a negative error code is returned.

void **unregister_sound_special**(int unit)
     unregister a special sound device

Parameters

int unit
     unit number to allocate

     Release a sound device that was allocated with register_sound_special(). The unit passed is the return value from the register function.

void **unregister_sound_mixer**(int unit)
     unregister a mixer

Parameters

int unit
    unit number to allocate

    Release a sound device that was allocated with *register_sound_mixer()*. The unit passed
    is the return value from the register function.

void **unregister_sound_dsp**(int unit)
    unregister a DSP device

Parameters

int unit
    unit number to allocate

    Release a sound device that was allocated with *register_sound_dsp()*. The unit passed
    is the return value from the register function.

    Both of the allocated units are released together automatically.

1.2 Writing an ALSA Driver

Author
    Takashi Iwai <tiwai@suse.de>

1.2.1 Preface

This document describes how to write an ALSA (Advanced Linux Sound Architecture) driver.
The document focuses mainly on PCI soundcards. In the case of other device types, the API
might be different, too. However, at least the ALSA kernel API is consistent, and therefore it
would be still a bit help for writing them.

This document targets people who already have enough C language skills and have basic linux
kernel programming knowledge. This document doesn't explain the general topic of linux kernel
coding and doesn't cover low-level driver implementation details. It only describes the standard
way to write a PCI sound driver on ALSA.

1.2.2 File Tree Structure

General

The file tree structure of ALSA driver is depicted below:

```
sound
        /core
                /oss
                /seq
                        /oss
        /include
        /drivers
                /mpu401
```

```
                /opl3
        /i2c
        /synth
                /emux
        /pci
                /(cards)
        /isa
                /(cards)
        /arm
        /ppc
        /sparc
        /usb
        /pcmcia /(cards)
        /soc
        /oss
```

core directory

This directory contains the middle layer which is the heart of ALSA drivers. In this directory,
the native ALSA modules are stored. The sub-directories contain different modules and are
dependent upon the kernel config.

core/oss

The code for OSS PCM and mixer emulation modules is stored in this directory. The OSS
rawmidi emulation is included in the ALSA rawmidi code since it's quite small. The sequencer
code is stored in core/seq/oss directory (see *below*).

core/seq

This directory and its sub-directories are for the ALSA sequencer. This directory contains the
sequencer core and primary sequencer modules such as snd-seq-midi, snd-seq-virmidi, etc.
They are compiled only when CONFIG_SND_SEQUENCER is set in the kernel config.

core/seq/oss

This contains the OSS sequencer emulation code.

include directory

This is the place for the public header files of ALSA drivers, which are to be exported to user-space, or included by several files in different directories. Basically, the private header files should not be placed in this directory, but you may still find files there, due to historical reasons :)

drivers directory

This directory contains code shared among different drivers on different architectures. They are hence supposed not to be architecture-specific. For example, the dummy PCM driver and the serial MIDI driver are found in this directory. In the sub-directories, there is code for components which are independent from bus and cpu architectures.

drivers/mpu401

The MPU401 and MPU401-UART modules are stored here.

drivers/opl3 and opl4

The OPL3 and OPL4 FM-synth stuff is found here.

i2c directory

This contains the ALSA i2c components.

Although there is a standard i2c layer on Linux, ALSA has its own i2c code for some cards, because the soundcard needs only a simple operation and the standard i2c API is too complicated for such a purpose.

synth directory

This contains the synth middle-level modules.

So far, there is only Emu8000/Emu10k1 synth driver under the `synth/emux` sub-directory.

pci directory

This directory and its sub-directories hold the top-level card modules for PCI soundcards and the code specific to the PCI BUS.

The drivers compiled from a single file are stored directly in the pci directory, while the drivers with several source files are stored on their own sub-directory (e.g. emu10k1, ice1712).

isa directory

This directory and its sub-directories hold the top-level card modules for ISA soundcards.

arm, ppc, and sparc directories

They are used for top-level card modules which are specific to one of these architectures.

usb directory

This directory contains the USB-audio driver. The USB MIDI driver is integrated in the usb-audio driver.

pcmcia directory

The PCMCIA, especially PCCard drivers will go here. CardBus drivers will be in the pci directory, because their API is identical to that of standard PCI cards.

soc directory

This directory contains the codes for ASoC (ALSA System on Chip) layer including ASoC core, codec and machine drivers.

oss directory

This contains OSS/Lite code. At the time of writing, all code has been removed except for dmasound on m68k.

1.2.3 Basic Flow for PCI Drivers

Outline

The minimum flow for PCI soundcards is as follows:
- define the PCI ID table (see the section *PCI Entries*).
- create `probe` callback.
- create `remove` callback.
- create a struct pci_driver structure containing the three pointers above.
- create an `init` function just calling the `pci_register_driver()` to register the pci_driver table defined above.
- create an `exit` function to call the `pci_unregister_driver()` function.

Full Code Example

The code example is shown below. Some parts are kept unimplemented at this moment but will be filled in the next sections. The numbers in the comment lines of the snd_mychip_probe() function refer to details explained in the following section.

```
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <sound/core.h>
#include <sound/initval.h>

/* module parameters (see "Module Parameters") */
/* SNDRV_CARDS: maximum number of cards supported by this module */
static int index[SNDRV_CARDS] = SNDRV_DEFAULT_IDX;
static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;
static bool enable[SNDRV_CARDS] = SNDRV_DEFAULT_ENABLE_PNP;

/* definition of the chip-specific record */
struct mychip {
        struct snd_card *card;
        /* the rest of the implementation will be in section
         * "PCI Resource Management"
         */
};

/* chip-specific destructor
 * (see "PCI Resource Management")
 */
static int snd_mychip_free(struct mychip *chip)
{
        .... /* will be implemented later... */
}

/* component-destructor
 * (see "Management of Cards and Components")
 */
static int snd_mychip_dev_free(struct snd_device *device)
{
        return snd_mychip_free(device->device_data);
}

/* chip-specific constructor
 * (see "Management of Cards and Components")
 */
static int snd_mychip_create(struct snd_card *card,
                             struct pci_dev *pci,
                             struct mychip **rchip)
{
        struct mychip *chip;
        int err;
```

```
        static const struct snd_device_ops ops = {
                .dev_free = snd_mychip_dev_free,
        };

        *rchip = NULL;

        /* check PCI availability here
         * (see "PCI Resource Management")
         */
        ....

        /* allocate a chip-specific data with zero filled */
        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        if (chip == NULL)
                return -ENOMEM;

        chip->card = card;

        /* rest of initialization here; will be implemented
         * later, see "PCI Resource Management"
         */
        ....

        err = snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
        if (err < 0) {
                snd_mychip_free(chip);
                return err;
        }

        *rchip = chip;
        return 0;
}

/* constructor -- see "Driver Constructor" sub-section */
static int snd_mychip_probe(struct pci_dev *pci,
                            const struct pci_device_id *pci_id)
{
        static int dev;
        struct snd_card *card;
        struct mychip *chip;
        int err;

        /* (1) */
        if (dev >= SNDRV_CARDS)
                return -ENODEV;
        if (!enable[dev]) {
                dev++;
                return -ENOENT;
        }
```

```
        /* (2) */
        err = snd_card_new(&pci->dev, index[dev], id[dev], THIS_MODULE,
                           0, &card);
        if (err < 0)
                return err;

        /* (3) */
        err = snd_mychip_create(card, pci, &chip);
        if (err < 0)
                goto error;

        /* (4) */
        strcpy(card->driver, "My Chip");
        strcpy(card->shortname, "My Own Chip 123");
        sprintf(card->longname, "%s at 0x%lx irq %i",
                card->shortname, chip->port, chip->irq);

        /* (5) */
        .... /* implemented later */

        /* (6) */
        err = snd_card_register(card);
        if (err < 0)
                goto error;

        /* (7) */
        pci_set_drvdata(pci, card);
        dev++;
        return 0;

error:
        snd_card_free(card);
        return err;
}

/* destructor -- see the "Destructor" sub-section */
static void snd_mychip_remove(struct pci_dev *pci)
{
        snd_card_free(pci_get_drvdata(pci));
}
```

Driver Constructor

The real constructor of PCI drivers is the `probe` callback. The `probe` callback and other component-constructors which are called from the `probe` callback cannot be used with the `__init` prefix because any PCI device could be a hotplug device.

In the `probe` callback, the following scheme is often used.

1) Check and increment the device index.

```
static int dev;
....
if (dev >= SNDRV_CARDS)
        return -ENODEV;
if (!enable[dev]) {
        dev++;
        return -ENOENT;
}
```

where `enable[dev]` is the module option.

Each time the `probe` callback is called, check the availability of the device. If not available, simply increment the device index and return. dev will be incremented also later (*step 7*).

2) Create a card instance

```
struct snd_card *card;
int err;
....
err = snd_card_new(&pci->dev, index[dev], id[dev], THIS_MODULE,
                   0, &card);
```

The details will be explained in the section *Management of Cards and Components*.

3) Create a main component

In this part, the PCI resources are allocated:

```
struct mychip *chip;
....
err = snd_mychip_create(card, pci, &chip);
if (err < 0)
        goto error;
```

The details will be explained in the section *PCI Resource Management*.

When something goes wrong, the probe function needs to deal with the error. In this example, we have a single error handling path placed at the end of the function:

```
error:
        snd_card_free(card);
        return err;
```

Since each component can be properly freed, the single *snd_card_free()* call should suffice in most cases.

4) Set the driver ID and name strings.

```
strcpy(card->driver, "My Chip");
strcpy(card->shortname, "My Own Chip 123");
sprintf(card->longname, "%s at 0x%lx irq %i",
        card->shortname, chip->port, chip->irq);
```

The driver field holds the minimal ID string of the chip. This is used by alsa-lib's configurator, so keep it simple but unique. Even the same driver can have different driver IDs to distinguish the functionality of each chip type.

The shortname field is a string shown as more verbose name. The longname field contains the information shown in /proc/asound/cards.

5) Create other components, such as mixer, MIDI, etc.

Here you define the basic components such as *PCM*, mixer (e.g. *AC97*), MIDI (e.g. *MPU-401*), and other interfaces. Also, if you want a *proc file*, define it here, too.

6) Register the card instance.

```
err = snd_card_register(card);
if (err < 0)
        goto error;
```

Will be explained in the section *Management of Cards and Components*, too.

7) Set the PCI driver data and return zero.

```
pci_set_drvdata(pci, card);
dev++;
return 0;
```

In the above, the card record is stored. This pointer is used in the remove callback and power-management callbacks, too.

Destructor

The destructor, the remove callback, simply releases the card instance. Then the ALSA middle layer will release all the attached components automatically.

It would be typically just calling *snd_card_free()*:

```
static void snd_mychip_remove(struct pci_dev *pci)
{
        snd_card_free(pci_get_drvdata(pci));
}
```

The above code assumes that the card pointer is set to the PCI driver data.

Header Files

For the above example, at least the following include files are necessary:

```
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <sound/core.h>
#include <sound/initval.h>
```

where the last one is necessary only when module options are defined in the source file. If the code is split into several files, the files without module options don't need them.

In addition to these headers, you'll need `<linux/interrupt.h>` for interrupt handling, and `<linux/io.h>` for I/O access. If you use the `mdelay()` or `udelay()` functions, you'll need to include `<linux/delay.h>` too.

The ALSA interfaces like the PCM and control APIs are defined in other `<sound/xxx.h>` header files. They have to be included after `<sound/core.h>`.

1.2.4 Management of Cards and Components

Card Instance

For each soundcard, a "card" record must be allocated.

A card record is the headquarters of the soundcard. It manages the whole list of devices (components) on the soundcard, such as PCM, mixers, MIDI, synthesizer, and so on. Also, the card record holds the ID and the name strings of the card, manages the root of proc files, and controls the power-management states and hotplug disconnections. The component list on the card record is used to manage the correct release of resources at destruction.

As mentioned above, to create a card instance, call *snd_card_new()*:

```
struct snd_card *card;
int err;
err = snd_card_new(&pci->dev, index, id, module, extra_size, &card);
```

The function takes six arguments: the parent device pointer, the card-index number, the id string, the module pointer (usually `THIS_MODULE`), the size of extra-data space, and the pointer to return the card instance. The extra_size argument is used to allocate card->private_data for the chip-specific data. Note that these data are allocated by *snd_card_new()*.

The first argument, the pointer of struct device, specifies the parent device. For PCI devices, typically `&pci->` is passed there.

Components

After the card is created, you can attach the components (devices) to the card instance. In an ALSA driver, a component is represented as a struct snd_device object. A component can be a PCM instance, a control interface, a raw MIDI interface, etc. Each such instance has one component entry.

A component can be created via the *snd_device_new()* function:

```
snd_device_new(card, SNDRV_DEV_XXX, chip, &ops);
```

This takes the card pointer, the device-level (`SNDRV_DEV_XXX`), the data pointer, and the callback pointers (`&ops`). The device-level defines the type of components and the order of registration and de-registration. For most components, the device-level is already defined. For a user-defined component, you can use `SNDRV_DEV_LOWLEVEL`.

This function itself doesn't allocate the data space. The data must be allocated manually beforehand, and its pointer is passed as the argument. This pointer (`chip` in the above example) is used as the identifier for the instance.

Each pre-defined ALSA component such as AC97 and PCM calls *snd_device_new()* inside its constructor. The destructor for each component is defined in the callback pointers. Hence, you don't need to take care of calling a destructor for such a component.

If you wish to create your own component, you need to set the destructor function to the dev_free callback in the `ops`, so that it can be released automatically via *snd_card_free()*. The next example will show an implementation of chip-specific data.

Chip-Specific Data

Chip-specific information, e.g. the I/O port address, its resource pointer, or the irq number, is stored in the chip-specific record:

```
struct mychip {
        ....
};
```

In general, there are two ways of allocating the chip record.

1. Allocating via `snd_card_new()`.

As mentioned above, you can pass the extra-data-length to the 5th argument of *snd_card_new()*, e.g.:

```
err = snd_card_new(&pci->dev, index[dev], id[dev], THIS_MODULE,
                   sizeof(struct mychip), &card);
```

struct mychip is the type of the chip record.

In return, the allocated record can be accessed as

```
struct mychip *chip = card->private_data;
```

With this method, you don't have to allocate twice. The record is released together with the card instance.

2. Allocating an extra device.

After allocating a card instance via *snd_card_new()* (with 0 on the 4th arg), call `kzalloc()`:

```
struct snd_card *card;
struct mychip *chip;
err = snd_card_new(&pci->dev, index[dev], id[dev], THIS_MODULE,
                   0, &card);
.....
chip = kzalloc(sizeof(*chip), GFP_KERNEL);
```

The chip record should have the field to hold the card pointer at least,

```
struct mychip {
        struct snd_card *card;
        ....
};
```

Then, set the card pointer in the returned chip instance:

```
chip->card = card;
```

Next, initialize the fields, and register this chip record as a low-level device with a specified ops:

```
static const struct snd_device_ops ops = {
        .dev_free =         snd_mychip_dev_free,
};
....
snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
```

snd_mychip_dev_free() is the device-destructor function, which will call the real destructor:

```
static int snd_mychip_dev_free(struct snd_device *device)
{
```

```
                return snd_mychip_free(device->device_data);
}
```

where `snd_mychip_free()` is the real destructor.

The demerit of this method is the obviously larger amount of code. The merit is, however, that you can trigger your own callback at registering and disconnecting the card via a setting in snd_device_ops. About registering and disconnecting the card, see the subsections below.

Registration and Release

After all components are assigned, register the card instance by calling *snd_card_register()*. Access to the device files is enabled at this point. That is, before *snd_card_register()* is called, the components are safely inaccessible from external side. If this call fails, exit the probe function after releasing the card via *snd_card_free()*.

For releasing the card instance, you can call simply *snd_card_free()*. As mentioned earlier, all components are released automatically by this call.

For a device which allows hotplugging, you can use *snd_card_free_when_closed()*. This one will postpone the destruction until all devices are closed.

1.2.5 PCI Resource Management

Full Code Example

In this section, we'll complete the chip-specific constructor, destructor and PCI entries. Example code is shown first, below:

```
struct mychip {
        struct snd_card *card;
        struct pci_dev *pci;

        unsigned long port;
        int irq;
};

static int snd_mychip_free(struct mychip *chip)
{
        /* disable hardware here if any */
        .... /* (not implemented in this document) */

        /* release the irq */
        if (chip->irq >= 0)
                free_irq(chip->irq, chip);
        /* release the I/O ports & memory */
        pci_release_regions(chip->pci);
        /* disable the PCI entry */
        pci_disable_device(chip->pci);
        /* release the data */
        kfree(chip);
```

```
                return 0;
}

/* chip-specific constructor */
static int snd_mychip_create(struct snd_card *card,
                             struct pci_dev *pci,
                             struct mychip **rchip)
{
        struct mychip *chip;
        int err;
        static const struct snd_device_ops ops = {
                .dev_free = snd_mychip_dev_free,
        };

        *rchip = NULL;

        /* initialize the PCI entry */
        err = pci_enable_device(pci);
        if (err < 0)
                return err;
        /* check PCI availability (28bit DMA) */
        if (pci_set_dma_mask(pci, DMA_BIT_MASK(28)) < 0 ||
            pci_set_consistent_dma_mask(pci, DMA_BIT_MASK(28)) < 0) {
                printk(KERN_ERR "error to set 28bit mask DMA\n");
                pci_disable_device(pci);
                return -ENXIO;
        }

        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        if (chip == NULL) {
                pci_disable_device(pci);
                return -ENOMEM;
        }

        /* initialize the stuff */
        chip->card = card;
        chip->pci = pci;
        chip->irq = -1;

        /* (1) PCI resource allocation */
        err = pci_request_regions(pci, "My Chip");
        if (err < 0) {
                kfree(chip);
                pci_disable_device(pci);
                return err;
        }
        chip->port = pci_resource_start(pci, 0);
        if (request_irq(pci->irq, snd_mychip_interrupt,
                        IRQF_SHARED, KBUILD_MODNAME, chip)) {
                printk(KERN_ERR "cannot grab irq %d\n", pci->irq);
```

```
                snd_mychip_free(chip);
                return -EBUSY;
        }
        chip->irq = pci->irq;
        card->sync_irq = chip->irq;

        /* (2) initialization of the chip hardware */
        .... /*   (not implemented in this document) */

        err = snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
        if (err < 0) {
                snd_mychip_free(chip);
                return err;
        }

        *rchip = chip;
        return 0;
}

/* PCI IDs */
static struct pci_device_id snd_mychip_ids[] = {
        { PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
          PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0, },
        ....
        { 0, }
};
MODULE_DEVICE_TABLE(pci, snd_mychip_ids);

/* pci_driver definition */
static struct pci_driver driver = {
        .name = KBUILD_MODNAME,
        .id_table = snd_mychip_ids,
        .probe = snd_mychip_probe,
        .remove = snd_mychip_remove,
};

/* module initialization */
static int __init alsa_card_mychip_init(void)
{
        return pci_register_driver(&driver);
}

/* module clean up */
static void __exit alsa_card_mychip_exit(void)
{
        pci_unregister_driver(&driver);
}

module_init(alsa_card_mychip_init)
module_exit(alsa_card_mychip_exit)
```

```
EXPORT_NO_SYMBOLS; /* for old kernels only */
```

Some Hafta's

The allocation of PCI resources is done in the `probe` function, and usually an extra `xxx_create()` function is written for this purpose.

In the case of PCI devices, you first have to call the `pci_enable_device()` function before allocating resources. Also, you need to set the proper PCI DMA mask to limit the accessed I/O range. In some cases, you might need to call `pci_set_master()` function, too.

Suppose a 28bit mask, the code to be added would look like:

```
err = pci_enable_device(pci);
if (err < 0)
        return err;
if (pci_set_dma_mask(pci, DMA_BIT_MASK(28)) < 0 ||
    pci_set_consistent_dma_mask(pci, DMA_BIT_MASK(28)) < 0) {
        printk(KERN_ERR "error to set 28bit mask DMA\n");
        pci_disable_device(pci);
        return -ENXIO;
}
```

Resource Allocation

The allocation of I/O ports and irqs is done via standard kernel functions. These resources must be released in the destructor function (see below).

Now assume that the PCI device has an I/O port with 8 bytes and an interrupt. Then struct mychip will have the following fields:

```
struct mychip {
        struct snd_card *card;

        unsigned long port;
        int irq;
};
```

For an I/O port (and also a memory region), you need to have the resource pointer for the standard resource management. For an irq, you have to keep only the irq number (integer). But you need to initialize this number to -1 before actual allocation, since irq 0 is valid. The port address and its resource pointer can be initialized as null by `kzalloc()` automatically, so you don't have to take care of resetting them.

The allocation of an I/O port is done like this:

```
err = pci_request_regions(pci, "My Chip");
if (err < 0) {
        kfree(chip);
        pci_disable_device(pci);
```

```
        return err;
}
chip->port = pci_resource_start(pci, 0);
```

It will reserve the I/O port region of 8 bytes of the given PCI device. The returned value, `chip->res_port`, is allocated via kmalloc() by `request_region()`. The pointer must be released via `kfree()`, but there is a problem with this. This issue will be explained later.

The allocation of an interrupt source is done like this:

```
if (request_irq(pci->irq, snd_mychip_interrupt,
                IRQF_SHARED, KBUILD_MODNAME, chip)) {
        printk(KERN_ERR "cannot grab irq %d\n", pci->irq);
        snd_mychip_free(chip);
        return -EBUSY;
}
chip->irq = pci->irq;
```

where `snd_mychip_interrupt()` is the interrupt handler defined *later*. Note that `chip->irq` should be defined only when `request_irq()` succeeded.

On the PCI bus, interrupts can be shared. Thus, `IRQF_SHARED` is used as the interrupt flag of `request_irq()`.

The last argument of `request_irq()` is the data pointer passed to the interrupt handler. Usually, the chip-specific record is used for that, but you can use what you like, too.

I won't give details about the interrupt handler at this point, but at least its appearance can be explained now. The interrupt handler looks usually as follows:

```
static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
        struct mychip *chip = dev_id;
        ....
        return IRQ_HANDLED;
}
```

After requesting the IRQ, you can passed it to `card->sync_irq` field:

```
card->irq = chip->irq;
```

This allows the PCM core to automatically call `synchronize_irq()` at the right time, like before hw_free. See the later section *sync_stop callback* for details.

Now let's write the corresponding destructor for the resources above. The role of destructor is simple: disable the hardware (if already activated) and release the resources. So far, we have no hardware part, so the disabling code is not written here.

To release the resources, the "check-and-release" method is a safer way. For the interrupt, do like this:

```
if (chip->irq >= 0)
        free_irq(chip->irq, chip);
```

Since the irq number can start from 0, you should initialize `chip->irq` with a negative value (e.g. -1), so that you can check the validity of the irq number as above.

When you requested I/O ports or memory regions via `pci_request_region()` or `pci_request_regions()` like in this example, release the resource(s) using the corresponding function, `pci_release_region()` or `pci_release_regions()`:

```
pci_release_regions(chip->pci);
```

When you requested manually via `request_region()` or `request_mem_region()`, you can release it via `release_resource()`. Suppose that you keep the resource pointer returned from `request_region()` in chip->res_port, the release procedure looks like:

```
release_and_free_resource(chip->res_port);
```

Don't forget to call `pci_disable_device()` before the end.

And finally, release the chip-specific record:

```
kfree(chip);
```

We didn't implement the hardware disabling part above. If you need to do this, please note that the destructor may be called even before the initialization of the chip is completed. It would be better to have a flag to skip hardware disabling if the hardware was not initialized yet.

When the chip-data is assigned to the card using *snd_device_new()* with `SNDRV_DEV_LOWLELVEL`, its destructor is called last. That is, it is assured that all other components like PCMs and controls have already been released. You don't have to stop PCMs, etc. explicitly, but just call low-level hardware stopping.

The management of a memory-mapped region is almost as same as the management of an I/O port. You'll need two fields as follows:

```
struct mychip {
        ....
        unsigned long iobase_phys;
        void __iomem *iobase_virt;
};
```

and the allocation would look like below:

```
err = pci_request_regions(pci, "My Chip");
if (err < 0) {
        kfree(chip);
        return err;
}
chip->iobase_phys = pci_resource_start(pci, 0);
chip->iobase_virt = ioremap(chip->iobase_phys,
                                    pci_resource_len(pci, 0));
```

and the corresponding destructor would be:

```
static int snd_mychip_free(struct mychip *chip)
{
```

```
        ....
        if (chip->iobase_virt)
                iounmap(chip->iobase_virt);
        ....
        pci_release_regions(chip->pci);
        ....
}
```

Of course, a modern way with `pci_iomap()` will make things a bit easier, too:

```
err = pci_request_regions(pci, "My Chip");
if (err < 0) {
        kfree(chip);
        return err;
}
chip->iobase_virt = pci_iomap(pci, 0, 0);
```

which is paired with `pci_iounmap()` at destructor.

PCI Entries

So far, so good. Let's finish the missing PCI stuff. At first, we need a struct pci_device_id table for this chipset. It's a table of PCI vendor/device ID number, and some masks.

For example:

```
static struct pci_device_id snd_mychip_ids[] = {
        { PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
          PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0, },
        ....
        { 0, }
};
MODULE_DEVICE_TABLE(pci, snd_mychip_ids);
```

The first and second fields of the struct pci_device_id are the vendor and device IDs. If you have no reason to filter the matching devices, you can leave the remaining fields as above. The last field of the struct pci_device_id contains private data for this entry. You can specify any value here, for example, to define specific operations for supported device IDs. Such an example is found in the intel8x0 driver.

The last entry of this list is the terminator. You must specify this all-zero entry.

Then, prepare the struct pci_driver record:

```
static struct pci_driver driver = {
        .name = KBUILD_MODNAME,
        .id_table = snd_mychip_ids,
        .probe = snd_mychip_probe,
        .remove = snd_mychip_remove,
};
```

The `probe` and `remove` functions have already been defined in the previous sections. The `name` field is the name string of this device. Note that you must not use slashes ("/") in this string.

And at last, the module entries:

```
static int __init alsa_card_mychip_init(void)
{
        return pci_register_driver(&driver);
}

static void __exit alsa_card_mychip_exit(void)
{
        pci_unregister_driver(&driver);
}

module_init(alsa_card_mychip_init)
module_exit(alsa_card_mychip_exit)
```

Note that these module entries are tagged with __init and __exit prefixes.

That's all!

1.2.6 PCM Interface

General

The PCM middle layer of ALSA is quite powerful and it is only necessary for each driver to implement the low-level functions to access its hardware.

To access the PCM layer, you need to include <sound/pcm.h> first. In addition, <sound/pcm_params.h> might be needed if you access some functions related with hw_param.

Each card device can have up to four PCM instances. A PCM instance corresponds to a PCM device file. The limitation of number of instances comes only from the available bit size of Linux' device numbers. Once 64bit device numbers are used, we'll have more PCM instances available.

A PCM instance consists of PCM playback and capture streams, and each PCM stream consists of one or more PCM substreams. Some soundcards support multiple playback functions. For example, emu10k1 has a PCM playback of 32 stereo substreams. In this case, at each open, a free substream is (usually) automatically chosen and opened. Meanwhile, when only one substream exists and it was already opened, a subsequent open will either block or error with `EAGAIN` according to the file open mode. But you don't have to care about such details in your driver. The PCM middle layer will take care of such work.

Full Code Example

The example code below does not include any hardware access routines but shows only the skeleton, how to build up the PCM interfaces:

```
#include <sound/pcm.h>
....

/* hardware definition */
static struct snd_pcm_hardware snd_mychip_playback_hw = {
        .info = (SNDRV_PCM_INFO_MMAP |
                 SNDRV_PCM_INFO_INTERLEAVED |
                 SNDRV_PCM_INFO_BLOCK_TRANSFER |
                 SNDRV_PCM_INFO_MMAP_VALID),
        .formats =          SNDRV_PCM_FMTBIT_S16_LE,
        .rates =            SNDRV_PCM_RATE_8000_48000,
        .rate_min =         8000,
        .rate_max =         48000,
        .channels_min =     2,
        .channels_max =     2,
        .buffer_bytes_max = 32768,
        .period_bytes_min = 4096,
        .period_bytes_max = 32768,
        .periods_min =      1,
        .periods_max =      1024,
};

/* hardware definition */
static struct snd_pcm_hardware snd_mychip_capture_hw = {
        .info = (SNDRV_PCM_INFO_MMAP |
                 SNDRV_PCM_INFO_INTERLEAVED |
                 SNDRV_PCM_INFO_BLOCK_TRANSFER |
                 SNDRV_PCM_INFO_MMAP_VALID),
        .formats =          SNDRV_PCM_FMTBIT_S16_LE,
        .rates =            SNDRV_PCM_RATE_8000_48000,
        .rate_min =         8000,
        .rate_max =         48000,
        .channels_min =     2,
        .channels_max =     2,
        .buffer_bytes_max = 32768,
        .period_bytes_min = 4096,
        .period_bytes_max = 32768,
        .periods_min =      1,
        .periods_max =      1024,
};

/* open callback */
static int snd_mychip_playback_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;
```

```
        runtime->hw = snd_mychip_playback_hw;
        /* more hardware-initialization will be done here */
        ....
        return 0;
}

/* close callback */
static int snd_mychip_playback_close(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        /* the hardware-specific codes will be here */
        ....
        return 0;

}

/* open callback */
static int snd_mychip_capture_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        runtime->hw = snd_mychip_capture_hw;
        /* more hardware-initialization will be done here */
        ....
        return 0;
}

/* close callback */
static int snd_mychip_capture_close(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        /* the hardware-specific codes will be here */
        ....
        return 0;
}

/* hw_params callback */
static int snd_mychip_pcm_hw_params(struct snd_pcm_substream *substream,
                          struct snd_pcm_hw_params *hw_params)
{
        /* the hardware-specific codes will be here */
        ....
        return 0;
}

/* hw_free callback */
static int snd_mychip_pcm_hw_free(struct snd_pcm_substream *substream)
{
```

```
        /* the hardware-specific codes will be here */
        ....
        return 0;
}

/* prepare callback */
static int snd_mychip_pcm_prepare(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        /* set up the hardware with the current configuration
         * for example...
         */
        mychip_set_sample_format(chip, runtime->format);
        mychip_set_sample_rate(chip, runtime->rate);
        mychip_set_channels(chip, runtime->channels);
        mychip_set_dma_setup(chip, runtime->dma_addr,
                               chip->buffer_size,
                               chip->period_size);
        return 0;
}

/* trigger callback */
static int snd_mychip_pcm_trigger(struct snd_pcm_substream *substream,
                                  int cmd)
{
        switch (cmd) {
        case SNDRV_PCM_TRIGGER_START:
                /* do something to start the PCM engine */
                ....
                break;
        case SNDRV_PCM_TRIGGER_STOP:
                /* do something to stop the PCM engine */
                ....
                break;
        default:
                return -EINVAL;
        }
}

/* pointer callback */
static snd_pcm_uframes_t
snd_mychip_pcm_pointer(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        unsigned int current_ptr;

        /* get the current hardware pointer */
        current_ptr = mychip_get_hw_pointer(chip);
```

```
        return current_ptr;
}

/* operators */
static struct snd_pcm_ops snd_mychip_playback_ops = {
        .open =         snd_mychip_playback_open,
        .close =        snd_mychip_playback_close,
        .hw_params =    snd_mychip_pcm_hw_params,
        .hw_free =      snd_mychip_pcm_hw_free,
        .prepare =      snd_mychip_pcm_prepare,
        .trigger =      snd_mychip_pcm_trigger,
        .pointer =      snd_mychip_pcm_pointer,
};

/* operators */
static struct snd_pcm_ops snd_mychip_capture_ops = {
        .open =         snd_mychip_capture_open,
        .close =        snd_mychip_capture_close,
        .hw_params =    snd_mychip_pcm_hw_params,
        .hw_free =      snd_mychip_pcm_hw_free,
        .prepare =      snd_mychip_pcm_prepare,
        .trigger =      snd_mychip_pcm_trigger,
        .pointer =      snd_mychip_pcm_pointer,
};

/*
 *  definitions of capture are omitted here...
 */

/* create a pcm device */
static int snd_mychip_new_pcm(struct mychip *chip)
{
        struct snd_pcm *pcm;
        int err;

        err = snd_pcm_new(chip->card, "My Chip", 0, 1, 1, &pcm);
        if (err < 0)
                return err;
        pcm->private_data = chip;
        strcpy(pcm->name, "My Chip");
        chip->pcm = pcm;
        /* set operators */
        snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                        &snd_mychip_playback_ops);
        snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE,
                        &snd_mychip_capture_ops);
        /* pre-allocation of buffers */
        /* NOTE: this may fail */
        snd_pcm_set_managed_buffer_all(pcm, SNDRV_DMA_TYPE_DEV,
                                        &chip->pci->dev,
```

```
                                              64*1024, 64*1024);
        return 0;
}
```

PCM Constructor

A PCM instance is allocated by the *snd_pcm_new()* function. It would be better to create a constructor for the PCM, namely:

```
static int snd_mychip_new_pcm(struct mychip *chip)
{
        struct snd_pcm *pcm;
        int err;

        err = snd_pcm_new(chip->card, "My Chip", 0, 1, 1, &pcm);
        if (err < 0)
                return err;
        pcm->private_data = chip;
        strcpy(pcm->name, "My Chip");
        chip->pcm = pcm;
        ...
        return 0;
}
```

The *snd_pcm_new()* function takes six arguments. The first argument is the card pointer to which this PCM is assigned, and the second is the ID string.

The third argument (index, 0 in the above) is the index of this new PCM. It begins from zero. If you create more than one PCM instances, specify the different numbers in this argument. For example, index = 1 for the second PCM device.

The fourth and fifth arguments are the number of substreams for playback and capture, respectively. Here 1 is used for both arguments. When no playback or capture substreams are available, pass 0 to the corresponding argument.

If a chip supports multiple playbacks or captures, you can specify more numbers, but they must be handled properly in open/close, etc. callbacks. When you need to know which substream you are referring to, then it can be obtained from struct snd_pcm_substream data passed to each callback as follows:

```
struct snd_pcm_substream *substream;
int index = substream->number;
```

After the PCM is created, you need to set operators for each PCM stream:

```
snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                &snd_mychip_playback_ops);
snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE,
                &snd_mychip_capture_ops);
```

The operators are defined typically like this:

```
static struct snd_pcm_ops snd_mychip_playback_ops = {
        .open =         snd_mychip_pcm_open,
        .close =        snd_mychip_pcm_close,
        .hw_params =    snd_mychip_pcm_hw_params,
        .hw_free =      snd_mychip_pcm_hw_free,
        .prepare =      snd_mychip_pcm_prepare,
        .trigger =      snd_mychip_pcm_trigger,
        .pointer =      snd_mychip_pcm_pointer,
};
```

All the callbacks are described in the *Operators* subsection.

After setting the operators, you probably will want to pre-allocate the buffer and set up the managed allocation mode. For that, simply call the following:

```
snd_pcm_set_managed_buffer_all(pcm, SNDRV_DMA_TYPE_DEV,
                               &chip->pci->dev,
                               64*1024, 64*1024);
```

It will allocate a buffer up to 64kB by default. Buffer management details will be described in the later section *Buffer and Memory Management*.

Additionally, you can set some extra information for this PCM in `pcm->info_flags`. The available values are defined as SNDRV_PCM_INFO_XXX in <sound/asound.h>, which is used for the hardware definition (described later). When your soundchip supports only half-duplex, specify it like this:

```
pcm->info_flags = SNDRV_PCM_INFO_HALF_DUPLEX;
```

... And the Destructor?

The destructor for a PCM instance is not always necessary. Since the PCM device will be released by the middle layer code automatically, you don't have to call the destructor explicitly.

The destructor would be necessary if you created special records internally and needed to release them. In such a case, set the destructor function to `pcm->private_free`:

```
static void mychip_pcm_free(struct snd_pcm *pcm)
{
        struct mychip *chip = snd_pcm_chip(pcm);
        /* free your own data */
        kfree(chip->my_private_pcm_data);
        /* do what you like else */
        ....
}

static int snd_mychip_new_pcm(struct mychip *chip)
{
        struct snd_pcm *pcm;
        ....
        /* allocate your own data */
        chip->my_private_pcm_data = kmalloc(...);
```

```
        /* set the destructor */
        pcm->private_data = chip;
        pcm->private_free = mychip_pcm_free;
        ....
}
```

Runtime Pointer - The Chest of PCM Information

When the PCM substream is opened, a PCM runtime instance is allocated and assigned to the substream. This pointer is accessible via `substream->runtime`. This runtime pointer holds most information you need to control the PCM: a copy of hw_params and sw_params configurations, the buffer pointers, mmap records, spinlocks, etc.

The definition of runtime instance is found in `<sound/pcm.h>`. Here is the relevant part of this file:

```
struct _snd_pcm_runtime {
        /* -- Status -- */
        struct snd_pcm_substream *trigger_master;
        snd_timestamp_t trigger_tstamp;      /* trigger timestamp */
        int overrange;
        snd_pcm_uframes_t avail_max;
        snd_pcm_uframes_t hw_ptr_base;       /* Position at buffer restart */
        snd_pcm_uframes_t hw_ptr_interrupt; /* Position at interrupt time*/

        /* -- HW params -- */
        snd_pcm_access_t access;      /* access mode */
        snd_pcm_format_t format;      /* SNDRV_PCM_FORMAT_* */
        snd_pcm_subformat_t subformat;      /* subformat */
        unsigned int rate;            /* rate in Hz */
        unsigned int channels;              /* channels */
        snd_pcm_uframes_t period_size;      /* period size */
        unsigned int periods;         /* periods */
        snd_pcm_uframes_t buffer_size;      /* buffer size */
        unsigned int tick_time;             /* tick time */
        snd_pcm_uframes_t min_align;  /* Min alignment for the format */
        size_t byte_align;
        unsigned int frame_bits;
        unsigned int sample_bits;
        unsigned int info;
        unsigned int rate_num;
        unsigned int rate_den;

        /* -- SW params -- */
        struct timespec tstamp_mode;  /* mmap timestamp is updated */
        unsigned int period_step;
        unsigned int sleep_min;                   /* min ticks to sleep */
        snd_pcm_uframes_t start_threshold;
        /*
         * The following two thresholds alleviate playback buffer underruns; when
```

```
        * hw_avail drops below the threshold, the respective action is␣
↪triggered:
        */
       snd_pcm_uframes_t stop_threshold;     /* - stop playback */
       snd_pcm_uframes_t silence_threshold;  /* - pre-fill buffer with␣
↪silence */
       snd_pcm_uframes_t silence_size;        /* max size of silence pre-fill;␣
↪when >= boundary,
                                              * fill played area with silence␣
↪immediately */
       snd_pcm_uframes_t boundary;   /* pointers wrap point */

       /* internal data of auto-silencer */
       snd_pcm_uframes_t silence_start; /* starting pointer to silence area */
       snd_pcm_uframes_t silence_filled; /* size filled with silence */

       snd_pcm_sync_id_t sync;                /* hardware synchronization ID */

       /* -- mmap -- */
       volatile struct snd_pcm_mmap_status *status;
       volatile struct snd_pcm_mmap_control *control;
       atomic_t mmap_count;

       /* -- locking / scheduling -- */
       spinlock_t lock;
       wait_queue_head_t sleep;
       struct timer_list tick_timer;
       struct fasync_struct *fasync;

       /* -- private section -- */
       void *private_data;
       void (*private_free)(struct snd_pcm_runtime *runtime);

       /* -- hardware description -- */
       struct snd_pcm_hardware hw;
       struct snd_pcm_hw_constraints hw_constraints;

       /* -- timer -- */
       unsigned int timer_resolution;        /* timer resolution */

       /* -- DMA -- */
       unsigned char *dma_area;      /* DMA area */
       dma_addr_t dma_addr;          /* physical bus address (not accessible␣
↪from main CPU) */
       size_t dma_bytes;             /* size of DMA area */

       struct snd_dma_buffer *dma_buffer_p;  /* allocated buffer */

#if defined(CONFIG_SND_PCM_OSS) || defined(CONFIG_SND_PCM_OSS_MODULE)
       /* -- OSS things -- */
```

```
        struct snd_pcm_oss_runtime oss;
#endif
};
```

For the operators (callbacks) of each sound driver, most of these records are supposed to be read-only. Only the PCM middle-layer changes / updates them. The exceptions are the hardware description (hw) DMA buffer information and the private data. Besides, if you use the standard managed buffer allocation mode, you don't need to set the DMA buffer information by yourself.

In the sections below, important records are explained.

Hardware Description

The hardware descriptor (struct snd_pcm_hardware) contains the definitions of the fundamental hardware configuration. Above all, you'll need to define this in the *PCM open callback*. Note that the runtime instance holds a copy of the descriptor, not a pointer to the existing descriptor. That is, in the open callback, you can modify the copied descriptor (`runtime->hw`) as you need. For example, if the maximum number of channels is 1 only on some chip models, you can still use the same hardware descriptor and change the channels_max later:

```
struct snd_pcm_runtime *runtime = substream->runtime;
...
runtime->hw = snd_mychip_playback_hw; /* common definition */
if (chip->model == VERY_OLD_ONE)
        runtime->hw.channels_max = 1;
```

Typically, you'll have a hardware descriptor as below:

```
static struct snd_pcm_hardware snd_mychip_playback_hw = {
        .info = (SNDRV_PCM_INFO_MMAP |
                SNDRV_PCM_INFO_INTERLEAVED |
                SNDRV_PCM_INFO_BLOCK_TRANSFER |
                SNDRV_PCM_INFO_MMAP_VALID),
        .formats =          SNDRV_PCM_FMTBIT_S16_LE,
        .rates =            SNDRV_PCM_RATE_8000_48000,
        .rate_min =         8000,
        .rate_max =         48000,
        .channels_min =     2,
        .channels_max =     2,
        .buffer_bytes_max = 32768,
        .period_bytes_min = 4096,
        .period_bytes_max = 32768,
        .periods_min =      1,
        .periods_max =      1024,
};
```

- The `info` field contains the type and capabilities of this PCM. The bit flags are defined in <sound/asound.h> as SNDRV_PCM_INFO_XXX. Here, at least, you have to specify whether mmap is supported and which interleaving formats are supported. When the hardware supports mmap, add the `SNDRV_PCM_INFO_MMAP` flag here. When the hardware supports the interleaved or the non-interleaved formats, the `SNDRV_PCM_INFO_INTERLEAVED`

or SNDRV_PCM_INFO_NONINTERLEAVED flag must be set, respectively. If both are supported, you can set both, too.

In the above example, MMAP_VALID and BLOCK_TRANSFER are specified for the OSS mmap mode. Usually both are set. Of course, MMAP_VALID is set only if mmap is really supported.

The other possible flags are SNDRV_PCM_INFO_PAUSE and SNDRV_PCM_INFO_RESUME. The PAUSE bit means that the PCM supports the "pause" operation, while the RESUME bit means that the PCM supports the full "suspend/resume" operation. If the PAUSE flag is set, the trigger callback below must handle the corresponding (pause push/release) commands. The suspend/resume trigger commands can be defined even without the RESUME flag. See the *Power Management* section for details.

When the PCM substreams can be synchronized (typically, synchronized start/stop of a playback and a capture stream), you can give SNDRV_PCM_INFO_SYNC_START, too. In this case, you'll need to check the linked-list of PCM substreams in the trigger callback. This will be described in a later section.

- The formats field contains the bit-flags of supported formats (SNDRV_PCM_FMTBIT_XXX). If the hardware supports more than one format, give all or'ed bits. In the example above, the signed 16bit little-endian format is specified.

- The rates field contains the bit-flags of supported rates (SNDRV_PCM_RATE_XXX). When the chip supports continuous rates, pass the CONTINUOUS bit additionally. The pre-defined rate bits are provided only for typical rates. If your chip supports unconventional rates, you need to add the KNOT bit and set up the hardware constraint manually (explained later).

- rate_min and rate_max define the minimum and maximum sample rate. This should correspond somehow to rates bits.

- channels_min and channels_max define, as you might have already expected, the minimum and maximum number of channels.

- buffer_bytes_max defines the maximum buffer size in bytes. There is no buffer_bytes_min field, since it can be calculated from the minimum period size and the minimum number of periods. Meanwhile, period_bytes_min and period_bytes_max define the minimum and maximum size of the period in bytes. periods_max and periods_min define the maximum and minimum number of periods in the buffer.

  The "period" is a term that corresponds to a fragment in the OSS world. The period defines the point at which a PCM interrupt is generated. This point strongly depends on the hardware. Generally, a smaller period size will give you more interrupts, which results in being able to fill/drain the buffer more timely. In the case of capture, this size defines the input latency. On the other hand, the whole buffer size defines the output latency for the playback direction.

- There is also a field fifo_size. This specifies the size of the hardware FIFO, but currently it is neither used by the drivers nor in the alsa-lib. So, you can ignore this field.

PCM Configurations

Ok, let's go back again to the PCM runtime records. The most frequently referred records in the runtime instance are the PCM configurations. The PCM configurations are stored in the runtime instance after the application sends hw_params data via alsa-lib. There are many fields copied from hw_params and sw_params structs. For example, `format` holds the format type chosen by the application. This field contains the enum value SNDRV_PCM_FORMAT_XXX.

One thing to be noted is that the configured buffer and period sizes are stored in "frames" in the runtime. In the ALSA world, `1 frame = channels \* samples-size`. For conversion between frames and bytes, you can use the *frames_to_bytes()* and *bytes_to_frames()* helper functions:

```
period_bytes = frames_to_bytes(runtime, runtime->period_size);
```

Also, many software parameters (sw_params) are stored in frames, too. Please check the type of the field. `snd_pcm_uframes_t` is for frames as unsigned integer while `snd_pcm_sframes_t` is for frames as signed integer.

DMA Buffer Information

The DMA buffer is defined by the following four fields: `dma_area`, `dma_addr`, `dma_bytes` and `dma_private`. `dma_area` holds the buffer pointer (the logical address). You can call `memcpy()` from/to this pointer. Meanwhile, `dma_addr` holds the physical address of the buffer. This field is specified only when the buffer is a linear buffer. `dma_bytes` holds the size of the buffer in bytes. `dma_private` is used for the ALSA DMA allocator.

If you use either the managed buffer allocation mode or the standard API function *snd_pcm_lib_malloc_pages()* for allocating the buffer, these fields are set by the ALSA middle layer, and you should *not* change them by yourself. You can read them but not write them. On the other hand, if you want to allocate the buffer by yourself, you'll need to manage it in the hw_params callback. At least, `dma_bytes` is mandatory. `dma_area` is necessary when the buffer is mmapped. If your driver doesn't support mmap, this field is not necessary. `dma_addr` is also optional. You can use dma_private as you like, too.

Running Status

The running status can be referred via `runtime->status`. This is a pointer to a struct snd_pcm_mmap_status record. For example, you can get the current DMA hardware pointer via `runtime->status->hw_ptr`.

The DMA application pointer can be referred via `runtime->control`, which points to a struct snd_pcm_mmap_control record. However, accessing this value directly is not recommended.

Private Data

You can allocate a record for the substream and store it in `runtime->private_data`. Usually, this is done in the *PCM open callback*. Don't mix this with `pcm->private_data`. The `pcm->private_data` usually points to the chip instance assigned statically at creation time of the PCM device, while `runtime->private_data` points to a dynamic data structure created in the PCM open callback:

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
        struct my_pcm_data *data;
        ....
        data = kmalloc(sizeof(*data), GFP_KERNEL);
        substream->runtime->private_data = data;
        ....
}
```

The allocated object must be released in the *close callback*.

Operators

OK, now let me give details about each PCM callback (`ops`). In general, every callback must return 0 if successful, or a negative error number such as `-EINVAL`. To choose an appropriate error number, it is advised to check what value other parts of the kernel return when the same kind of request fails.

Each callback function takes at least one argument containing a struct snd_pcm_substream pointer. To retrieve the chip record from the given substream instance, you can use the following macro:

```
int xxx(...) {
        struct mychip *chip = snd_pcm_substream_chip(substream);
        ....
}
```

The macro reads `substream->private_data`, which is a copy of `pcm->private_data`. You can override the former if you need to assign different data records per PCM substream. For example, the cmi8330 driver assigns different `private_data` for playback and capture directions, because it uses two different codecs (SB- and AD-compatible) for different directions.

PCM open callback

```
static int snd_xxx_open(struct snd_pcm_substream *substream);
```

This is called when a PCM substream is opened.

At least, here you have to initialize the `runtime->hw` record. Typically, this is done like this:

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
```

```
        struct snd_pcm_runtime *runtime = substream->runtime;

        runtime->hw = snd_mychip_playback_hw;
        return 0;
}
```

where `snd_mychip_playback_hw` is the pre-defined hardware description.

You can allocate private data in this callback, as described in the *Private Data* section.

If the hardware configuration needs more constraints, set the hardware constraints here, too. See *Constraints* for more details.

close callback

```
static int snd_xxx_close(struct snd_pcm_substream *substream);
```

Obviously, this is called when a PCM substream is closed.

Any private instance for a PCM substream allocated in the `open` callback will be released here:

```
static int snd_xxx_close(struct snd_pcm_substream *substream)
{
        ....
        kfree(substream->runtime->private_data);
        ....
}
```

ioctl callback

This is used for any special call to PCM ioctls. But usually you can leave it NULL, then the PCM core calls the generic ioctl callback function *snd_pcm_lib_ioctl()*. If you need to deal with a unique setup of channel info or reset procedure, you can pass your own callback function here.

hw_params callback

```
static int snd_xxx_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_hw_params *hw_params);
```

This is called when the hardware parameters (`hw_params`) are set up by the application, that is, once when the buffer size, the period size, the format, etc. are defined for the PCM substream.

Many hardware setups should be done in this callback, including the allocation of buffers.

Parameters to be initialized are retrieved by the `params_xxx()` macros.

When you choose managed buffer allocation mode for the substream, a buffer is already allocated before this callback gets called. Alternatively, you can call a helper function below for allocating the buffer:

```
snd_pcm_lib_malloc_pages(substream, params_buffer_bytes(hw_params));
```

snd_pcm_lib_malloc_pages() is available only when the DMA buffers have been pre-allocated. See the section *Buffer Types* for more details.

Note that this one and the `prepare` callback may be called multiple times per initialization. For example, the OSS emulation may call these callbacks at each change via its ioctl.

Thus, you need to be careful not to allocate the same buffers many times, which will lead to memory leaks! Calling the helper function above many times is OK. It will release the previous buffer automatically when it was already allocated.

Another note is that this callback is non-atomic (schedulable) by default, i.e. when no `nonatomic` flag set. This is important, because the `trigger` callback is atomic (non-schedulable). That is, mutexes or any schedule-related functions are not available in the `trigger` callback. Please see the subsection *Atomicity* for details.

hw_free callback

```
static int snd_xxx_hw_free(struct snd_pcm_substream *substream);
```

This is called to release the resources allocated via `hw_params`.

This function is always called before the close callback is called. Also, the callback may be called multiple times, too. Keep track whether each resource was already released.

When you have chosen managed buffer allocation mode for the PCM substream, the allocated PCM buffer will be automatically released after this callback gets called. Otherwise you'll have to release the buffer manually. Typically, when the buffer was allocated from the pre-allocated pool, you can use the standard API function *snd_pcm_lib_malloc_pages()* like:

```
snd_pcm_lib_free_pages(substream);
```

prepare callback

```
static int snd_xxx_prepare(struct snd_pcm_substream *substream);
```

This callback is called when the PCM is "prepared". You can set the format type, sample rate, etc. here. The difference from `hw_params` is that the `prepare` callback will be called each time *snd_pcm_prepare()* is called, i.e. when recovering after underruns, etc.

Note that this callback is non-atomic. You can use schedule-related functions safely in this callback.

In this and the following callbacks, you can refer to the values via the runtime record, `substream->runtime`. For example, to get the current rate, format or channels, access to `runtime->rate`, `runtime->format` or `runtime->channels`, respectively. The physical address of the allocated buffer is set to `runtime->dma_area`. The buffer and period sizes are in `runtime->buffer_size` and `runtime->period_size`, respectively.

Be careful that this callback will be called many times at each setup, too.

trigger callback

```
static int snd_xxx_trigger(struct snd_pcm_substream *substream, int cmd);
```

This is called when the PCM is started, stopped or paused.

The action is specified in the second argument, SNDRV_PCM_TRIGGER_XXX defined in <sound/pcm.h>. At least, the START and STOP commands must be defined in this callback:

```
switch (cmd) {
case SNDRV_PCM_TRIGGER_START:
        /* do something to start the PCM engine */
        break;
case SNDRV_PCM_TRIGGER_STOP:
        /* do something to stop the PCM engine */
        break;
default:
        return -EINVAL;
}
```

When the PCM supports the pause operation (given in the info field of the hardware table), the PAUSE_PUSH and PAUSE_RELEASE commands must be handled here, too. The former is the command to pause the PCM, and the latter to restart the PCM again.

When the PCM supports the suspend/resume operation, regardless of full or partial suspend/resume support, the SUSPEND and RESUME commands must be handled, too. These commands are issued when the power-management status is changed. Obviously, the SUSPEND and RESUME commands suspend and resume the PCM substream, and usually, they are identical to the STOP and START commands, respectively. See the *Power Management* section for details.

As mentioned, this callback is atomic by default unless the nonatomic flag set, and you cannot call functions which may sleep. The trigger callback should be as minimal as possible, just really triggering the DMA. The other stuff should be initialized in hw_params and prepare callbacks properly beforehand.

sync_stop callback

```
static int snd_xxx_sync_stop(struct snd_pcm_substream *substream);
```

This callback is optional, and NULL can be passed. It's called after the PCM core stops the stream, before it changes the stream state via prepare, hw_params or hw_free. Since the IRQ handler might be still pending, we need to wait until the pending task finishes before moving to the next step; otherwise it might lead to a crash due to resource conflicts or access to freed resources. A typical behavior is to call a synchronization function like synchronize_irq() here.

For the majority of drivers that need only a call of synchronize_irq(), there is a simpler setup, too. While keeping the sync_stop PCM callback NULL, the driver can set the card->sync_irq field to the returned interrupt number after requesting an IRQ, instead. Then PCM core will call synchronize_irq() with the given IRQ appropriately.

If the IRQ handler is released by the card destructor, you don't need to clear card->sync_irq, as the card itself is being released. So, usually you'll need to add just a single line for assigning

`card->sync_irq` in the driver code unless the driver re-acquires the IRQ. When the driver frees and re-acquires the IRQ dynamically (e.g. for suspend/resume), it needs to clear and re-set `card->sync_irq` again appropriately.

pointer callback

```
static snd_pcm_uframes_t snd_xxx_pointer(struct snd_pcm_substream *substream)
```

This callback is called when the PCM middle layer inquires the current hardware position in the buffer. The position must be returned in frames, ranging from 0 to `buffer_size - 1`.

This is usually called from the buffer-update routine in the PCM middle layer, which is invoked when *snd_pcm_period_elapsed()* is called by the interrupt routine. Then the PCM middle layer updates the position and calculates the available space, and wakes up the sleeping poll threads, etc.

This callback is also atomic by default.

copy and fill_silence ops

These callbacks are not mandatory, and can be omitted in most cases. These callbacks are used when the hardware buffer cannot be in the normal memory space. Some chips have their own buffer in the hardware which is not mappable. In such a case, you have to transfer the data manually from the memory buffer to the hardware buffer. Or, if the buffer is non-contiguous on both physical and virtual memory spaces, these callbacks must be defined, too.

If these two callbacks are defined, copy and set-silence operations are done by them. The details will be described in the later section *Buffer and Memory Management*.

ack callback

This callback is also not mandatory. This callback is called when the `appl_ptr` is updated in read or write operations. Some drivers like emu10k1-fx and cs46xx need to track the current `appl_ptr` for the internal buffer, and this callback is useful only for such a purpose.

The callback function may return 0 or a negative error. When the return value is `-EPIPE`, PCM core treats that as a buffer XRUN, and changes the state to `SNDRV_PCM_STATE_XRUN` automatically.

This callback is atomic by default.

page callback

This callback is optional too. The mmap calls this callback to get the page fault address.

You need no special callback for the standard SG-buffer or vmalloc- buffer. Hence this callback should be rarely used.

mmap callback

This is another optional callback for controlling mmap behavior. When defined, the PCM core calls this callback when a page is memory-mapped, instead of using the standard helper. If you need special handling (due to some architecture or device-specific issues), implement everything here as you like.

PCM Interrupt Handler

The remainder of the PCM stuff is the PCM interrupt handler. The role of the PCM interrupt handler in the sound driver is to update the buffer position and to tell the PCM middle layer when the buffer position goes across the specified period boundary. To inform about this, call the *snd_pcm_period_elapsed()* function.

There are several ways sound chips can generate interrupts.

Interrupts at the period (fragment) boundary

This is the most frequently found type: the hardware generates an interrupt at each period boundary. In this case, you can call *snd_pcm_period_elapsed()* at each interrupt.

snd_pcm_period_elapsed() takes the substream pointer as its argument. Thus, you need to keep the substream pointer accessible from the chip instance. For example, define `substream` field in the chip record to hold the current running substream pointer, and set the pointer value at `open` callback (and reset at `close` callback).

If you acquire a spinlock in the interrupt handler, and the lock is used in other PCM callbacks, too, then you have to release the lock before calling *snd_pcm_period_elapsed()*, because *snd_pcm_period_elapsed()* calls other PCM callbacks inside.

Typical code would look like:

```
static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
        struct mychip *chip = dev_id;
        spin_lock(&chip->lock);
        ....
        if (pcm_irq_invoked(chip)) {
                /* call updater, unlock before it */
                spin_unlock(&chip->lock);
                snd_pcm_period_elapsed(chip->substream);
                spin_lock(&chip->lock);
                /* acknowledge the interrupt if necessary */
        }
```

```
        ....
        spin_unlock(&chip->lock);
        return IRQ_HANDLED;
}
```

Also, when the device can detect a buffer underrun/overrun, the driver can notify the XRUN status to the PCM core by calling *snd_pcm_stop_xrun()*. This function stops the stream and sets the PCM state to SNDRV_PCM_STATE_XRUN. Note that it must be called outside the PCM stream lock, hence it can't be called from the atomic callback.

High frequency timer interrupts

This happens when the hardware doesn't generate interrupts at the period boundary but issues timer interrupts at a fixed timer rate (e.g. es1968 or ymfpci drivers). In this case, you need to check the current hardware position and accumulate the processed sample length at each interrupt. When the accumulated size exceeds the period size, call *snd_pcm_period_elapsed()* and reset the accumulator.

Typical code would look as follows:

```
static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
        struct mychip *chip = dev_id;
        spin_lock(&chip->lock);
        ....
        if (pcm_irq_invoked(chip)) {
                unsigned int last_ptr, size;
                /* get the current hardware pointer (in frames) */
                last_ptr = get_hw_ptr(chip);
                /* calculate the processed frames since the
                 * last update
                 */
                if (last_ptr < chip->last_ptr)
                        size = runtime->buffer_size + last_ptr
                                  - chip->last_ptr;
                else
                        size = last_ptr - chip->last_ptr;
                /* remember the last updated point */
                chip->last_ptr = last_ptr;
                /* accumulate the size */
                chip->size += size;
                /* over the period boundary? */
                if (chip->size >= runtime->period_size) {
                        /* reset the accumulator */
                        chip->size %= runtime->period_size;
                        /* call updater */
                        spin_unlock(&chip->lock);
                        snd_pcm_period_elapsed(substream);
                        spin_lock(&chip->lock);
                }
```

```
                /* acknowledge the interrupt if necessary */
        }
        ....
        spin_unlock(&chip->lock);
        return IRQ_HANDLED;
}
```

On calling `snd_pcm_period_elapsed()`

In both cases, even if more than one period has elapsed, you don't have to call *snd_pcm_period_elapsed()* many times. Call only once. And the PCM layer will check the current hardware pointer and update to the latest status.

Atomicity

One of the most important (and thus difficult to debug) problems in kernel programming are race conditions. In the Linux kernel, they are usually avoided via spin-locks, mutexes or semaphores. In general, if a race condition can happen in an interrupt handler, it has to be managed atomically, and you have to use a spinlock to protect the critical section. If the critical section is not in interrupt handler code and if taking a relatively long time to execute is acceptable, you should use mutexes or semaphores instead.

As already seen, some PCM callbacks are atomic and some are not. For example, the `hw_params` callback is non-atomic, while the `trigger` callback is atomic. This means, the latter is called already in a spinlock held by the PCM middle layer, the PCM stream lock. Please take this atomicity into account when you choose a locking scheme in the callbacks.

In the atomic callbacks, you cannot use functions which may call `schedule()` or go to `sleep()`. Semaphores and mutexes can sleep, and hence they cannot be used inside the atomic callbacks (e.g. `trigger` callback). To implement some delay in such a callback, please use `udelay()` or `mdelay()`.

All three atomic callbacks (trigger, pointer, and ack) are called with local interrupts disabled.

However, it is possible to request all PCM operations to be non-atomic. This assumes that all call sites are in non-atomic contexts. For example, the function *snd_pcm_period_elapsed()* is called typically from the interrupt handler. But, if you set up the driver to use a threaded interrupt handler, this call can be in non-atomic context, too. In such a case, you can set the `nonatomic` field of the struct snd_pcm object after creating it. When this flag is set, mutex and rwsem are used internally in the PCM core instead of spin and rwlocks, so that you can call all PCM functions safely in a non-atomic context.

Also, in some cases, you might need to call *snd_pcm_period_elapsed()* in the atomic context (e.g. the period gets elapsed during `ack` or other callback). There is a variant that can be called inside the PCM stream lock *snd_pcm_period_elapsed_under_stream_lock()* for that purpose, too.

Constraints

Due to physical limitations, hardware is not infinitely configurable. These limitations are expressed by setting constraints.

For example, in order to restrict the sample rates to some supported values, use *snd_pcm_hw_constraint_list()*. You need to call this function in the open callback:

```
static unsigned int rates[] =
        {4000, 10000, 22050, 44100};
static struct snd_pcm_hw_constraint_list constraints_rates = {
        .count = ARRAY_SIZE(rates),
        .list = rates,
        .mask = 0,
};

static int snd_mychip_pcm_open(struct snd_pcm_substream *substream)
{
        int err;
        ....
        err = snd_pcm_hw_constraint_list(substream->runtime, 0,
                                         SNDRV_PCM_HW_PARAM_RATE,
                                         &constraints_rates);
        if (err < 0)
                return err;
        ....
}
```

There are many different constraints. Look at `sound/pcm.h` for a complete list. You can even define your own constraint rules. For example, let's suppose my_chip can manage a substream of 1 channel if and only if the format is `S16_LE`, otherwise it supports any format specified in struct snd_pcm_hardware (or in any other constraint_list). You can build a rule like this:

```
static int hw_rule_channels_by_format(struct snd_pcm_hw_params *params,
                                      struct snd_pcm_hw_rule *rule)
{
        struct snd_interval *c = hw_param_interval(params,
                        SNDRV_PCM_HW_PARAM_CHANNELS);
        struct snd_mask *f = hw_param_mask(params, SNDRV_PCM_HW_PARAM_FORMAT);
        struct snd_interval ch;

        snd_interval_any(&ch);
        if (f->bits[0] == SNDRV_PCM_FMTBIT_S16_LE) {
                ch.min = ch.max = 1;
                ch.integer = 1;
                return snd_interval_refine(c, &ch);
        }
        return 0;
}
```

Then you need to call this function to add your rule:

```
snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_CHANNELS,
                    hw_rule_channels_by_format, NULL,
                    SNDRV_PCM_HW_PARAM_FORMAT, -1);
```

The rule function is called when an application sets the PCM format, and it refines the number of channels accordingly. But an application may set the number of channels before setting the format. Thus you also need to define the inverse rule:

```
static int hw_rule_format_by_channels(struct snd_pcm_hw_params *params,
                                      struct snd_pcm_hw_rule *rule)
{
        struct snd_interval *c = hw_param_interval(params,
                SNDRV_PCM_HW_PARAM_CHANNELS);
        struct snd_mask *f = hw_param_mask(params, SNDRV_PCM_HW_PARAM_FORMAT);
        struct snd_mask fmt;

        snd_mask_any(&fmt);     /* Init the struct */
        if (c->min < 2) {
                fmt.bits[0] &= SNDRV_PCM_FMTBIT_S16_LE;
                return snd_mask_refine(f, &fmt);
        }
        return 0;
}
```

... and in the open callback:

```
snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_FORMAT,
                    hw_rule_format_by_channels, NULL,
                    SNDRV_PCM_HW_PARAM_CHANNELS, -1);
```

One typical usage of the hw constraints is to align the buffer size with the period size. By default, ALSA PCM core doesn't enforce the buffer size to be aligned with the period size. For example, it'd be possible to have a combination like 256 period bytes with 999 buffer bytes.

Many device chips, however, require the buffer to be a multiple of periods. In such a case, call *snd_pcm_hw_constraint_integer()* for SNDRV_PCM_HW_PARAM_PERIODS:

```
snd_pcm_hw_constraint_integer(substream->runtime,
                              SNDRV_PCM_HW_PARAM_PERIODS);
```

This assures that the number of periods is integer, hence the buffer size is aligned with the period size.

The hw constraint is a very powerful mechanism to define the preferred PCM configuration, and there are relevant helpers. I won't give more details here, rather I would like to say, "Luke, use the source."

1.2.7 Control Interface

General

The control interface is used widely for many switches, sliders, etc. which are accessed from user-space. Its most important use is the mixer interface. In other words, since ALSA 0.9.x, all the mixer stuff is implemented on the control kernel API.

ALSA has a well-defined AC97 control module. If your chip supports only the AC97 and nothing else, you can skip this section.

The control API is defined in `<sound/control.h>`. Include this file if you want to add your own controls.

Definition of Controls

To create a new control, you need to define the following three callbacks: `info`, `get` and `put`. Then, define a struct snd_kcontrol_new record, such as:

```
static struct snd_kcontrol_new my_control = {
        .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
        .name = "PCM Playback Switch",
        .index = 0,
        .access = SNDRV_CTL_ELEM_ACCESS_READWRITE,
        .private_value = 0xffff,
        .info = my_control_info,
        .get = my_control_get,
        .put = my_control_put
};
```

The `iface` field specifies the control type, SNDRV_CTL_ELEM_IFACE_XXX, which is usually MIXER. Use CARD for global controls that are not logically part of the mixer. If the control is closely associated with some specific device on the sound card, use HWDEP, PCM, RAWMIDI, TIMER, or SEQUENCER, and specify the device number with the `device` and `subdevice` fields.

The `name` is the name identifier string. Since ALSA 0.9.x, the control name is very important, because its role is classified from its name. There are pre-defined standard control names. The details are described in the *Control Names* subsection.

The `index` field holds the index number of this control. If there are several different controls with the same name, they can be distinguished by the index number. This is the case when several codecs exist on the card. If the index is zero, you can omit the definition above.

The `access` field contains the access type of this control. Give the combination of bit masks, SNDRV_CTL_ELEM_ACCESS_XXX, there. The details will be explained in the *Access Flags* subsection.

The `private_value` field contains an arbitrary long integer value for this record. When using the generic `info`, `get` and `put` callbacks, you can pass a value through this field. If several small numbers are necessary, you can combine them in bitwise. Or, it's possible to store a pointer (casted to unsigned long) of some record in this field, too.

The `tlv` field can be used to provide metadata about the control; see the *Metadata* subsection.

The other three are *Control Callbacks*.

Control Names

There are some standards to define the control names. A control is usually defined from the three parts as "SOURCE DIRECTION FUNCTION".

The first, `SOURCE`, specifies the source of the control, and is a string such as "Master", "PCM", "CD" and "Line". There are many pre-defined sources.

The second, `DIRECTION`, is one of the following strings according to the direction of the control: "Playback", "Capture", "Bypass Playback" and "Bypass Capture". Or, it can be omitted, meaning both playback and capture directions.

The third, `FUNCTION`, is one of the following strings according to the function of the control: "Switch", "Volume" and "Route".

The example of control names are, thus, "Master Capture Switch" or "PCM Playback Volume".

There are some exceptions:

Global capture and playback

"Capture Source", "Capture Switch" and "Capture Volume" are used for the global capture (input) source, switch and volume. Similarly, "Playback Switch" and "Playback Volume" are used for the global output gain switch and volume.

Tone-controls

tone-control switch and volumes are specified like "Tone Control - XXX", e.g. "Tone Control - Switch", "Tone Control - Bass", "Tone Control - Center".

3D controls

3D-control switches and volumes are specified like "3D Control - XXX", e.g. "3D Control - Switch", "3D Control - Center", "3D Control - Space".

Mic boost

Mic-boost switch is set as "Mic Boost" or "Mic Boost (6dB)".

More precise information can be found in `Documentation/sound/designs/control-names.rst`.

Access Flags

The access flag is the bitmask which specifies the access type of the given control. The default access type is SNDRV_CTL_ELEM_ACCESS_READWRITE, which means both read and write are allowed to this control. When the access flag is omitted (i.e. = 0), it is considered as READWRITE access by default.

When the control is read-only, pass SNDRV_CTL_ELEM_ACCESS_READ instead. In this case, you don't have to define the put callback. Similarly, when the control is write-only (although it's a rare case), you can use the WRITE flag instead, and you don't need the get callback.

If the control value changes frequently (e.g. the VU meter), VOLATILE flag should be given. This means that the control may be changed without *Change notification*. Applications should poll such a control constantly.

When the control may be updated, but currently has no effect on anything, setting the INACTIVE flag may be appropriate. For example, PCM controls should be inactive while no PCM device is open.

There are LOCK and OWNER flags to change the write permissions.

Control Callbacks

info callback

The info callback is used to get detailed information on this control. This must store the values of the given struct snd_ctl_elem_info object. For example, for a boolean control with a single element:

```
static int snd_myctl_mono_info(struct snd_kcontrol *kcontrol,
                      struct snd_ctl_elem_info *uinfo)
{
        uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;
        uinfo->count = 1;
        uinfo->value.integer.min = 0;
        uinfo->value.integer.max = 1;
        return 0;
}
```

The type field specifies the type of the control. There are BOOLEAN, INTEGER, ENUMERATED, BYTES, IEC958 and INTEGER64. The count field specifies the number of elements in this control. For example, a stereo volume would have count = 2. The value field is a union, and the values stored depend on the type. The boolean and integer types are identical.

The enumerated type is a bit different from the others. You'll need to set the string for the selectec item index:

```
static int snd_myctl_enum_info(struct snd_kcontrol *kcontrol,
                      struct snd_ctl_elem_info *uinfo)
{
        static char *texts[4] = {
                "First", "Second", "Third", "Fourth"
        };
```

```
        uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;
        uinfo->count = 1;
        uinfo->value.enumerated.items = 4;
        if (uinfo->value.enumerated.item > 3)
                uinfo->value.enumerated.item = 3;
        strcpy(uinfo->value.enumerated.name,
                texts[uinfo->value.enumerated.item]);
        return 0;
}
```

The above callback can be simplified with a helper function, *snd_ctl_enum_info()*. The final code looks like below. (You can pass ARRAY_SIZE(texts) instead of 4 in the third argument; it's a matter of taste.)

```
static int snd_myctl_enum_info(struct snd_kcontrol *kcontrol,
                        struct snd_ctl_elem_info *uinfo)
{
        static char *texts[4] = {
                "First", "Second", "Third", "Fourth"
        };
        return snd_ctl_enum_info(uinfo, 1, 4, texts);
}
```

Some common info callbacks are available for your convenience: *snd_ctl_boolean_mono_info()* and *snd_ctl_boolean_stereo_info()*. Obviously, the former is an info callback for a mono channel boolean item, just like snd_myctl_mono_info() above, and the latter is for a stereo channel boolean item.

get callback

This callback is used to read the current value of the control, so it can be returned to user-space.

For example:

```
static int snd_myctl_get(struct snd_kcontrol *kcontrol,
                        struct snd_ctl_elem_value *ucontrol)
{
        struct mychip *chip = snd_kcontrol_chip(kcontrol);
        ucontrol->value.integer.value[0] = get_some_value(chip);
        return 0;
}
```

The value field depends on the type of control as well as on the info callback. For example, the sb driver uses this field to store the register offset, the bit-shift and the bit-mask. The private_value field is set as follows:

```
.private_value = reg | (shift << 16) | (mask << 24)
```

and is retrieved in callbacks like:

```
static int snd_sbmixer_get_single(struct snd_kcontrol *kcontrol,
                                  struct snd_ctl_elem_value *ucontrol)
{
        int reg = kcontrol->private_value & 0xff;
        int shift = (kcontrol->private_value >> 16) & 0xff;
        int mask = (kcontrol->private_value >> 24) & 0xff;
        ....
}
```

In the get callback, you have to fill all the elements if the control has more than one element, i.e. count > 1. In the example above, we filled only one element (value.integer.value[0]) since count = 1 is assumed.

put callback

This callback is used to write a value coming from user-space.

For example:

```
static int snd_myctl_put(struct snd_kcontrol *kcontrol,
                         struct snd_ctl_elem_value *ucontrol)
{
        struct mychip *chip = snd_kcontrol_chip(kcontrol);
        int changed = 0;
        if (chip->current_value !=
            ucontrol->value.integer.value[0]) {
                change_current_value(chip,
                            ucontrol->value.integer.value[0]);
                changed = 1;
        }
        return changed;
}
```

As seen above, you have to return 1 if the value is changed. If the value is not changed, return 0 instead. If any fatal error happens, return a negative error code as usual.

As in the get callback, when the control has more than one element, all elements must be evaluated in this callback, too.

Callbacks are not atomic

All these three callbacks are not-atomic.

Control Constructor

When everything is ready, finally we can create a new control. To create a control, there are two functions to be called, *snd_ctl_new1()* and *snd_ctl_add()*.

In the simplest way, you can do it like this:

```
err = snd_ctl_add(card, snd_ctl_new1(&my_control, chip));
if (err < 0)
        return err;
```

where `my_control` is the struct snd_kcontrol_new object defined above, and chip is the object pointer to be passed to kcontrol->private_data which can be referred to in callbacks.

snd_ctl_new1() allocates a new struct snd_kcontrol instance, and *snd_ctl_add()* assigns the given control component to the card.

Change Notification

If you need to change and update a control in the interrupt routine, you can call *snd_ctl_notify()*. For example:

```
snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_VALUE, id_pointer);
```

This function takes the card pointer, the event-mask, and the control id pointer for the notification. The event-mask specifies the types of notification, for example, in the above example, the change of control values is notified. The id pointer is the pointer of struct snd_ctl_elem_id to be notified. You can find some examples in `es1938.c` or `es1968.c` for hardware volume interrupts.

Metadata

To provide information about the dB values of a mixer control, use one of the `DECLARE_TLV_xxx` macros from `<sound/tlv.h>` to define a variable containing this information, set the `tlv.p` field to point to this variable, and include the `SNDRV_CTL_ELEM_ACCESS_TLV_READ` flag in the `access` field; like this:

```
static DECLARE_TLV_DB_SCALE(db_scale_my_control, -4050, 150, 0);

static struct snd_kcontrol_new my_control = {
        ...
        .access = SNDRV_CTL_ELEM_ACCESS_READWRITE |
                  SNDRV_CTL_ELEM_ACCESS_TLV_READ,
        ...
        .tlv.p = db_scale_my_control,
};
```

The DECLARE_TLV_DB_SCALE() macro defines information about a mixer control where each step in the control's value changes the dB value by a constant dB amount. The first parameter is the name of the variable to be defined. The second parameter is the minimum value, in units of 0.01 dB. The third parameter is the step size, in units of 0.01 dB. Set the fourth parameter to 1 if the minimum value actually mutes the control.

The DECLARE_TLV_DB_LINEAR() macro defines information about a mixer control where the control's value affects the output linearly. The first parameter is the name of the variable to be defined. The second parameter is the minimum value, in units of 0.01 dB. The third parameter is the maximum value, in units of 0.01 dB. If the minimum value mutes the control, set the second parameter to TLV_DB_GAIN_MUTE.

1.2.8 API for AC97 Codec

General

The ALSA AC97 codec layer is a well-defined one, and you don't have to write much code to control it. Only low-level control routines are necessary. The AC97 codec API is defined in <sound/ac97_codec.h>.

Full Code Example

```
struct mychip {
        ....
        struct snd_ac97 *ac97;
        ....
};

static unsigned short snd_mychip_ac97_read(struct snd_ac97 *ac97,
                                           unsigned short reg)
{
        struct mychip *chip = ac97->private_data;
        ....
        /* read a register value here from the codec */
        return the_register_value;
}

static void snd_mychip_ac97_write(struct snd_ac97 *ac97,
                                  unsigned short reg, unsigned short val)
{
        struct mychip *chip = ac97->private_data;
        ....
        /* write the given register value to the codec */
}

static int snd_mychip_ac97(struct mychip *chip)
{
        struct snd_ac97_bus *bus;
        struct snd_ac97_template ac97;
        int err;
```

```
                static struct snd_ac97_bus_ops ops = {
                        .write = snd_mychip_ac97_write,
                        .read = snd_mychip_ac97_read,
                };

                err = snd_ac97_bus(chip->card, 0, &ops, NULL, &bus);
                if (err < 0)
                        return err;
                memset(&ac97, 0, sizeof(ac97));
                ac97.private_data = chip;
                return snd_ac97_mixer(bus, &ac97, &chip->ac97);
}
```

AC97 Constructor

To create an ac97 instance, first call *snd_ac97_bus()* with an `ac97_bus_ops_t` record with callback functions:

```
struct snd_ac97_bus *bus;
static struct snd_ac97_bus_ops ops = {
        .write = snd_mychip_ac97_write,
        .read = snd_mychip_ac97_read,
};

snd_ac97_bus(card, 0, &ops, NULL, &pbus);
```

The bus record is shared among all belonging ac97 instances.

And then call *snd_ac97_mixer()* with a struct snd_ac97_template record together with the bus pointer created above:

```
struct snd_ac97_template ac97;
int err;

memset(&ac97, 0, sizeof(ac97));
ac97.private_data = chip;
snd_ac97_mixer(bus, &ac97, &chip->ac97);
```

where chip->ac97 is a pointer to a newly created `ac97_t` instance. In this case, the chip pointer is set as the private data, so that the read/write callback functions can refer to this chip instance. This instance is not necessarily stored in the chip record. If you need to change the register values from the driver, or need the suspend/resume of ac97 codecs, keep this pointer to pass to the corresponding functions.

AC97 Callbacks

The standard callbacks are `read` and `write`. Obviously they correspond to the functions for read and write accesses to the hardware low-level codes.

The `read` callback returns the register value specified in the argument:

```
static unsigned short snd_mychip_ac97_read(struct snd_ac97 *ac97,
                                           unsigned short reg)
{
        struct mychip *chip = ac97->private_data;
        ....
        return the_register_value;
}
```

Here, the chip can be cast from `ac97->private_data`.

Meanwhile, the `write` callback is used to set the register value:

```
static void snd_mychip_ac97_write(struct snd_ac97 *ac97,
                    unsigned short reg, unsigned short val)
```

These callbacks are non-atomic like the control API callbacks.

There are also other callbacks: `reset`, `wait` and `init`.

The `reset` callback is used to reset the codec. If the chip requires a special kind of reset, you can define this callback.

The `wait` callback is used to add some waiting time in the standard initialization of the codec. If the chip requires the extra waiting time, define this callback.

The `init` callback is used for additional initialization of the codec.

Updating Registers in The Driver

If you need to access to the codec from the driver, you can call the following functions: *snd_ac97_write()*, *snd_ac97_read()*, *snd_ac97_update()* and *snd_ac97_update_bits()*.

Both *snd_ac97_write()* and *snd_ac97_update()* functions are used to set a value to the given register (AC97_XXX). The difference between them is that *snd_ac97_update()* doesn't write a value if the given value has been already set, while *snd_ac97_write()* always rewrites the value:

```
snd_ac97_write(ac97, AC97_MASTER, 0x8080);
snd_ac97_update(ac97, AC97_MASTER, 0x8080);
```

snd_ac97_read() is used to read the value of the given register. For example:

```
value = snd_ac97_read(ac97, AC97_MASTER);
```

snd_ac97_update_bits() is used to update some bits in the given register:

```
snd_ac97_update_bits(ac97, reg, mask, value);
```

Also, there is a function to change the sample rate (of a given register such as AC97_PCM_FRONT_DAC_RATE) when VRA or DRA is supported by the codec: *snd_ac97_set_rate()*:

```
snd_ac97_set_rate(ac97, AC97_PCM_FRONT_DAC_RATE, 44100);
```

The following registers are available to set the rate: AC97_PCM_MIC_ADC_RATE, AC97_PCM_FRONT_DAC_RATE, AC97_PCM_LR_ADC_RATE, AC97_SPDIF. When AC97_SPDIF is specified, the register is not really changed but the corresponding IEC958 status bits will be updated.

Clock Adjustment

In some chips, the clock of the codec isn't 48000 but using a PCI clock (to save a quartz!). In this case, change the field `bus->clock` to the corresponding value. For example, intel8x0 and es1968 drivers have their own function to read from the clock.

Proc Files

The ALSA AC97 interface will create a proc file such as `/proc/asound/card0/codec97#0/ac97#0-0` and `ac97#0-0+regs`. You can refer to these files to see the current status and registers of the codec.

Multiple Codecs

When there are several codecs on the same card, you need to call *snd_ac97_mixer()* multiple times with `ac97.num=1` or greater. The num field specifies the codec number.

If you set up multiple codecs, you either need to write different callbacks for each codec or check `ac97->num` in the callback routines.

1.2.9 MIDI (MPU401-UART) Interface

General

Many soundcards have built-in MIDI (MPU401-UART) interfaces. When the soundcard supports the standard MPU401-UART interface, most likely you can use the ALSA MPU401-UART API. The MPU401-UART API is defined in `<sound/mpu401.h>`.

Some soundchips have a similar but slightly different implementation of mpu401 stuff. For example, emu10k1 has its own mpu401 routines.

MIDI Constructor

To create a rawmidi object, call *snd_mpu401_uart_new()*:

```
struct snd_rawmidi *rmidi;
snd_mpu401_uart_new(card, 0, MPU401_HW_MPU401, port, info_flags,
                    irq, &rmidi);
```

The first argument is the card pointer, and the second is the index of this component. You can create up to 8 rawmidi devices.

The third argument is the type of the hardware, MPU401_HW_XXX. If it's not a special one, you can use MPU401_HW_MPU401.

The 4th argument is the I/O port address. Many backward-compatible MPU401 have an I/O port such as 0x330. Or, it might be a part of its own PCI I/O region. It depends on the chip design.

The 5th argument is a bitflag for additional information. When the I/O port address above is part of the PCI I/O region, the MPU401 I/O port might have been already allocated (reserved) by the driver itself. In such a case, pass a bit flag MPU401_INFO_INTEGRATED, and the mpu401-uart layer will allocate the I/O ports by itself.

When the controller supports only the input or output MIDI stream, pass the MPU401_INFO_INPUT or MPU401_INFO_OUTPUT bitflag, respectively. Then the rawmidi instance is created as a single stream.

MPU401_INFO_MMIO bitflag is used to change the access method to MMIO (via readb and writeb) instead of iob and outb. In this case, you have to pass the iomapped address to *snd_mpu401_uart_new()*.

When MPU401_INFO_TX_IRQ is set, the output stream isn't checked in the default interrupt handler. The driver needs to call *snd_mpu401_uart_interrupt_tx()* by itself to start processing the output stream in the irq handler.

If the MPU-401 interface shares its interrupt with the other logical devices on the card, set MPU401_INFO_IRQ_HOOK (see *below*).

Usually, the port address corresponds to the command port and port + 1 corresponds to the data port. If not, you may change the cport field of struct snd_mpu401 manually afterward. However, struct snd_mpu401 pointer is not returned explicitly by *snd_mpu401_uart_new()*. You need to cast rmidi->private_data to struct snd_mpu401 explicitly:

```
struct snd_mpu401 *mpu;
mpu = rmidi->private_data;
```

and reset the cport as you like:

```
mpu->cport = my_own_control_port;
```

The 6th argument specifies the ISA irq number that will be allocated. If no interrupt is to be allocated (because your code is already allocating a shared interrupt, or because the device does not use interrupts), pass -1 instead. For a MPU-401 device without an interrupt, a polling timer will be used instead.

MIDI Interrupt Handler

When the interrupt is allocated in *snd_mpu401_uart_new()*, an exclusive ISA interrupt handler is automatically used, hence you don't have anything else to do than creating the mpu401 stuff. Otherwise, you have to set MPU401_INFO_IRQ_HOOK, and call *snd_mpu401_uart_interrupt()* explicitly from your own interrupt handler when it has determined that a UART interrupt has occurred.

In this case, you need to pass the private_data of the returned rawmidi object from *snd_mpu401_uart_new()* as the second argument of *snd_mpu401_uart_interrupt()*:

```
snd_mpu401_uart_interrupt(irq, rmidi->private_data, regs);
```

1.2.10 RawMIDI Interface

Overview

The raw MIDI interface is used for hardware MIDI ports that can be accessed as a byte stream. It is not used for synthesizer chips that do not directly understand MIDI.

ALSA handles file and buffer management. All you have to do is to write some code to move data between the buffer and the hardware.

The rawmidi API is defined in <sound/rawmidi.h>.

RawMIDI Constructor

To create a rawmidi device, call the *snd_rawmidi_new()* function:

```
struct snd_rawmidi *rmidi;
err = snd_rawmidi_new(chip->card, "MyMIDI", 0, outs, ins, &rmidi);
if (err < 0)
        return err;
rmidi->private_data = chip;
strcpy(rmidi->name, "My MIDI");
rmidi->info_flags = SNDRV_RAWMIDI_INFO_OUTPUT |
                    SNDRV_RAWMIDI_INFO_INPUT |
                    SNDRV_RAWMIDI_INFO_DUPLEX;
```

The first argument is the card pointer, the second argument is the ID string.

The third argument is the index of this component. You can create up to 8 rawmidi devices.

The fourth and fifth arguments are the number of output and input substreams, respectively, of this device (a substream is the equivalent of a MIDI port).

Set the info_flags field to specify the capabilities of the device. Set SNDRV_RAWMIDI_INFO_OUTPUT if there is at least one output port, SNDRV_RAWMIDI_INFO_INPUT if there is at least one input port, and SNDRV_RAWMIDI_INFO_DUPLEX if the device can handle output and input at the same time.

After the rawmidi device is created, you need to set the operators (callbacks) for each substream. There are helper functions to set the operators for all the substreams of a device:

```
snd_rawmidi_set_ops(rmidi, SNDRV_RAWMIDI_STREAM_OUTPUT, &snd_mymidi_output_
↪ops);
snd_rawmidi_set_ops(rmidi, SNDRV_RAWMIDI_STREAM_INPUT, &snd_mymidi_input_ops);
```

The operators are usually defined like this:

```
static struct snd_rawmidi_ops snd_mymidi_output_ops = {
        .open =    snd_mymidi_output_open,
        .close =   snd_mymidi_output_close,
        .trigger = snd_mymidi_output_trigger,
};
```

These callbacks are explained in the *RawMIDI Callbacks* section.

If there are more than one substream, you should give a unique name to each of them:

```
struct snd_rawmidi_substream *substream;
list_for_each_entry(substream,
                    &rmidi->streams[SNDRV_RAWMIDI_STREAM_OUTPUT].substreams,
                    list {
        sprintf(substream->name, "My MIDI Port %d", substream->number + 1);
}
/* same for SNDRV_RAWMIDI_STREAM_INPUT */
```

RawMIDI Callbacks

In all the callbacks, the private data that you've set for the rawmidi device can be accessed as `substream->rmidi->private_data`.

If there is more than one port, your callbacks can determine the port index from the struct snd_rawmidi_substream data passed to each callback:

```
struct snd_rawmidi_substream *substream;
int index = substream->number;
```

RawMIDI open callback

```
static int snd_xxx_open(struct snd_rawmidi_substream *substream);
```

This is called when a substream is opened. You can initialize the hardware here, but you shouldn't start transmitting/receiving data yet.

RawMIDI close callback

```
static int snd_xxx_close(struct snd_rawmidi_substream *substream);
```

Guess what.

The `open` and `close` callbacks of a rawmidi device are serialized with a mutex, and can sleep.

Rawmidi trigger callback for output substreams

```
static void snd_xxx_output_trigger(struct snd_rawmidi_substream *substream,
                                   int up);
```

This is called with a nonzero `up` parameter when there is some data in the substream buffer that must be transmitted.

To read data from the buffer, call *snd_rawmidi_transmit_peek()*. It will return the number of bytes that have been read; this will be less than the number of bytes requested when there are no more data in the buffer. After the data have been transmitted successfully, call *snd_rawmidi_transmit_ack()* to remove the data from the substream buffer:

```
unsigned char data;
while (snd_rawmidi_transmit_peek(substream, &data, 1) == 1) {
        if (snd_mychip_try_to_transmit(data))
                snd_rawmidi_transmit_ack(substream, 1);
        else
                break; /* hardware FIFO full */
}
```

If you know beforehand that the hardware will accept data, you can use the *snd_rawmidi_transmit()* function which reads some data and removes them from the buffer at once:

```
while (snd_mychip_transmit_possible()) {
        unsigned char data;
        if (snd_rawmidi_transmit(substream, &data, 1) != 1)
                break; /* no more data */
        snd_mychip_transmit(data);
}
```

If you know beforehand how many bytes you can accept, you can use a buffer size greater than one with the `snd_rawmidi_transmit*()` functions.

The `trigger` callback must not sleep. If the hardware FIFO is full before the substream buffer has been emptied, you have to continue transmitting data later, either in an interrupt handler, or with a timer if the hardware doesn't have a MIDI transmit interrupt.

The `trigger` callback is called with a zero `up` parameter when the transmission of data should be aborted.

RawMIDI trigger callback for input substreams

```
static void snd_xxx_input_trigger(struct snd_rawmidi_substream *substream, int␣
↪up);
```

This is called with a nonzero up parameter to enable receiving data, or with a zero up parameter do disable receiving data.

The `trigger` callback must not sleep; the actual reading of data from the device is usually done in an interrupt handler.

When data reception is enabled, your interrupt handler should call *snd_rawmidi_receive()* for all received data:

```
void snd_mychip_midi_interrupt(...)
{
        while (mychip_midi_available()) {
                unsigned char data;
                data = mychip_midi_read();
                snd_rawmidi_receive(substream, &data, 1);
        }
}
```

drain callback

```
static void snd_xxx_drain(struct snd_rawmidi_substream *substream);
```

This is only used with output substreams. This function should wait until all data read from the substream buffer have been transmitted. This ensures that the device can be closed and the driver unloaded without losing data.

This callback is optional. If you do not set `drain` in the struct snd_rawmidi_ops structure, ALSA will simply wait for 50 milliseconds instead.

1.2.11 Miscellaneous Devices

FM OPL3

The FM OPL3 is still used in many chips (mainly for backward compatibility). ALSA has a nice OPL3 FM control layer, too. The OPL3 API is defined in `<sound/opl3.h>`.

FM registers can be directly accessed through the direct-FM API, defined in `<sound/asound_fm.h>`. In ALSA native mode, FM registers are accessed through the Hardware-Dependent Device direct-FM extension API, whereas in OSS compatible mode, FM registers can be accessed with the OSS direct-FM compatible API in `/dev/dmfmX` device.

To create the OPL3 component, you have two functions to call. The first one is a constructor for the `opl3_t` instance:

```
struct snd_opl3 *opl3;
snd_opl3_create(card, lport, rport, OPL3_HW_OPL3_XXX,
                integrated, &opl3);
```

The first argument is the card pointer, the second one is the left port address, and the third is the right port address. In most cases, the right port is placed at the left port + 2.

The fourth argument is the hardware type.

When the left and right ports have been already allocated by the card driver, pass non-zero to the fifth argument (`integrated`). Otherwise, the opl3 module will allocate the specified ports by itself.

When the accessing the hardware requires special method instead of the standard I/O access, you can create opl3 instance separately with `snd_opl3_new()`:

```
struct snd_opl3 *opl3;
snd_opl3_new(card, OPL3_HW_OPL3_XXX, &opl3);
```

Then set `command`, `private_data` and `private_free` for the private access function, the private data and the destructor. The `l_port` and `r_port` are not necessarily set. Only the command must be set properly. You can retrieve the data from the `opl3->private_data` field.

After creating the opl3 instance via `snd_opl3_new()`, call `snd_opl3_init()` to initialize the chip to the proper state. Note that `snd_opl3_create()` always calls it internally.

If the opl3 instance is created successfully, then create a hwdep device for this opl3:

```
struct snd_hwdep *opl3hwdep;
snd_opl3_hwdep_new(opl3, 0, 1, &opl3hwdep);
```

The first argument is the `opl3_t` instance you created, and the second is the index number, usually 0.

The third argument is the index-offset for the sequencer client assigned to the OPL3 port. When there is an MPU401-UART, give 1 for here (UART always takes 0).

Hardware-Dependent Devices

Some chips need user-space access for special controls or for loading the micro code. In such a case, you can create a hwdep (hardware-dependent) device. The hwdep API is defined in <sound/hwdep.h>. You can find examples in opl3 driver or `isa/sb/sb16_csp.c`.

The creation of the `hwdep` instance is done via *snd_hwdep_new()*:

```
struct snd_hwdep *hw;
snd_hwdep_new(card, "My HWDEP", 0, &hw);
```

where the third argument is the index number.

You can then pass any pointer value to the `private_data`. If you assign private data, you should define a destructor, too. The destructor function is set in the `private_free` field:

```
struct mydata *p = kmalloc(sizeof(*p), GFP_KERNEL);
hw->private_data = p;
hw->private_free = mydata_free;
```

and the implementation of the destructor would be:

```
static void mydata_free(struct snd_hwdep *hw)
{
        struct mydata *p = hw->private_data;
        kfree(p);
}
```

The arbitrary file operations can be defined for this instance. The file operators are defined in the `ops` table. For example, assume that this chip needs an ioctl:

```
hw->ops.open = mydata_open;
hw->ops.ioctl = mydata_ioctl;
hw->ops.release = mydata_release;
```

And implement the callback functions as you like.

IEC958 (S/PDIF)

Usually the controls for IEC958 devices are implemented via the control interface. There is a macro to compose a name string for IEC958 controls, `SNDRV_CTL_NAME_IEC958()` defined in `<include/asound.h>`.

There are some standard controls for IEC958 status bits. These controls use the type `SNDRV_CTL_ELEM_TYPE_IEC958`, and the size of element is fixed as 4 bytes array (value.iec958.status[x]). For the `info` callback, you don't specify the value field for this type (the count field must be set, though).

"IEC958 Playback Con Mask" is used to return the bit-mask for the IEC958 status bits of consumer mode. Similarly, "IEC958 Playback Pro Mask" returns the bitmask for professional mode. They are read-only controls.

Meanwhile, "IEC958 Playback Default" control is defined for getting and setting the current default IEC958 bits.

Due to historical reasons, both variants of the Playback Mask and the Playback Default controls can be implemented on either a `SNDRV_CTL_ELEM_IFACE_PCM` or a `SNDRV_CTL_ELEM_IFACE_MIXER` iface. Drivers should expose the mask and default on the same iface though.

In addition, you can define the control switches to enable/disable or to set the raw bit mode. The implementation will depend on the chip, but the control should be named as "IEC958 xxx", preferably using the `SNDRV_CTL_NAME_IEC958()` macro.

You can find several cases, for example, `pci/emu10k1`, `pci/ice1712`, or `pci/cmipci.c`.

1.2.12 Buffer and Memory Management

Buffer Types

ALSA provides several different buffer allocation functions depending on the bus and the architecture. All these have a consistent API. The allocation of physically-contiguous pages is done via the `snd_malloc_xxx_pages()` function, where xxx is the bus type.

The allocation of pages with fallback is done via *snd_dma_alloc_pages_fallback()*. This function tries to allocate the specified number of pages, but if not enough pages are available, it tries to reduce the request size until enough space is found, down to one page.

To release the pages, call the *snd_dma_free_pages()* function.

Usually, ALSA drivers try to allocate and reserve a large contiguous physical space at the time the module is loaded for later use. This is called "pre-allocation". As already written, you can call the following function at PCM instance construction time (in the case of PCI bus):

```
snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                      &pci->dev, size, max);
```

where `size` is the byte size to be pre-allocated and `max` is the maximum size settable via the `prealloc` proc file. The allocator will try to get an area as large as possible within the given size.

The second argument (type) and the third argument (device pointer) are dependent on the bus. For normal devices, pass the device pointer (typically identical as `card->dev`) to the third argument with `SNDRV_DMA_TYPE_DEV` type.

A continuous buffer unrelated to the bus can be pre-allocated with `SNDRV_DMA_TYPE_CONTINUOUS` type. You can pass NULL to the device pointer in that case, which is the default mode implying to allocate with the `GFP_KERNEL` flag. If you need a restricted (lower) address, set up the coherent DMA mask bits for the device, and pass the device pointer, like the normal device memory allocations. For this type, it's still allowed to pass NULL to the device pointer, too, if no address restriction is needed.

For the scatter-gather buffers, use `SNDRV_DMA_TYPE_DEV_SG` with the device pointer (see the *Non-Contiguous Buffers* section).

Once the buffer is pre-allocated, you can use the allocator in the `hw_params` callback:

```
snd_pcm_lib_malloc_pages(substream, size);
```

Note that you have to pre-allocate to use this function.

But most drivers use the "managed buffer allocation mode" instead of manual allocation and release. This is done by calling *snd_pcm_set_managed_buffer_all()* instead of *snd_pcm_lib_preallocate_pages_for_all()*:

```
snd_pcm_set_managed_buffer_all(pcm, SNDRV_DMA_TYPE_DEV,
                               &pci->dev, size, max);
```

where the passed arguments are identical for both functions. The difference in the managed mode is that PCM core will call *snd_pcm_lib_malloc_pages()* internally already before calling

the PCM `hw_params` callback, and call *snd_pcm_lib_free_pages()* after the PCM `hw_free` callback automatically. So the driver doesn't have to call these functions explicitly in its callback any longer. This allows many drivers to have NULL `hw_params` and `hw_free` entries.

External Hardware Buffers

Some chips have their own hardware buffers and DMA transfer from the host memory is not available. In such a case, you need to either 1) copy/set the audio data directly to the external hardware buffer, or 2) make an intermediate buffer and copy/set the data from it to the external hardware buffer in interrupts (or in tasklets, preferably).

The first case works fine if the external hardware buffer is large enough. This method doesn't need any extra buffers and thus is more efficient. You need to define the `copy` callback for the data transfer, in addition to the `fill_silence` callback for playback. However, there is a drawback: it cannot be mmapped. The examples are GUS's GF1 PCM or emu8000's wavetable PCM.

The second case allows for mmap on the buffer, although you have to handle an interrupt or a tasklet to transfer the data from the intermediate buffer to the hardware buffer. You can find an example in the vxpocket driver.

Another case is when the chip uses a PCI memory-map region for the buffer instead of the host memory. In this case, mmap is available only on certain architectures like the Intel one. In non-mmap mode, the data cannot be transferred as in the normal way. Thus you need to define the `copy` and `fill_silence` callbacks as well, as in the cases above. Examples are found in `rme32.c` and `rme96.c`.

The implementation of the `copy` and `silence` callbacks depends upon whether the hardware supports interleaved or non-interleaved samples. The `copy` callback is defined like below, a bit differently depending on whether the direction is playback or capture:

```
static int playback_copy(struct snd_pcm_substream *substream,
            int channel, unsigned long pos,
            struct iov_iter *src, unsigned long count);
static int capture_copy(struct snd_pcm_substream *substream,
            int channel, unsigned long pos,
            struct iov_iter *dst, unsigned long count);
```

In the case of interleaved samples, the second argument (`channel`) is not used. The third argument (`pos`) specifies the position in bytes.

The meaning of the fourth argument is different between playback and capture. For playback, it holds the source data pointer, and for capture, it's the destination data pointer.

The last argument is the number of bytes to be copied.

What you have to do in this callback is again different between playback and capture directions. In the playback case, you copy the given amount of data (`count`) at the specified pointer (`src`) to the specified offset (`pos`) in the hardware buffer. When coded like memcpy-like way, the copy would look like:

```
my_memcpy_from_iter(my_buffer + pos, src, count);
```

For the capture direction, you copy the given amount of data (`count`) at the specified offset (`pos`) in the hardware buffer to the specified pointer (`dst`):

```
my_memcpy_to_iter(dst, my_buffer + pos, count);
```

The given `src` or `dst` a struct iov_iter pointer containing the pointer and the size. Use the existing helpers to copy or access the data as defined in `linux/uio.h`.

Careful readers might notice that these callbacks receive the arguments in bytes, not in frames like other callbacks. It's because this makes coding easier like in the examples above, and also it makes it easier to unify both the interleaved and non-interleaved cases, as explained below.

In the case of non-interleaved samples, the implementation will be a bit more complicated. The callback is called for each channel, passed in the second argument, so in total it's called N times per transfer.

The meaning of the other arguments are almost the same as in the interleaved case. The callback is supposed to copy the data from/to the given user-space buffer, but only for the given channel. For details, please check `isa/gus/gus_pcm.c` or `pci/rme9652/rme9652.c` as examples.

Usually for the playback, another callback `fill_silence` is defined. It's implemented in a similar way as the copy callbacks above:

```
static int silence(struct snd_pcm_substream *substream, int channel,
                   unsigned long pos, unsigned long count);
```

The meanings of arguments are the same as in the `copy` callback, although there is no buffer pointer argument. In the case of interleaved samples, the channel argument has no meaning, as for the `copy` callback.

The role of the `fill_silence` callback is to set the given amount (`count`) of silence data at the specified offset (`pos`) in the hardware buffer. Suppose that the data format is signed (that is, the silent-data is 0), and the implementation using a memset-like function would look like:

```
my_memset(my_buffer + pos, 0, count);
```

In the case of non-interleaved samples, again, the implementation becomes a bit more complicated, as it's called N times per transfer for each channel. See, for example, `isa/gus/gus_pcm.c`.

Non-Contiguous Buffers

If your hardware supports a page table as in emu10k1 or buffer descriptors as in via82xx, you can use scatter-gather (SG) DMA. ALSA provides an interface for handling SG-buffers. The API is provided in <sound/pcm.h>.

For creating the SG-buffer handler, call *snd_pcm_set_managed_buffer()* or *snd_pcm_set_managed_buffer_all()* with SNDRV_DMA_TYPE_DEV_SG in the PCM constructor like for other PCI pre-allocations. You need to pass &pci->dev, where pci is the struct pci_dev pointer of the chip as well:

```
snd_pcm_set_managed_buffer_all(pcm, SNDRV_DMA_TYPE_DEV_SG,
                               &pci->dev, size, max);
```

The `struct snd_sg_buf` instance is created as `substream->dma_private` in turn. You can cast the pointer like:

```
struct snd_sg_buf *sgbuf = (struct snd_sg_buf *)substream->dma_private;
```

Then in the *snd_pcm_lib_malloc_pages()* call, the common SG-buffer handler will allocate the non-contiguous kernel pages of the given size and map them as virtually contiguous memory. The virtual pointer is addressed via runtime->dma_area. The physical address (runtime->dma_addr) is set to zero, because the buffer is physically non-contiguous. The physical address table is set up in sgbuf->table. You can get the physical address at a certain offset via *snd_pcm_sgbuf_get_addr()*.

If you need to release the SG-buffer data explicitly, call the standard API function *snd_pcm_lib_free_pages()* as usual.

Vmalloc'ed Buffers

It's possible to use a buffer allocated via vmalloc(), for example, for an intermediate buffer. You can simply allocate it via the standard *snd_pcm_lib_malloc_pages()* and co. after setting up the buffer preallocation with SNDRV_DMA_TYPE_VMALLOC type:

```
snd_pcm_set_managed_buffer_all(pcm, SNDRV_DMA_TYPE_VMALLOC,
                               NULL, 0, 0);
```

NULL is passed as the device pointer argument, which indicates that default pages (GFP_KERNEL and GFP_HIGHMEM) will be allocated.

Also, note that zero is passed as both the size and the max size argument here. Since each vmalloc call should succeed at any time, we don't need to pre-allocate the buffers like other continuous pages.

1.2.13 Proc Interface

ALSA provides an easy interface for procfs. The proc files are very useful for debugging. I recommend you set up proc files if you write a driver and want to get a running status or register dumps. The API is found in <sound/info.h>.

To create a proc file, call snd_card_proc_new():

```
struct snd_info_entry *entry;
int err = snd_card_proc_new(card, "my-file", &entry);
```

where the second argument specifies the name of the proc file to be created. The above example will create a file my-file under the card directory, e.g. /proc/asound/card0/my-file.

Like other components, the proc entry created via snd_card_proc_new() will be registered and released automatically in the card registration and release functions.

When the creation is successful, the function stores a new instance in the pointer given in the third argument. It is initialized as a text proc file for read only. To use this proc file as a read-only text file as-is, set the read callback with private data via snd_info_set_text_ops():

```
snd_info_set_text_ops(entry, chip, my_proc_read);
```

where the second argument (chip) is the private data to be used in the callback. The third parameter specifies the read buffer size and the fourth (my_proc_read) is the callback function, which is defined like:

```
static void my_proc_read(struct snd_info_entry *entry,
                         struct snd_info_buffer *buffer);
```

In the read callback, use snd_iprintf() for output strings, which works just like normal printf(). For example:

```
static void my_proc_read(struct snd_info_entry *entry,
                         struct snd_info_buffer *buffer)
{
        struct my_chip *chip = entry->private_data;

        snd_iprintf(buffer, "This is my chip!\n");
        snd_iprintf(buffer, "Port = %ld\n", chip->port);
}
```

The file permissions can be changed afterwards. By default, they are read only for all users. If you want to add write permission for the user (root by default), do as follows:

```
entry->mode = S_IFREG | S_IRUGO | S_IWUSR;
```

and set the write buffer size and the callback:

```
entry->c.text.write = my_proc_write;
```

In the write callback, you can use *snd_info_get_line()* to get a text line, and *snd_info_get_str()* to retrieve a string from the line. Some examples are found in core/oss/mixer_oss.c, core/oss/and pcm_oss.c.

For a raw-data proc-file, set the attributes as follows:

```
static const struct snd_info_entry_ops my_file_io_ops = {
        .read = my_file_io_read,
};

entry->content = SNDRV_INFO_CONTENT_DATA;
entry->private_data = chip;
entry->c.ops = &my_file_io_ops;
entry->size = 4096;
entry->mode = S_IFREG | S_IRUGO;
```

For raw data, size field must be set properly. This specifies the maximum size of the proc file access.

The read/write callbacks of raw mode are more direct than the text mode. You need to use a low-level I/O functions such as copy_from_user() and copy_to_user() to transfer the data:

```
static ssize_t my_file_io_read(struct snd_info_entry *entry,
                               void *file_private_data,
                               struct file *file,
                               char *buf,
```

```
                               size_t count,
                               loff_t pos)
{
        if (copy_to_user(buf, local_data + pos, count))
                return -EFAULT;
        return count;
}
```

If the size of the info entry has been set up properly, `count` and `pos` are guaranteed to fit within 0 and the given size. You don't have to check the range in the callbacks unless any other condition is required.

1.2.14 Power Management

If the chip is supposed to work with suspend/resume functions, you need to add power-management code to the driver. The additional code for power-management should be ifdef-ed with `CONFIG_PM`, or annotated with __maybe_unused attribute; otherwise the compiler will complain.

If the driver *fully* supports suspend/resume that is, the device can be properly resumed to its state when suspend was called, you can set the `SNDRV_PCM_INFO_RESUME` flag in the PCM info field. Usually, this is possible when the registers of the chip can be safely saved and restored to RAM. If this is set, the trigger callback is called with `SNDRV_PCM_TRIGGER_RESUME` after the resume callback completes.

Even if the driver doesn't support PM fully but partial suspend/resume is still possible, it's still worthy to implement suspend/resume callbacks. In such a case, applications would reset the status by calling *snd_pcm_prepare()* and restart the stream appropriately. Hence, you can define suspend/resume callbacks below but don't set the `SNDRV_PCM_INFO_RESUME` info flag to the PCM.

Note that the trigger with SUSPEND can always be called when *snd_pcm_suspend_all()* is called, regardless of the `SNDRV_PCM_INFO_RESUME` flag. The `RESUME` flag affects only the behavior of `snd_pcm_resume()`. (Thus, in theory, `SNDRV_PCM_TRIGGER_RESUME` isn't needed to be handled in the trigger callback when no `SNDRV_PCM_INFO_RESUME` flag is set. But, it's better to keep it for compatibility reasons.)

The driver needs to define the suspend/resume hooks according to the bus the device is connected to. In the case of PCI drivers, the callbacks look like below:

```
static int __maybe_unused snd_my_suspend(struct device *dev)
{
        .... /* do things for suspend */
        return 0;
}
static int __maybe_unused snd_my_resume(struct device *dev)
{
        .... /* do things for suspend */
        return 0;
}
```

The scheme of the real suspend job is as follows:

1. Retrieve the card and the chip data.

2. Call `snd_power_change_state()` with SNDRV_CTL_POWER_D3hot to change the power status.

3. If AC97 codecs are used, call *snd_ac97_suspend()* for each codec.

4. Save the register values if necessary.

5. Stop the hardware if necessary.

Typical code would look like:

```
static int __maybe_unused mychip_suspend(struct device *dev)
{
        /* (1) */
        struct snd_card *card = dev_get_drvdata(dev);
        struct mychip *chip = card->private_data;
        /* (2) */
        snd_power_change_state(card, SNDRV_CTL_POWER_D3hot);
        /* (3) */
        snd_ac97_suspend(chip->ac97);
        /* (4) */
        snd_mychip_save_registers(chip);
        /* (5) */
        snd_mychip_stop_hardware(chip);
        return 0;
}
```

The scheme of the real resume job is as follows:

1. Retrieve the card and the chip data.

2. Re-initialize the chip.

3. Restore the saved registers if necessary.

4. Resume the mixer, e.g. by calling *snd_ac97_resume()*.

5. Restart the hardware (if any).

6. Call `snd_power_change_state()` with SNDRV_CTL_POWER_D0 to notify the processes.

Typical code would look like:

```
static int __maybe_unused mychip_resume(struct pci_dev *pci)
{
        /* (1) */
        struct snd_card *card = dev_get_drvdata(dev);
        struct mychip *chip = card->private_data;
        /* (2) */
        snd_mychip_reinit_chip(chip);
        /* (3) */
        snd_mychip_restore_registers(chip);
        /* (4) */
        snd_ac97_resume(chip->ac97);
        /* (5) */
```

```
        snd_mychip_restart_chip(chip);
        /* (6) */
        snd_power_change_state(card, SNDRV_CTL_POWER_D0);
        return 0;
}
```

Note that, at the time this callback gets called, the PCM stream has been already suspended via its own PM ops calling *snd_pcm_suspend_all()* internally.

OK, we have all callbacks now. Let's set them up. In the initialization of the card, make sure that you can get the chip data from the card instance, typically via `private_data` field, in case you created the chip data individually:

```
static int snd_mychip_probe(struct pci_dev *pci,
                            const struct pci_device_id *pci_id)
{
        ....
        struct snd_card *card;
        struct mychip *chip;
        int err;
        ....
        err = snd_card_new(&pci->dev, index[dev], id[dev], THIS_MODULE,
                           0, &card);
        ....
        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        ....
        card->private_data = chip;
        ....
}
```

When you created the chip data with *snd_card_new()*, it's anyway accessible via `private_data` field:

```
static int snd_mychip_probe(struct pci_dev *pci,
                            const struct pci_device_id *pci_id)
{
        ....
        struct snd_card *card;
        struct mychip *chip;
        int err;
        ....
        err = snd_card_new(&pci->dev, index[dev], id[dev], THIS_MODULE,
                           sizeof(struct mychip), &card);
        ....
        chip = card->private_data;
        ....
}
```

If you need space to save the registers, allocate the buffer for it here, too, since it would be fatal if you cannot allocate a memory in the suspend phase. The allocated buffer should be released in the corresponding destructor.

And next, set suspend/resume callbacks to the pci_driver:

```
static SIMPLE_DEV_PM_OPS(snd_my_pm_ops, mychip_suspend, mychip_resume);

static struct pci_driver driver = {
        .name = KBUILD_MODNAME,
        .id_table = snd_my_ids,
        .probe = snd_my_probe,
        .remove = snd_my_remove,
        .driver.pm = &snd_my_pm_ops,
};
```

1.2.15 Module Parameters

There are standard module options for ALSA. At least, each module should have the `index`, `id` and `enable` options.

If the module supports multiple cards (usually up to 8 = SNDRV_CARDS cards), they should be arrays. The default initial values are defined already as constants for easier programming:

```
static int index[SNDRV_CARDS] = SNDRV_DEFAULT_IDX;
static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;
static int enable[SNDRV_CARDS] = SNDRV_DEFAULT_ENABLE_PNP;
```

If the module supports only a single card, they could be single variables, instead. `enable` option is not always necessary in this case, but it would be better to have a dummy option for compatibility.

The module parameters must be declared with the standard `module_param()`, `module_param_array()` and `MODULE_PARM_DESC()` macros.

Typical code would look as below:

```
#define CARD_NAME "My Chip"

module_param_array(index, int, NULL, 0444);
MODULE_PARM_DESC(index, "Index value for " CARD_NAME " soundcard.");
module_param_array(id, charp, NULL, 0444);
MODULE_PARM_DESC(id, "ID string for " CARD_NAME " soundcard.");
module_param_array(enable, bool, NULL, 0444);
MODULE_PARM_DESC(enable, "Enable " CARD_NAME " soundcard.");
```

Also, don't forget to define the module description and the license. Especially, the recent modprobe requires to define the module license as GPL, etc., otherwise the system is shown as "tainted":

```
MODULE_DESCRIPTION("Sound driver for My Chip");
MODULE_LICENSE("GPL");
```

1.2.16 Device-Managed Resources

In the examples above, all resources are allocated and released manually. But human beings are lazy in nature, especially developers are lazier. So there are some ways to automate the release part; it's the (device-)managed resources aka devres or devm family. For example, an object allocated via `devm_kmalloc()` will be freed automatically at unbinding the device.

ALSA core provides also the device-managed helper, namely, *snd_devm_card_new()* for creating a card object. Call this functions instead of the normal *snd_card_new()*, and you can forget the explicit *snd_card_free()* call, as it's called automagically at error and removal paths.

One caveat is that the call of *snd_card_free()* would be put at the beginning of the call chain only after you call *snd_card_register()*.

Also, the `private_free` callback is always called at the card free, so be careful to put the hardware clean-up procedure in `private_free` callback. It might be called even before you actually set up at an earlier error path. For avoiding such an invalid initialization, you can set `private_free` callback after *snd_card_register()* call succeeds.

Another thing to be remarked is that you should use device-managed helpers for each component as much as possible once when you manage the card in that way. Mixing up with the normal and the managed resources may screw up the release order.

1.2.17 How To Put Your Driver Into ALSA Tree

General

So far, you've learned how to write the driver codes. And you might have a question now: how to put my own driver into the ALSA driver tree? Here (finally :) the standard procedure is described briefly.

Suppose that you create a new PCI driver for the card "xyz". The card module name would be snd-xyz. The new driver is usually put into the alsa-driver tree, `sound/pci` directory in the case of PCI cards.

In the following sections, the driver code is supposed to be put into Linux kernel tree. The two cases are covered: a driver consisting of a single source file and one consisting of several source files.

Driver with A Single Source File

1. Modify sound/pci/Makefile

   Suppose you have a file xyz.c. Add the following two lines:

   ```
   snd-xyz-objs := xyz.o
   obj-$(CONFIG_SND_XYZ) += snd-xyz.o
   ```

2. Create the Kconfig entry

   Add the new entry of Kconfig for your xyz driver:

```
config SND_XYZ
  tristate "Foobar XYZ"
  depends on SND
  select SND_PCM
  help
    Say Y here to include support for Foobar XYZ soundcard.
    To compile this driver as a module, choose M here:
    the module will be called snd-xyz.
```

The line `select SND_PCM` specifies that the driver xyz supports PCM. In addition to SND_PCM, the following components are supported for select command: SND_RAWMIDI, SND_TIMER, SND_HWDEP, SND_MPU401_UART, SND_OPL3_LIB, SND_OPL4_LIB, SND_VX_LIB, SND_AC97_CODEC. Add the select command for each supported component.

Note that some selections imply the lowlevel selections. For example, PCM includes TIMER, MPU401_UART includes RAWMIDI, AC97_CODEC includes PCM, and OPL3_LIB includes HWDEP. You don't need to give the lowlevel selections again.

For the details of Kconfig script, refer to the kbuild documentation.

Drivers with Several Source Files

Suppose that the driver snd-xyz have several source files. They are located in the new subdirectory, sound/pci/xyz.

1. Add a new directory (`sound/pci/xyz`) in `sound/pci/Makefile` as below:

```
obj-$(CONFIG_SND) += sound/pci/xyz/
```

2. Under the directory `sound/pci/xyz`, create a Makefile:

```
snd-xyz-objs := xyz.o abc.o def.o
obj-$(CONFIG_SND_XYZ) += snd-xyz.o
```

3. Create the Kconfig entry

   This procedure is as same as in the last section.

1.2.18 Useful Functions

snd_printk() and friends

Note: This subsection describes a few helper functions for decorating a bit more on the standard `printk()` & co. However, in general, the use of such helpers is no longer recommended. If possible, try to stick with the standard functions like `dev_err()` or `pr_err()`.

ALSA provides a verbose version of the `printk()` function. If a kernel config `CONFIG_SND_VERBOSE_PRINTK` is set, this function prints the given message together with the file name and the line of the caller. The `KERN_XXX` prefix is processed as well as the original

`printk()` does, so it's recommended to add this prefix, e.g. snd_printk(KERN_ERR "Oh my, sorry, it's extremely bad!\n");

There are also `printk()`'s for debugging. *snd_printd()* can be used for general debugging purposes. If `CONFIG_SND_DEBUG` is set, this function is compiled, and works just like *snd_printk()*. If the ALSA is compiled without the debugging flag, it's ignored.

snd_printdd() is compiled in only when `CONFIG_SND_DEBUG_VERBOSE` is set.

snd_BUG()

It shows the BUG? message and stack trace as well as *snd_BUG_ON()* at the point. It's useful to show that a fatal error happens there.

When no debug flag is set, this macro is ignored.

snd_BUG_ON()

snd_BUG_ON() macro is similar with `WARN_ON()` macro. For example, snd_BUG_ON(!pointer); or it can be used as the condition, if (snd_BUG_ON(non_zero_is_bug)) return -EINVAL;

The macro takes an conditional expression to evaluate. When `CONFIG_SND_DEBUG`, is set, if the expression is non-zero, it shows the warning message such as `BUG? (xxx)` normally followed by stack trace. In both cases it returns the evaluated value.

1.2.19 Acknowledgments

I would like to thank Phil Kerr for his help for improvement and corrections of this document.

Kevin Conder reformatted the original plain-text to the DocBook format.

Giuliano Pochini corrected typos and contributed the example codes in the hardware constraints section.

DESIGNS AND IMPLEMENTATIONS

2.1 Standard ALSA Control Names

This document describes standard names of mixer controls.

2.1.1 Standard Syntax

Syntax: [LOCATION] SOURCE [CHANNEL] [DIRECTION] FUNCTION

DIRECTION

<nothing>	both directions
Playback	one direction
Capture	one direction
Bypass Playback	one direction
Bypass Capture	one direction

FUNCTION

Switch	on/off switch
Volume	amplifier
Route	route control, hardware specific

CHANNEL

<nothing>	channel independent, or applies to all channels
Front	front left/right channels
Surround	rear left/right in 4.0/5.1 surround
CLFE	C/LFE channels
Center	center channel
LFE	LFE channel
Side	side left/right for 7.1 surround

LOCATION (Physical location of source)

Front	front position
Rear	rear position
Dock	on docking station
Internal	internal

SOURCE

Master	
Master Mono	
Hardware Master	
Speaker	internal speaker
Bass Speaker	internal LFE speaker
Headphone	
Line Out	
Beep	beep generator
Phone	
Phone Input	
Phone Output	
Synth	
FM	
Mic	
Headset Mic	mic part of combined headset jack - 4-pin headphone + mic
Headphone Mic	mic part of either/or - 3-pin headphone or mic
Line	input only, use "Line Out" for output
CD	
Video	
Zoom Video	
Aux	
PCM	
PCM Pan	
Loopback	
Analog Loopback	D/A -> A/D loopback
Digital Loopback	playback -> capture loopback - without analog path
Mono	
Mono Output	
Multi	
ADC	
Wave	
Music	
I2S	
IEC958	
HDMI	
SPDIF	output only
SPDIF In	
Digital In	

Table 1 – continued from previous page

HDMI/DP	either HDMI or DisplayPort

2.1.2 Exceptions (deprecated)

[Analogue|Digital] Capture Source	
[Analogue|Digital] Capture Switch	aka input gain switch
[Analogue|Digital] Capture Volume	aka input gain volume
[Analogue|Digital] Playback Switch	aka output gain switch
[Analogue|Digital] Playback Volume	aka output gain volume
Tone Control - Switch	
Tone Control - Bass	
Tone Control - Treble	
3D Control - Switch	
3D Control - Center	
3D Control - Depth	
3D Control - Wide	
3D Control - Space	
3D Control - Level	
Mic Boost [(?dB)]	

2.1.3 PCM interface

Sample Clock Source	{ "Word", "Internal", "AutoSync" }
Clock Sync Status	{ "Lock", "Sync", "No Lock" }
External Rate	external capture rate
Capture Rate	capture rate taken from external source

2.1.4 IEC958 (S/PDIF) interface

IEC958 [...] [Playback|Capture] Switch	turn on/off the IEC958 interface
IEC958 [...] [Playback|Capture] Volume	digital volume control
IEC958 [...] [Playback|Capture] Default	default or global value - read/write
IEC958 [...] [Playback|Capture] Mask	consumer and professional mask
IEC958 [...] [Playback|Capture] Con Mask	consumer mask
IEC958 [...] [Playback|Capture] Pro Mask	professional mask
IEC958 [...] [Playback|Capture] PCM Stream	the settings assigned to a PCM stream
IEC958 Q-subcode [Playback|Capture] Default	Q-subcode bits
IEC958 Preamble [Playback|Capture] Default	burst preamble words (4*16bits)

2.2 ALSA PCM channel-mapping API

Takashi Iwai <tiwai@suse.de>

2.2.1 General

The channel mapping API allows user to query the possible channel maps and the current channel map, also optionally to modify the channel map of the current stream.

A channel map is an array of position for each PCM channel. Typically, a stereo PCM stream has a channel map of `{ front_left, front_right }` while a 4.0 surround PCM stream has a channel map of `{ front left, front right, rear left, rear right }`.

The problem, so far, was that we had no standard channel map explicitly, and applications had no way to know which channel corresponds to which (speaker) position. Thus, applications applied wrong channels for 5.1 outputs, and you hear suddenly strange sound from rear. Or, some devices secretly assume that center/LFE is the third/fourth channels while others that C/LFE as 5th/6th channels.

Also, some devices such as HDMI are configurable for different speaker positions even with the same number of total channels. However, there was no way to specify this because of lack of channel map specification. These are the main motivations for the new channel mapping API.

2.2.2 Design

Actually, "the channel mapping API" doesn't introduce anything new in the kernel/user-space ABI perspective. It uses only the existing control element features.

As a ground design, each PCM substream may contain a control element providing the channel mapping information and configuration. This element is specified by:

- iface = SNDRV_CTL_ELEM_IFACE_PCM
- name = "Playback Channel Map" or "Capture Channel Map"
- device = the same device number for the assigned PCM substream
- index = the same index number for the assigned PCM substream

Note the name is different depending on the PCM substream direction.

Each control element provides at least the TLV read operation and the read operation. Optionally, the write operation can be provided to allow user to change the channel map dynamically.

TLV

The TLV operation gives the list of available channel maps. A list item of a channel map is usually a TLV of `type data-bytes ch0 ch1 ch2...` where type is the TLV type value, the second argument is the total bytes (not the numbers) of channel values, and the rest are the position value for each channel.

As a TLV type, either `SNDRV_CTL_TLVT_CHMAP_FIXED`, `SNDRV_CTL_TLV_CHMAP_VAR` or `SNDRV_CTL_TLVT_CHMAP_PAIRED` can be used. The _FIXED type is for a channel map with the fixed channel position while the latter two are for flexible channel positions. _VAR type is

for a channel map where all channels are freely swappable and _PAIRED type is where pair-wise channels are swappable. For example, when you have {FL/FR/RL/RR} channel map, _PAIRED type would allow you to swap only {RL/RR/FL/FR} while _VAR type would allow even swapping FL and RR.

These new TLV types are defined in `sound/tlv.h`.

The available channel position values are defined in `sound/asound.h`, here is a cut:

```
/* channel positions */
enum {
      SNDRV_CHMAP_UNKNOWN = 0,
      SNDRV_CHMAP_NA,           /* N/A, silent */
      SNDRV_CHMAP_MONO,         /* mono stream */
      /* this follows the alsa-lib mixer channel value + 3 */
      SNDRV_CHMAP_FL,           /* front left */
      SNDRV_CHMAP_FR,           /* front right */
      SNDRV_CHMAP_RL,           /* rear left */
      SNDRV_CHMAP_RR,           /* rear right */
      SNDRV_CHMAP_FC,           /* front center */
      SNDRV_CHMAP_LFE,          /* LFE */
      SNDRV_CHMAP_SL,           /* side left */
      SNDRV_CHMAP_SR,           /* side right */
      SNDRV_CHMAP_RC,           /* rear center */
      /* new definitions */
      SNDRV_CHMAP_FLC,          /* front left center */
      SNDRV_CHMAP_FRC,          /* front right center */
      SNDRV_CHMAP_RLC,          /* rear left center */
      SNDRV_CHMAP_RRC,          /* rear right center */
      SNDRV_CHMAP_FLW,          /* front left wide */
      SNDRV_CHMAP_FRW,          /* front right wide */
      SNDRV_CHMAP_FLH,          /* front left high */
      SNDRV_CHMAP_FCH,          /* front center high */
      SNDRV_CHMAP_FRH,          /* front right high */
      SNDRV_CHMAP_TC,           /* top center */
      SNDRV_CHMAP_TFL,          /* top front left */
      SNDRV_CHMAP_TFR,          /* top front right */
      SNDRV_CHMAP_TFC,          /* top front center */
      SNDRV_CHMAP_TRL,          /* top rear left */
      SNDRV_CHMAP_TRR,          /* top rear right */
      SNDRV_CHMAP_TRC,          /* top rear center */
      SNDRV_CHMAP_LAST = SNDRV_CHMAP_TRC,
};
```

When a PCM stream can provide more than one channel map, you can provide multiple channel maps in a TLV container type. The TLV data to be returned will contain such as:

```
SNDRV_CTL_TLVT_CONTAINER 96
    SNDRV_CTL_TLVT_CHMAP_FIXED 4 SNDRV_CHMAP_FC
    SNDRV_CTL_TLVT_CHMAP_FIXED 8 SNDRV_CHMAP_FL SNDRV_CHMAP_FR
    SNDRV_CTL_TLVT_CHMAP_FIXED 16 NDRV_CHMAP_FL SNDRV_CHMAP_FR \
        SNDRV_CHMAP_RL SNDRV_CHMAP_RR
```

The channel position is provided in LSB 16bits. The upper bits are used for bit flags.

```
#define SNDRV_CHMAP_POSITION_MASK      0xffff
#define SNDRV_CHMAP_PHASE_INVERSE      (0x01 << 16)
#define SNDRV_CHMAP_DRIVER_SPEC        (0x02 << 16)
```

`SNDRV_CHMAP_PHASE_INVERSE` indicates the channel is phase inverted, (thus summing left and right channels would result in almost silence). Some digital mic devices have this.

When `SNDRV_CHMAP_DRIVER_SPEC` is set, all the channel position values don't follow the standard definition above but driver-specific.

Read Operation

The control read operation is for providing the current channel map of the given stream. The control element returns an integer array containing the position of each channel.

When this is performed before the number of the channel is specified (i.e. hw_params is set), it should return all channels set to `UNKNOWN`.

Write Operation

The control write operation is optional, and only for devices that can change the channel configuration on the fly, such as HDMI. User needs to pass an integer value containing the valid channel positions for all channels of the assigned PCM substream.

This operation is allowed only at PCM PREPARED state. When called in other states, it shall return an error.

2.3 ALSA Compress-Offload API

Pierre-Louis.Bossart <pierre-louis.bossart@linux.intel.com>

Vinod Koul <vinod.koul@linux.intel.com>

2.3.1 Overview

Since its early days, the ALSA API was defined with PCM support or constant bitrates payloads such as IEC61937 in mind. Arguments and returned values in frames are the norm, making it a challenge to extend the existing API to compressed data streams.

In recent years, audio digital signal processors (DSP) were integrated in system-on-chip designs, and DSPs are also integrated in audio codecs. Processing compressed data on such DSPs results in a dramatic reduction of power consumption compared to host-based processing. Support for such hardware has not been very good in Linux, mostly because of a lack of a generic API available in the mainline kernel.

Rather than requiring a compatibility break with an API change of the ALSA PCM interface, a new 'Compressed Data' API is introduced to provide a control and data-streaming interface for audio DSPs.

The design of this API was inspired by the 2-year experience with the Intel Moorestown SOC, with many corrections required to upstream the API in the mainline kernel instead of the staging tree and make it usable by others.

2.3.2 Requirements

The main requirements are:

- separation between byte counts and time. Compressed formats may have a header per file, per frame, or no header at all. The payload size may vary from frame-to-frame. As a result, it is not possible to estimate reliably the duration of audio buffers when handling compressed data. Dedicated mechanisms are required to allow for reliable audio-video synchronization, which requires precise reporting of the number of samples rendered at any given time.

- Handling of multiple formats. PCM data only requires a specification of the sampling rate, number of channels and bits per sample. In contrast, compressed data comes in a variety of formats. Audio DSPs may also provide support for a limited number of audio encoders and decoders embedded in firmware, or may support more choices through dynamic download of libraries.

- Focus on main formats. This API provides support for the most popular formats used for audio and video capture and playback. It is likely that as audio compression technology advances, new formats will be added.

- Handling of multiple configurations. Even for a given format like AAC, some implementations may support AAC multichannel but HE-AAC stereo. Likewise WMA10 level M3 may require too much memory and cpu cycles. The new API needs to provide a generic way of listing these formats.

- Rendering/Grabbing only. This API does not provide any means of hardware acceleration, where PCM samples are provided back to user-space for additional processing. This API focuses instead on streaming compressed data to a DSP, with the assumption that the decoded samples are routed to a physical output or logical back-end.

- Complexity hiding. Existing user-space multimedia frameworks all have existing enums/structures for each compressed format. This new API assumes the existence of a platform-specific compatibility layer to expose, translate and make use of the capabilities of the audio DSP, eg. Android HAL or PulseAudio sinks. By construction, regular applications are not supposed to make use of this API.

2.3.3 Design

The new API shares a number of concepts with the PCM API for flow control. Start, pause, resume, drain and stop commands have the same semantics no matter what the content is.

The concept of memory ring buffer divided in a set of fragments is borrowed from the ALSA PCM API. However, only sizes in bytes can be specified.

Seeks/trick modes are assumed to be handled by the host.

The notion of rewinds/forwards is not supported. Data committed to the ring buffer cannot be invalidated, except when dropping all buffers.

The Compressed Data API does not make any assumptions on how the data is transmitted to the audio DSP. DMA transfers from main memory to an embedded audio cluster or to a SPI interface for external DSPs are possible. As in the ALSA PCM case, a core set of routines is exposed; each driver implementer will have to write support for a set of mandatory routines and possibly make use of optional ones.

The main additions are

get_caps
> This routine returns the list of audio formats supported. Querying the codecs on a capture stream will return encoders, decoders will be listed for playback streams.

get_codec_caps
> For each codec, this routine returns a list of capabilities. The intent is to make sure all the capabilities correspond to valid settings, and to minimize the risks of configuration failures. For example, for a complex codec such as AAC, the number of channels supported may depend on a specific profile. If the capabilities were exposed with a single descriptor, it may happen that a specific combination of profiles/channels/formats may not be supported. Likewise, embedded DSPs have limited memory and cpu cycles, it is likely that some implementations make the list of capabilities dynamic and dependent on existing workloads. In addition to codec settings, this routine returns the minimum buffer size handled by the implementation. This information can be a function of the DMA buffer sizes, the number of bytes required to synchronize, etc, and can be used by userspace to define how much needs to be written in the ring buffer before playback can start.

set_params
> This routine sets the configuration chosen for a specific codec. The most important field in the parameters is the codec type; in most cases decoders will ignore other fields, while encoders will strictly comply to the settings

get_params
> This routines returns the actual settings used by the DSP. Changes to the settings should remain the exception.
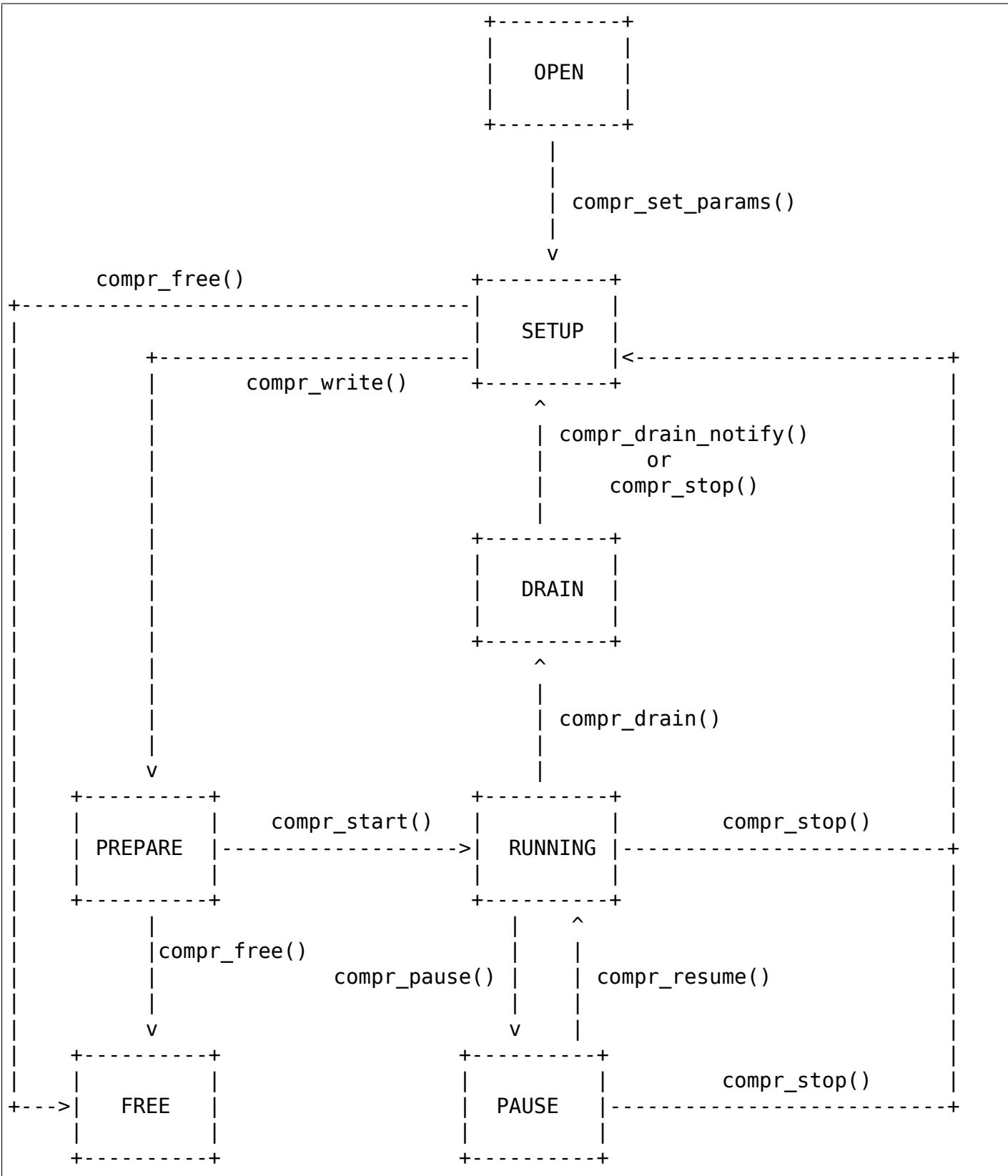
get_timestamp
> The timestamp becomes a multiple field structure. It lists the number of bytes transferred, the number of samples processed and the number of samples rendered/grabbed. All these values can be used to determine the average bitrate, figure out if the ring buffer needs to be refilled or the delay due to decoding/encoding/io on the DSP.

Note that the list of codecs/profiles/modes was derived from the OpenMAX AL specification instead of reinventing the wheel. Modifications include: - Addition of FLAC and IEC formats - Merge of encoder/decoder capabilities - Profiles/modes listed as bitmasks to make descriptors more compact - Addition of set_params for decoders (missing in OpenMAX AL) - Addition of AMR/AMR-WB encoding modes (missing in OpenMAX AL) - Addition of format information for WMA - Addition of encoding options when required (derived from OpenMAX IL) - Addition of rateControlSupported (missing in OpenMAX AL)

2.3.4 State Machine

The compressed audio stream state machine is described below

```
                                    +---------+
                                    |         |
                                    |  OPEN   |
                                    |         |
                                    +---------+
                                         |
                                         |
                                         | compr_set_params()
                                         |
                                         v
        compr_free()                +---------+
  +---------------------------------|         |
  |                                 |  SETUP  |
  |         +-----------------------|         |<----------------------+
  |         |       compr_write()   +---------+                       |
  |         |                            ^                            |
  |         |                            | compr_drain_notify()       |
  |         |                            |         or                 |
  |         |                            |      compr_stop()          |
  |         |                            |                            |
  |         |                       +---------+                       |
  |         |                       |         |                       |
  |         |                       |  DRAIN  |                       |
  |         |                       |         |                       |
  |         |                       +---------+                       |
  |         |                            ^                            |
  |         |                            |                            |
  |         |                            | compr_drain()              |
  |         |                            |                            |
  |         v                            |                            |
  |   +---------+                   +---------+                       |
  |   |         | compr_start()     |         |     compr_stop()      |
  |   | PREPARE |------------------>| RUNNING |-----------------------+
  |   |         |                   |         |                       |
  |   +---------+                   +---------+                       |
  |        |                            |    ^                        |
  |        |compr_free()                |    |                        |
  |        |            compr_pause() | | compr_resume()              |
  |        |                            |    |                        |
  |        v                            v    |                        |
  |   +---------+                   +---------+                       |
  |   |         |                   |         |     compr_stop()      |
  +-->|  FREE   |                   |  PAUSE  |-----------------------+
      |         |                   |         |
      +---------+                   +---------+
```

2.3.5 Gapless Playback

When playing thru an album, the decoders have the ability to skip the encoder delay and padding and directly move from one track content to another. The end user can perceive this as gapless playback as we don't have silence while switching from one track to another

Also, there might be low-intensity noises due to encoding. Perfect gapless is difficult to reach with all types of compressed data, but works fine with most music content. The decoder needs to know the encoder delay and encoder padding. So we need to pass this to DSP. This metadata is extracted from ID3/MP4 headers and are not present by default in the bitstream, hence the need for a new interface to pass this information to the DSP. Also DSP and userspace needs to switch from one track to another and start using data for second track.

The main additions are:

set_metadata
> This routine sets the encoder delay and encoder padding. This can be used by decoder to strip the silence. This needs to be set before the data in the track is written.

set_next_track
> This routine tells DSP that metadata and write operation sent after this would correspond to subsequent track
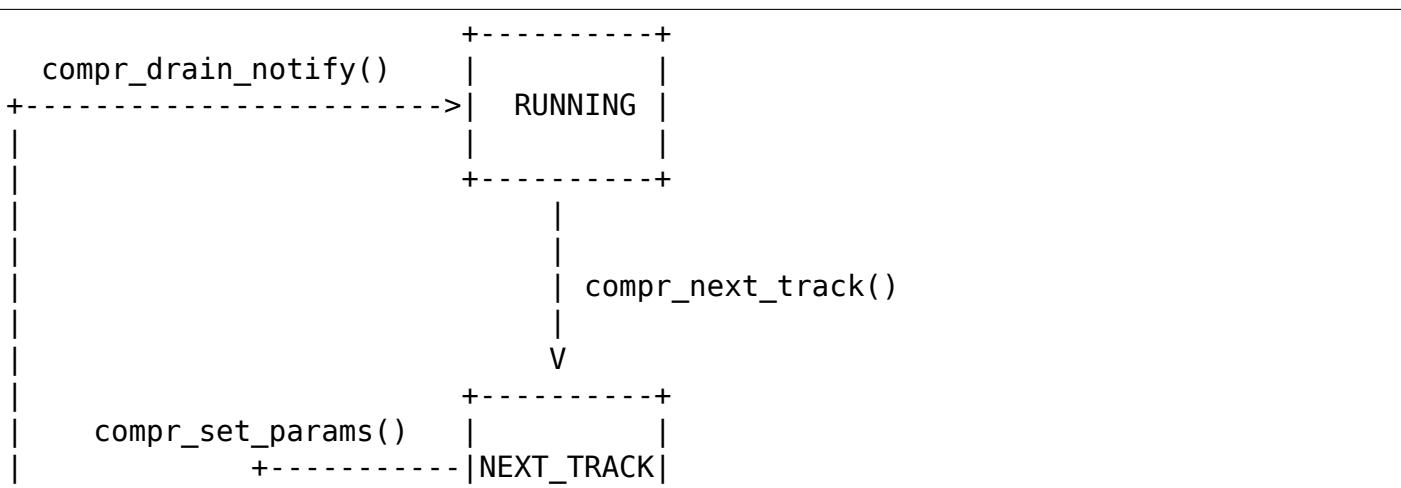
partial drain
> This is called when end of file is reached. The userspace can inform DSP that EOF is reached and now DSP can start skipping padding delay. Also next write data would belong to next track
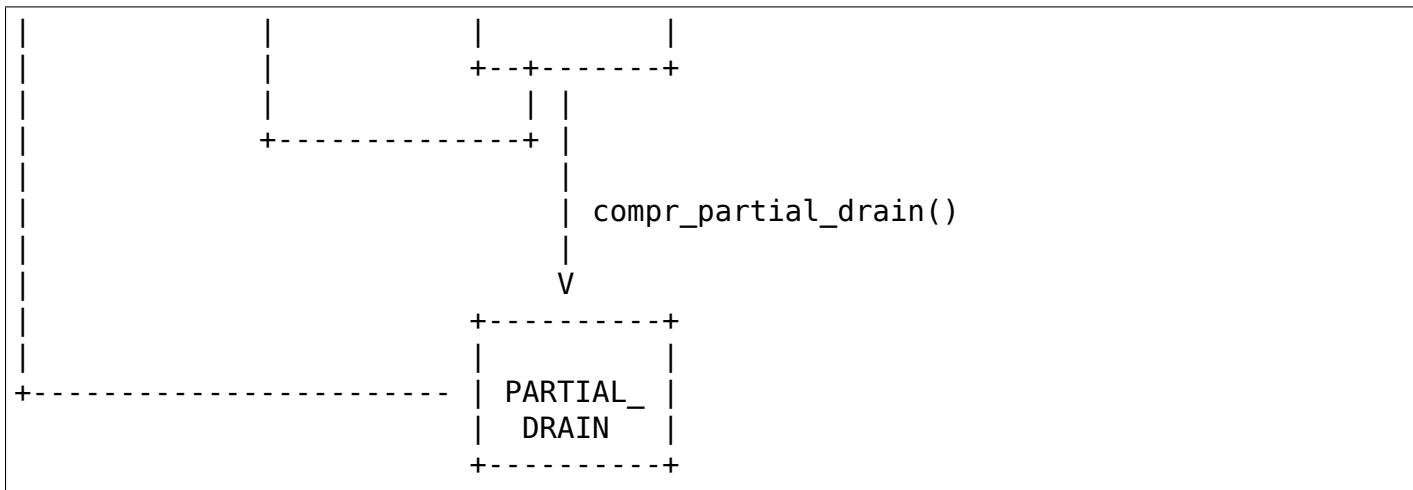
Sequence flow for gapless would be: - Open - Get caps / codec caps - Set params - Set metadata of the first track - Fill data of the first track - Trigger start - User-space finished sending all, - Indicate next track data by sending set_next_track - Set metadata of the next track - then call partial_drain to flush most of buffer in DSP - Fill data of the next track - DSP switches to second track

(note: order for partial_drain and write for next track can be reversed as well)

2.3.6 Gapless Playback SM

For Gapless, we move from running state to partial drain and back, along with setting of meta_data and signalling for next track

```
                        +----------+
  compr_drain_notify()  |          |
+----------------------->|  RUNNING |
|                        |          |
|                        +----------+
|                             |
|                             |
|                             |
|                             | compr_next_track()
|                             |
|                             V
|                        +----------+
|     compr_set_params() |          |
|            +-----------|NEXT_TRACK|
```

```
|                    |              |           |
|                    |              +--+-------+
|                    |              |  | |
|              +-------------+ |
|                    |              |
|                    |              | compr_partial_drain()
|                    |              |
|                    |              V
|                    +----------+
|                    |          |
+----------------------- | PARTIAL_ |
                    |  DRAIN   |
                    +----------+
```

2.3.7 Not supported

- Support for VoIP/circuit-switched calls is not the target of this API. Support for dynamic bit-rate changes would require a tight coupling between the DSP and the host stack, limiting power savings.

- Packet-loss concealment is not supported. This would require an additional interface to let the decoder synthesize data when frames are lost during transmission. This may be added in the future.

- Volume control/routing is not handled by this API. Devices exposing a compressed data interface will be considered as regular ALSA devices; volume changes and routing information will be provided with regular ALSA kcontrols.

- Embedded audio effects. Such effects should be enabled in the same manner, no matter if the input was PCM or compressed.

- multichannel IEC encoding. Unclear if this is required.

- Encoding/decoding acceleration is not supported as mentioned above. It is possible to route the output of a decoder to a capture stream, or even implement transcoding capabilities. This routing would be enabled with ALSA kcontrols.

- Audio policy/resource management. This API does not provide any hooks to query the utilization of the audio DSP, nor any preemption mechanisms.

- No notion of underrun/overrun. Since the bytes written are compressed in nature and data written/read doesn't translate directly to rendered output in time, this does not deal with underrun/overrun and maybe dealt in user-library

2.3.8 Credits

- Mark Brown and Liam Girdwood for discussions on the need for this API
- Harsha Priya for her work on intel_sst compressed API
- Rakesh Ughreja for valuable feedback
- Sing Nallasellan, Sikkandar Madar and Prasanna Samaga for demonstrating and quantifying the benefits of audio offload on a real platform.

2.4 ALSA PCM Timestamping

The ALSA API can provide two different system timestamps:

- Trigger_tstamp is the system time snapshot taken when the .trigger callback is invoked. This snapshot is taken by the ALSA core in the general case, but specific hardware may have synchronization capabilities or conversely may only be able to provide a correct estimate with a delay. In the latter two cases, the low-level driver is responsible for updating the trigger_tstamp at the most appropriate and precise moment. Applications should not rely solely on the first trigger_tstamp but update their internal calculations if the driver provides a refined estimate with a delay.

- tstamp is the current system timestamp updated during the last event or application query. The difference (tstamp - trigger_tstamp) defines the elapsed time.
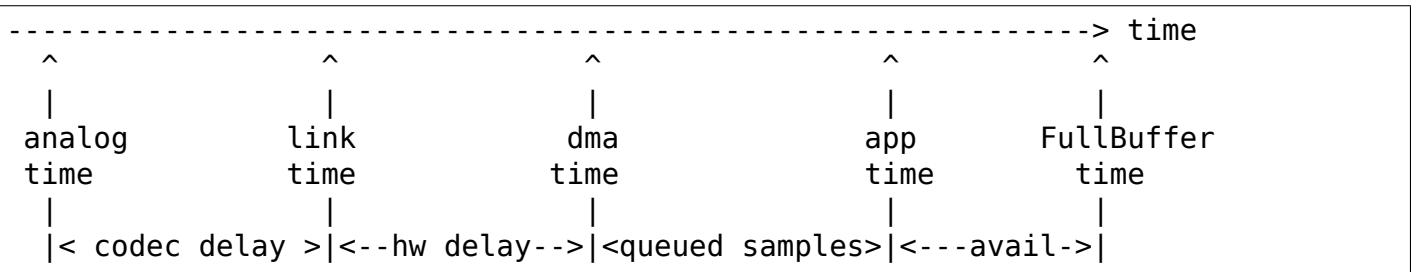
The ALSA API provides two basic pieces of information, avail and delay, which combined with the trigger and current system timestamps allow for applications to keep track of the 'fullness' of the ring buffer and the amount of queued samples.
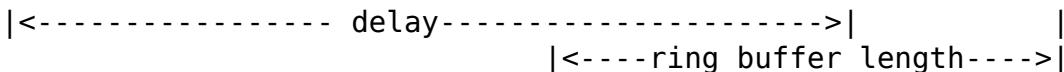
The use of these different pointers and time information depends on the application needs:

- `avail` reports how much can be written in the ring buffer
- `delay` reports the time it will take to hear a new sample after all queued samples have been played out.

When timestamps are enabled, the avail/delay information is reported along with a snapshot of system time. Applications can select from CLOCK_REALTIME (NTP corrections including going backwards), CLOCK_MONOTONIC (NTP corrections but never going backwards), CLOCK_MONOTIC_RAW (without NTP corrections) and change the mode dynamically with sw_params

The ALSA API also provide an audio_tstamp which reflects the passage of time as measured by different components of audio hardware. In ascii-art, this could be represented as follows (for the playback case):

```
-------------------------------------------------------------> time
  ^              ^              ^              ^              ^
  |              |              |              |              |
 analog         link           dma            app         FullBuffer
 time           time           time           time           time
  |              |              |              |              |
  |< codec delay >|<--hw delay-->|<queued samples>|<---avail->|
```

```
|<---------------- delay---------------------->|            |
                             |<----ring buffer length---->|
```

The analog time is taken at the last stage of the playback, as close as possible to the actual transducer

The link time is taken at the output of the SoC/chipset as the samples are pushed on a link. The link time can be directly measured if supported in hardware by sample counters or wallclocks (e.g. with HDAudio 24MHz or PTP clock for networked solutions) or indirectly estimated (e.g. with the frame counter in USB).

The DMA time is measured using counters - typically the least reliable of all measurements due to the bursty nature of DMA transfers.

The app time corresponds to the time tracked by an application after writing in the ring buffer.

The application can query the hardware capabilities, define which audio time it wants reported by selecting the relevant settings in audio_tstamp_config fields, thus get an estimate of the timestamp accuracy. It can also request the delay-to-analog be included in the measurement. Direct access to the link time is very interesting on platforms that provide an embedded DSP; measuring directly the link time with dedicated hardware, possibly synchronized with system time, removes the need to keep track of internal DSP processing times and latency.

In case the application requests an audio tstamp that is not supported in hardware/low-level driver, the type is overridden as DEFAULT and the timestamp will report the DMA time based on the hw_pointer value.

For backwards compatibility with previous implementations that did not provide timestamp selection, with a zero-valued COMPAT timestamp type the results will default to the HDAudio wall clock for playback streams and to the DMA time (hw_ptr) in all other cases.

The audio timestamp accuracy can be returned to user-space, so that appropriate decisions are made:

- for dma time (default), the granularity of the transfers can be inferred from the steps between updates and in turn provide information on how much the application pointer can be rewound safely.

- the link time can be used to track long-term drifts between audio and system time using the (tstamp-trigger_tstamp)/audio_tstamp ratio, the precision helps define how much smoothing/low-pass filtering is required. The link time can be either reset on startup or reported as is (the latter being useful to compare progress of different streams - but may require the wallclock to be always running and not wrap-around during idle periods). If supported in hardware, the absolute link time could also be used to define a precise start time (patches WIP)

- including the delay in the audio timestamp may counter-intuitively not increase the precision of timestamps, e.g. if a codec includes variable-latency DSP processing or a chain of hardware components the delay is typically not known with precision.

The accuracy is reported in nanosecond units (using an unsigned 32-bit word), which gives a max precision of 4.29s, more than enough for audio applications...

Due to the varied nature of timestamping needs, even for a single application, the audio_tstamp_config can be changed dynamically. In the STATUS ioctl, the parameters are read-only and do not allow for any application selection. To work around this limitation without

impacting legacy applications, a new STATUS_EXT ioctl is introduced with read/write parameters. ALSA-lib will be modified to make use of STATUS_EXT and effectively deprecate STATUS.

The ALSA API only allows for a single audio timestamp to be reported at a time. This is a conscious design decision, reading the audio timestamps from hardware registers or from IPC takes time, the more timestamps are read the more imprecise the combined measurements are. To avoid any interpretation issues, a single (system, audio) timestamp is reported. Applications that need different timestamps will be required to issue multiple queries and perform an interpolation of the results

In some hardware-specific configuration, the system timestamp is latched by a low-level audio subsystem, and the information provided back to the driver. Due to potential delays in the communication with the hardware, there is a risk of misalignment with the avail and delay information. To make sure applications are not confused, a driver_timestamp field is added in the snd_pcm_status structure; this timestamp shows when the information is put together by the driver before returning from the STATUS and STATUS_EXT ioctl. in most cases this driver_timestamp will be identical to the regular system tstamp.

Examples of timestamping with HDAudio:

1. DMA timestamp, no compensation for DMA+analog delay

```
$ ./audio_time  -p --ts_type=1
playback: systime: 341121338 nsec, audio time 342000000 nsec,        systime␣
↪delta -878662
playback: systime: 426236663 nsec, audio time 427187500 nsec,        systime␣
↪delta -950837
playback: systime: 597080580 nsec, audio time 598000000 nsec,        systime␣
↪delta -919420
playback: systime: 682059782 nsec, audio time 683020833 nsec,        systime␣
↪delta -961051
playback: systime: 852896415 nsec, audio time 853854166 nsec,        systime␣
↪delta -957751
playback: systime: 937903344 nsec, audio time 938854166 nsec,        systime␣
↪delta -950822
```

2. DMA timestamp, compensation for DMA+analog delay

```
$ ./audio_time  -p --ts_type=1 -d
playback: systime: 341053347 nsec, audio time 341062500 nsec,        systime␣
↪delta -9153
playback: systime: 426072447 nsec, audio time 426062500 nsec,        systime␣
↪delta 9947
playback: systime: 596899518 nsec, audio time 596895833 nsec,        systime␣
↪delta 3685
playback: systime: 681915317 nsec, audio time 681916666 nsec,        systime␣
↪delta -1349
playback: systime: 852741306 nsec, audio time 852750000 nsec,        systime␣
↪delta -8694
```

3. link timestamp, compensation for DMA+analog delay

```
$ ./audio_time  -p --ts_type=2 -d
playback: systime: 341060004 nsec, audio time 341062791 nsec,        systime␣
↪delta -2787
```

```
playback: systime: 426242074 nsec, audio time 426244875 nsec,          systime␣
↪delta -2801
playback: systime: 597080992 nsec, audio time 597084583 nsec,          systime␣
↪delta -3591
playback: systime: 682084512 nsec, audio time 682088291 nsec,          systime␣
↪delta -3779
playback: systime: 852936229 nsec, audio time 852940916 nsec,          systime␣
↪delta -4687
playback: systime: 938107562 nsec, audio time 938112708 nsec,          systime␣
↪delta -5146
```

Example 1 shows that the timestamp at the DMA level is close to 1ms ahead of the actual playback time (as a side time this sort of measurement can help define rewind safeguards). Compensating for the DMA-link delay in example 2 helps remove the hardware buffering but the information is still very jittery, with up to one sample of error. In example 3 where the timestamps are measured with the link wallclock, the timestamps show a monotonic behavior and a lower dispersion.

Example 3 and 4 are with USB audio class. Example 3 shows a high offset between audio time and system time due to buffering. Example 4 shows how compensating for the delay exposes a 1ms accuracy (due to the use of the frame counter by the driver)

Example 3: DMA timestamp, no compensation for delay, delta of ~5ms

```
$ ./audio_time -p -Dhw:1 -t1
playback: systime: 120174019 nsec, audio time 125000000 nsec,          systime␣
↪delta -4825981
playback: systime: 245041136 nsec, audio time 250000000 nsec,          systime␣
↪delta -4958864
playback: systime: 370106088 nsec, audio time 375000000 nsec,          systime␣
↪delta -4893912
playback: systime: 495040065 nsec, audio time 500000000 nsec,          systime␣
↪delta -4959935
playback: systime: 620038179 nsec, audio time 625000000 nsec,          systime␣
↪delta -4961821
playback: systime: 745087741 nsec, audio time 750000000 nsec,          systime␣
↪delta -4912259
playback: systime: 870037336 nsec, audio time 875000000 nsec,          systime␣
↪delta -4962664
```

Example 4: DMA timestamp, compensation for delay, delay of ~1ms

```
$ ./audio_time -p -Dhw:1 -t1 -d
playback: systime: 120190520 nsec, audio time 120000000 nsec,          systime␣
↪delta 190520
playback: systime: 245036740 nsec, audio time 244000000 nsec,          systime␣
↪delta 1036740
playback: systime: 370034081 nsec, audio time 369000000 nsec,          systime␣
↪delta 1034081
playback: systime: 495159907 nsec, audio time 494000000 nsec,          systime␣
↪delta 1159907
playback: systime: 620098824 nsec, audio time 619000000 nsec,          systime␣
↪delta 1098824
```

```
playback: systime: 745031847 nsec, audio time 744000000 nsec,          systime␣
→delta 1031847
```

2.5 ALSA Jack Controls

2.5.1 Why we need Jack kcontrols

ALSA uses kcontrols to export audio controls(switch, volume, Mux, ...) to user space. This means userspace applications like pulseaudio can switch off headphones and switch on speakers when no headphones are plugged in.

The old ALSA jack code only created input devices for each registered jack. These jack input devices are not readable by userspace devices that run as non root.

The new jack code creates embedded jack kcontrols for each jack that can be read by any process.

This can be combined with UCM to allow userspace to route audio more intelligently based on jack insertion or removal events.

2.5.2 Jack Kcontrol Internals

Each jack will have a kcontrol list, so that we can create a kcontrol and attach it to the jack, at jack creation stage. We can also add a kcontrol to an existing jack, at anytime when required.

Those kcontrols will be freed automatically when the Jack is freed.

2.5.3 How to use jack kcontrols

In order to keep compatibility, *snd_jack_new()* has been modified by adding two params:

initial_kctl
> if true, create a kcontrol and add it to the jack list.

phantom_jack
> Don't create a input device for phantom jacks.

HDA jacks can set phantom_jack to true in order to create a phantom jack and set initial_kctl to true to create an initial kcontrol with the correct id.

ASoC jacks should set initial_kctl as false. The pin name will be assigned as the jack kcontrol name.

2.6 Tracepoints in ALSA

2017/07/02 Takasahi Sakamoto

2.6.1 Tracepoints in ALSA PCM core

ALSA PCM core registers `snd_pcm` subsystem to kernel tracepoint system. This subsystem includes two categories of tracepoints; for state of PCM buffer and for processing of PCM hardware parameters. These tracepoints are available when corresponding kernel configurations are enabled. When `CONFIG_SND_DEBUG` is enabled, the latter tracepoints are available. When additional `SND_PCM_XRUN_DEBUG` is enabled too, the former trace points are enabled.

Tracepoints for state of PCM buffer

This category includes four tracepoints; `hwptr`, `applptr`, `xrun` and `hw_ptr_error`.

Tracepoints for processing of PCM hardware parameters

This category includes two tracepoints; `hw_mask_param` and `hw_interval_param`.

In a design of ALSA PCM core, data transmission is abstracted as PCM substream. Applications manage PCM substream to maintain data transmission for PCM frames. Before starting the data transmission, applications need to configure PCM substream. In this procedure, PCM hardware parameters are decided by interaction between applications and ALSA PCM core. Once decided, runtime of the PCM substream keeps the parameters.

The parameters are described in struct snd_pcm_hw_params. This structure includes several types of parameters. Applications set preferable value to these parameters, then execute ioctl(2) with SNDRV_PCM_IOCTL_HW_REFINE or SNDRV_PCM_IOCTL_HW_PARAMS. The former is used just for refining available set of parameters. The latter is used for an actual decision of the parameters.

The struct snd_pcm_hw_params structure has below members:

flags
Configurable. ALSA PCM core and some drivers handle this flag to select convenient parameters or change their behaviour.

masks
Configurable. This type of parameter is described in struct snd_mask and represent mask values. As of PCM protocol v2.0.13, three types are defined.

- SNDRV_PCM_HW_PARAM_ACCESS
- SNDRV_PCM_HW_PARAM_FORMAT
- SNDRV_PCM_HW_PARAM_SUBFORMAT

intervals
Configurable. This type of parameter is described in struct snd_interval and represent values with a range. As of PCM protocol v2.0.13, twelve types are defined.

- SNDRV_PCM_HW_PARAM_SAMPLE_BITS

- SNDRV_PCM_HW_PARAM_FRAME_BITS

- SNDRV_PCM_HW_PARAM_CHANNELS

- SNDRV_PCM_HW_PARAM_RATE

- SNDRV_PCM_HW_PARAM_PERIOD_TIME

- SNDRV_PCM_HW_PARAM_PERIOD_SIZE

- SNDRV_PCM_HW_PARAM_PERIOD_BYTES

- SNDRV_PCM_HW_PARAM_PERIODS

- SNDRV_PCM_HW_PARAM_BUFFER_TIME

- SNDRV_PCM_HW_PARAM_BUFFER_SIZE

- SNDRV_PCM_HW_PARAM_BUFFER_BYTES

- SNDRV_PCM_HW_PARAM_TICK_TIME

rmask
> Configurable. This is evaluated at ioctl(2) with SNDRV_PCM_IOCTL_HW_REFINE only. Applications can select which mask/interval parameter can be changed by ALSA PCM core. For SNDRV_PCM_IOCTL_HW_PARAMS, this mask is ignored and all of parameters are going to be changed.

cmask
> Read-only. After returning from ioctl(2), buffer in user space for struct snd_pcm_hw_params includes result of each operation. This mask represents which mask/interval parameter is actually changed.

info
> Read-only. This represents hardware/driver capabilities as bit flags with SNDRV_PCM_INFO_XXX. Typically, applications execute ioctl(2) with SNDRV_PCM_IOCTL_HW_REFINE to retrieve this flag, then decide candidates of parameters and execute ioctl(2) with SNDRV_PCM_IOCTL_HW_PARAMS to configure PCM substream.

msbits
> Read-only. This value represents available bit width in MSB side of a PCM sample. When a parameter of SNDRV_PCM_HW_PARAM_SAMPLE_BITS was decided as a fixed number, this value is also calculated according to it. Else, zero. But this behaviour depends on implementations in driver side.

rate_num
> Read-only. This value represents numerator of sampling rate in fraction notation. Basically, when a parameter of SNDRV_PCM_HW_PARAM_RATE was decided as a single value, this value is also calculated according to it. Else, zero. But this behaviour depends on implementations in driver side.

rate_den
> Read-only. This value represents denominator of sampling rate in fraction notation. Basically, when a parameter of SNDRV_PCM_HW_PARAM_RATE was decided as a single value, this value is also calculated according to it. Else, zero. But this behaviour depends on implementations in driver side.

fifo_size
> Read-only. This value represents the size of FIFO in serial sound interface of hardware.

Basically, each driver can assigns a proper value to this parameter but some drivers in-
tentionally set zero with a care of hardware design or data transmission protocol.

ALSA PCM core handles buffer of struct snd_pcm_hw_params when applications execute
ioctl(2) with SNDRV_PCM_HW_REFINE or SNDRV_PCM_HW_PARAMS. Parameters in the
buffer are changed according to struct snd_pcm_hardware and rules of constraints in the run-
time. The structure describes capabilities of handled hardware. The rules describes dependen-
cies on which a parameter is decided according to several parameters. A rule has a callback
function, and drivers can register arbitrary functions to compute the target parameter. ALSA
PCM core registers some rules to the runtime as a default.

Each driver can join in the interaction as long as it prepared for two stuffs in a callback of struct
snd_pcm_ops.open.

1. In the callback, drivers are expected to change a member of struct snd_pcm_hardware
   type in the runtime, according to capacities of corresponding hardware.

2. In the same callback, drivers are also expected to register additional rules of constraints
   into the runtime when several parameters have dependencies due to hardware design.

The driver can refers to result of the interaction in a callback of struct snd_pcm_ops.hw_params,
however it should not change the content.

Tracepoints in this category are designed to trace changes of the mask/interval parameters.
When ALSA PCM core changes them, `hw_mask_param` or `hw_interval_param` event is probed
according to type of the changed parameter.

ALSA PCM core also has a pretty print format for each of the tracepoints. Below is an example
for `hw_mask_param`.

```
hw_mask_param: pcmC0D0p 001/023 FORMAT 000000000000000000000001000000044␣
↪000000000000000000000001000000044
```

Below is an example for `hw_interval_param`.

```
hw_interval_param: pcmC0D0p 000/023 BUFFER_SIZE 0 0 [0 4294967295] 0 1 [0␣
↪4294967295]
```

The first three fields are common. They represent name of ALSA PCM character device, rules of
constraint and name of the changed parameter, in order. The field for rules of constraint consists
of two sub-fields; index of applied rule and total number of rules added to the runtime. As an
exception, the index 000 means that the parameter is changed by ALSA PCM core, regardless
of the rules.

The rest of field represent state of the parameter before/after changing. These fields are dif-
ferent according to type of the parameter. For parameters of mask type, the fields represent
hexadecimal dump of content of the parameter. For parameters of interval type, the fields
represent values of each member of `empty`, `integer`, `openmin`, `min`, `max`, `openmax` in struct
snd_interval in this order.

2.6.2 Tracepoints in drivers

Some drivers have tracepoints for developers' convenience. For them, please refer to each documentation or implementation.

2.7 Proc Files of ALSA Drivers

Takashi Iwai <tiwai@suse.de>

2.7.1 General

ALSA has its own proc tree, /proc/asound. Many useful information are found in this tree. When you encounter a problem and need debugging, check the files listed in the following sections.

Each card has its subtree cardX, where X is from 0 to 7. The card-specific files are stored in the `card*` subdirectories.

2.7.2 Global Information

cards
    Shows the list of currently configured ALSA drivers, index, the id string, short and long descriptions.

version
    Shows the version string and compile date.

modules
    Lists the module of each card

devices
    Lists the ALSA native device mappings.

meminfo
    Shows the status of allocated pages via ALSA drivers. Appears only when `CONFIG_SND_DEBUG=y`.

hwdep
    Lists the currently available hwdep devices in format of `<card>-<device>: <name>`

pcm
    Lists the currently available PCM devices in format of `<card>-<device>: <id>: <name>` `: <sub-streams>`

timer
    Lists the currently available timer devices

oss/devices
    Lists the OSS device mappings.

oss/sndstat
    Provides the output compatible with /dev/sndstat. You can symlink this to /dev/sndstat.

2.7.3 Card Specific Files

The card-specific files are found in `/proc/asound/card*` directories. Some drivers (e.g. cmipci) have their own proc entries for the register dump, etc (e.g. `/proc/asound/card*/cmipci` shows the register dump). These files would be really helpful for debugging.

When PCM devices are available on this card, you can see directories like pcm0p or pcm1c. They hold the PCM information for each PCM stream. The number after `pcm` is the PCM device number from 0, and the last `p` or `c` means playback or capture direction. The files in this subtree is described later.

The status of MIDI I/O is found in `midi*` files. It shows the device name and the received/transmitted bytes through the MIDI device.

When the card is equipped with AC97 codecs, there are `codec97#*` subdirectories (described later).

When the OSS mixer emulation is enabled (and the module is loaded), oss_mixer file appears here, too. This shows the current mapping of OSS mixer elements to the ALSA control elements. You can change the mapping by writing to this device. Read OSS-Emulation.txt for details.

2.7.4 PCM Proc Files

card*/pcm*/info
> The general information of this PCM device: card #, device #, substreams, etc.

card*/pcm*/xrun_debug
> This file appears when `CONFIG_SND_DEBUG=y` and `CONFIG_SND_PCM_XRUN_DEBUG=y`. This shows the status of xrun (= buffer overrun/xrun) and invalid PCM position debug/check of ALSA PCM middle layer. It takes an integer value, can be changed by writing to this file, such as:

```
# echo 5 > /proc/asound/card0/pcm0p/xrun_debug
```

> The value consists of the following bit flags:
>
> - bit 0 = Enable XRUN/jiffies debug messages
> - bit 1 = Show stack trace at XRUN / jiffies check
> - bit 2 = Enable additional jiffies check
>
> When the bit 0 is set, the driver will show the messages to kernel log when an xrun is detected. The debug message is shown also when the invalid H/W pointer is detected at the update of periods (usually called from the interrupt handler).
>
> When the bit 1 is set, the driver will show the stack trace additionally. This may help the debugging.
>
> Since 2.6.30, this option can enable the hwptr check using jiffies. This detects spontaneous invalid pointer callback values, but can be lead to too much corrections for a (mostly buggy) hardware that doesn't give smooth pointer updates. This feature is enabled via the bit 2.

card*/pcm*/sub*/info
> The general information of this PCM sub-stream.

card*/pcm*/sub*/status

    The current status of this PCM sub-stream, elapsed time, H/W position, etc.

card*/pcm*/sub*/hw_params

    The hardware parameters set for this sub-stream.

card*/pcm*/sub*/sw_params

    The soft parameters set for this sub-stream.

card*/pcm*/sub*/prealloc

    The buffer pre-allocation information.

card*/pcm*/sub*/xrun_injection

    Triggers an XRUN to the running stream when any value is written to this proc file. Used for fault injection. This entry is write-only.

2.7.5 AC97 Codec Information

card*/codec97#*/ac97#?-?

    Shows the general information of this AC97 codec chip, such as name, capabilities, set up.

card*/codec97#0/ac97#?-?+regs

    Shows the AC97 register dump. Useful for debugging.

    When CONFIG_SND_DEBUG is enabled, you can write to this file for changing an AC97 register directly. Pass two hex numbers. For example,

```
# echo 02 9f1f > /proc/asound/card0/codec97#0/ac97#0-0+regs
```

2.7.6 USB Audio Streams

card*/stream*

    Shows the assignment and the current status of each audio stream of the given card. This information is very useful for debugging.

2.7.7 HD-Audio Codecs

card*/codec#*

    Shows the general codec information and the attribute of each widget node.

card*/eld#*

    Available for HDMI or DisplayPort interfaces. Shows ELD(EDID Like Data) info retrieved from the attached HDMI sink, and describes its audio capabilities and configurations.

    Some ELD fields may be modified by doing `echo name hex_value > eld#*`. Only do this if you are sure the HDMI sink provided value is wrong. And if that makes your HDMI audio work, please report to us so that we can fix it in future kernel releases.

2.7.8 Sequencer Information

seq/drivers
> Lists the currently available ALSA sequencer drivers.

seq/clients
> Shows the list of currently available sequencer clients and ports. The connection status and the running status are shown in this file, too.

seq/queues
> Lists the currently allocated/running sequencer queues.

seq/timer
> Lists the currently allocated/running sequencer timers.

seq/oss
> Lists the OSS-compatible sequencer stuffs.

2.7.9 Help For Debugging?

When the problem is related with PCM, first try to turn on xrun_debug mode. This will give you the kernel messages when and where xrun happened.

If it's really a bug, report it with the following information:

- the name of the driver/card, show in `/proc/asound/cards`
- the register dump, if available (e.g. `card*/cmipci`)

when it's a PCM problem,

- set-up of PCM, shown in hw_parms, sw_params, and status in the PCM sub-stream directory

when it's a mixer problem,

- AC97 proc files, `codec97#*/*` files

for USB audio/midi,

- output of `lsusb -v`
- `stream*` files in card directory

The ALSA bug-tracking system is found at: https://bugtrack.alsa-project.org/alsa-bug/

2.8 Notes on Power-Saving Mode

AC97 and HD-audio drivers have the automatic power-saving mode. This feature is enabled via Kconfig `CONFIG_SND_AC97_POWER_SAVE` and `CONFIG_SND_HDA_POWER_SAVE` options, respectively.

With the automatic power-saving, the driver turns off the codec power appropriately when no operation is required. When no applications use the device and/or no analog loopback is set, the power disablement is done fully or partially. It'll save a certain power consumption, thus good for laptops (even for desktops).

The time-out for automatic power-off can be specified via `power_save` module option of snd-ac97-codec and snd-hda-intel modules. Specify the time-out value in seconds. 0

---

means to disable the automatic power-saving. The default value of timeout is given via `CONFIG_SND_AC97_POWER_SAVE_DEFAULT` and `CONFIG_SND_HDA_POWER_SAVE_DEFAULT` Kconfig options. Setting this to 1 (the minimum value) isn't recommended because many applications try to reopen the device frequently. 10 would be a good choice for normal operations.

The `power_save` option is exported as writable. This means you can adjust the value via sysfs on the fly. For example, to turn on the automatic power-save mode with 10 seconds, write to `/sys/modules/snd_ac97_codec/parameters/power_save` (usually as root):

```
# echo 10 > /sys/modules/snd_ac97_codec/parameters/power_save
```

Note that you might hear click noise/pop when changing the power state. Also, it often takes certain time to wake up from the power-down to the active state. These are often hardly to fix, so don't report extra bug reports unless you have a fix patch ;-)

For HD-audio interface, there is another module option, power_save_controller. This enables/disables the power-save mode of the controller side. Setting this on may reduce a bit more power consumption, but might result in longer wake-up time and click noise. Try to turn it off when you experience such a thing too often.

2.9 Notes on Kernel OSS-Emulation

Jan. 22, 2004 Takashi Iwai <tiwai@suse.de>

2.9.1 Modules

ALSA provides a powerful OSS emulation on the kernel. The OSS emulation for PCM, mixer and sequencer devices is implemented as add-on kernel modules, snd-pcm-oss, snd-mixer-oss and snd-seq-oss. When you need to access the OSS PCM, mixer or sequencer devices, the corresponding module has to be loaded.

These modules are loaded automatically when the corresponding service is called. The alias is defined `sound-service-x-y`, where x and y are the card number and the minor unit number. Usually you don't have to define these aliases by yourself.

Only necessary step for auto-loading of OSS modules is to define the card alias in `/etc/modprobe.d/alsa.conf`, such as:

```
alias sound-slot-0 snd-emu10k1
```

As the second card, define `sound-slot-1` as well. Note that you can't use the aliased name as the target name (i.e. `alias sound-slot-0 snd-card-0` doesn't work any more like the old modutils).

The currently available OSS configuration is shown in /proc/asound/oss/sndstat. This shows in the same syntax of /dev/sndstat, which is available on the commercial OSS driver. On ALSA, you can symlink /dev/sndstat to this proc file.

Please note that the devices listed in this proc file appear only after the corresponding OSS-emulation module is loaded. Don't worry even if "NOT ENABLED IN CONFIG" is shown in it.

2.9.2 Device Mapping

ALSA supports the following OSS device files:

```
PCM:
        /dev/dspX
        /dev/adspX

Mixer:
        /dev/mixerX

MIDI:
        /dev/midi0X
        /dev/amidi0X

Sequencer:
        /dev/sequencer
        /dev/sequencer2 (aka /dev/music)
```

where X is the card number from 0 to 7.

(NOTE: Some distributions have the device files like /dev/midi0 and /dev/midi1. They are NOT for OSS but for tclmidi, which is a totally different thing.)

Unlike the real OSS, ALSA cannot use the device files more than the assigned ones. For example, the first card cannot use /dev/dsp1 or /dev/dsp2, but only /dev/dsp0 and /dev/adsp0.

As seen above, PCM and MIDI may have two devices. Usually, the first PCM device (`hw:0,0` in ALSA) is mapped to /dev/dsp and the secondary device (`hw:0,1`) to /dev/adsp (if available). For MIDI, /dev/midi and /dev/amidi, respectively.

You can change this device mapping via the module options of snd-pcm-oss and snd-rawmidi. In the case of PCM, the following options are available for snd-pcm-oss:

dsp_map
    PCM device number assigned to /dev/dspX (default = 0)

adsp_map
    PCM device number assigned to /dev/adspX (default = 1)

For example, to map the third PCM device (`hw:0,2`) to /dev/adsp0, define like this:

```
options snd-pcm-oss adsp_map=2
```

The options take arrays. For configuring the second card, specify two entries separated by comma. For example, to map the third PCM device on the second card to /dev/adsp1, define like below:

```
options snd-pcm-oss adsp_map=0,2
```

To change the mapping of MIDI devices, the following options are available for snd-rawmidi:

midi_map
    MIDI device number assigned to /dev/midi0X (default = 0)

amidi_map
    MIDI device number assigned to /dev/amidi0X (default = 1)

For example, to assign the third MIDI device on the first card to /dev/midi00, define as follows:

```
options snd-rawmidi midi_map=2
```

2.9.3 PCM Mode

As default, ALSA emulates the OSS PCM with so-called plugin layer, i.e. tries to convert the sample format, rate or channels automatically when the card doesn't support it natively. This will lead to some problems for some applications like quake or wine, especially if they use the card only in the MMAP mode.

In such a case, you can change the behavior of PCM per application by writing a command to the proc file. There is a proc file for each PCM stream, `/proc/asound/cardX/pcmY[cp]/oss`, where X is the card number (zero-based), Y the PCM device number (zero-based), and `p` is for playback and `c` for capture, respectively. Note that this proc file exists only after snd-pcm-oss module is loaded.

The command sequence has the following syntax:

```
app_name fragments fragment_size [options]
```

`app_name` is the name of application with (higher priority) or without path. `fragments` specifies the number of fragments or zero if no specific number is given. `fragment_size` is the size of fragment in bytes or zero if not given. `options` is the optional parameters. The following options are available:

disable
    the application tries to open a pcm device for this channel but does not want to use it.

direct
    don't use plugins

block
    force block open mode

non-block
    force non-block open mode

partial-frag
    write also partial fragments (affects playback only)

no-silence
    do not fill silence ahead to avoid clicks

The `disable` option is useful when one stream direction (playback or capture) is not handled correctly by the application although the hardware itself does support both directions. The `direct` option is used, as mentioned above, to bypass the automatic conversion and useful for MMAP-applications. For example, to playback the first PCM device without plugins for quake, send a command via echo like the following:

```
% echo "quake 0 0 direct" > /proc/asound/card0/pcm0p/oss
```

While quake wants only playback, you may append the second command to notify driver that only this direction is about to be allocated:

```
% echo "quake 0 0 disable" > /proc/asound/card0/pcm0c/oss
```

The permission of proc files depend on the module options of snd. As default it's set as root, so you'll likely need to be superuser for sending the command above.

The block and non-block options are used to change the behavior of opening the device file.

As default, ALSA behaves as original OSS drivers, i.e. does not block the file when it's busy. The -EBUSY error is returned in this case.

This blocking behavior can be changed globally via nonblock_open module option of snd-pcm-oss. For using the blocking mode as default for OSS devices, define like the following:

```
options snd-pcm-oss nonblock_open=0
```

The `partial-frag` and `no-silence` commands have been added recently. Both commands are for optimization use only. The former command specifies to invoke the write transfer only when the whole fragment is filled. The latter stops writing the silence data ahead automatically. Both are disabled as default.

You can check the currently defined configuration by reading the proc file. The read image can be sent to the proc file again, hence you can save the current configuration

```
% cat /proc/asound/card0/pcm0p/oss > /somewhere/oss-cfg
```

and restore it like

```
% cat /somewhere/oss-cfg > /proc/asound/card0/pcm0p/oss
```

Also, for clearing all the current configuration, send `erase` command as below:

```
% echo "erase" > /proc/asound/card0/pcm0p/oss
```

2.9.4 Mixer Elements

Since ALSA has completely different mixer interface, the emulation of OSS mixer is relatively complicated. ALSA builds up a mixer element from several different ALSA (mixer) controls based on the name string. For example, the volume element SOUND_MIXER_PCM is composed from "PCM Playback Volume" and "PCM Playback Switch" controls for the playback direction and from "PCM Capture Volume" and "PCM Capture Switch" for the capture directory (if exists). When the PCM volume of OSS is changed, all the volume and switch controls above are adjusted automatically.

As default, ALSA uses the following control for OSS volumes:

OSS volume	ALSA control	Index
SOUND_MIXER_VOLUME	Master	0
SOUND_MIXER_BASS	Tone Control - Bass	0
SOUND_MIXER_TREBLE	Tone Control - Treble	0
SOUND_MIXER_SYNTH	Synth	0
SOUND_MIXER_PCM	PCM	0
SOUND_MIXER_SPEAKER	PC Speaker	0
SOUND_MIXER_LINE	Line	0
SOUND_MIXER_MIC	Mic	0
SOUND_MIXER_CD	CD	0
SOUND_MIXER_IMIX	Monitor Mix	0
SOUND_MIXER_ALTPCM	PCM	1
SOUND_MIXER_RECLEV	(not assigned)	
SOUND_MIXER_IGAIN	Capture	0
SOUND_MIXER_OGAIN	Playback	0
SOUND_MIXER_LINE1	Aux	0
SOUND_MIXER_LINE2	Aux	1
SOUND_MIXER_LINE3	Aux	2
SOUND_MIXER_DIGITAL1	Digital	0
SOUND_MIXER_DIGITAL2	Digital	1
SOUND_MIXER_DIGITAL3	Digital	2
SOUND_MIXER_PHONEIN	Phone	0
SOUND_MIXER_PHONEOUT	Phone	1
SOUND_MIXER_VIDEO	Video	0
SOUND_MIXER_RADIO	Radio	0
SOUND_MIXER_MONITOR	Monitor	0

The second column is the base-string of the corresponding ALSA control. In fact, the controls with `XXX [Playback|Capture] [Volume|Switch]` will be checked in addition.

The current assignment of these mixer elements is listed in the proc file, /proc/asound/cardX/oss_mixer, which will be like the following

```
VOLUME "Master" 0
BASS "" 0
TREBLE "" 0
SYNTH "" 0
PCM "PCM" 0
...
```

where the first column is the OSS volume element, the second column the base-string of the corresponding ALSA control, and the third the control index. When the string is empty, it means that the corresponding OSS control is not available.

For changing the assignment, you can write the configuration to this proc file. For example, to map "Wave Playback" to the PCM volume, send the command like the following:

```
% echo 'VOLUME "Wave Playback" 0' > /proc/asound/card0/oss_mixer
```

The command is exactly as same as listed in the proc file. You can change one or more elements, one volume per line. In the last example, both "Wave Playback Volume" and "Wave Playback

Switch" will be affected when PCM volume is changed.

Like the case of PCM proc file, the permission of proc files depend on the module options of snd. you'll likely need to be superuser for sending the command above.

As well as in the case of PCM proc file, you can save and restore the current mixer configuration by reading and writing the whole file image.

2.9.5 Duplex Streams

Note that when attempting to use a single device file for playback and capture, the OSS API provides no way to set the format, sample rate or number of channels different in each direction. Thus

```
io_handle = open("device", O_RDWR)
```

will only function correctly if the values are the same in each direction.

To use different values in the two directions, use both

```
input_handle = open("device", O_RDONLY)
output_handle = open("device", O_WRONLY)
```

and set the values for the corresponding handle.

2.9.6 Unsupported Features

MMAP on ICE1712 driver

ICE1712 supports only the unconventional format, interleaved 10-channels 24bit (packed in 32bit) format. Therefore you cannot mmap the buffer as the conventional (mono or 2-channels, 8 or 16bit) format on OSS.

2.10 OSS Sequencer Emulation on ALSA

Copyright (c) 1998,1999 by Takashi Iwai

ver.0.1.8; Nov. 16, 1999

2.10.1 Description

This directory contains the OSS sequencer emulation driver on ALSA. Note that this program is still in the development state.

What this does - it provides the emulation of the OSS sequencer, access via /dev/sequencer and /dev/music devices. The most of applications using OSS can run if the appropriate ALSA sequencer is prepared.

The following features are emulated by this driver:

- Normal sequencer and MIDI events:

They are converted to the ALSA sequencer events, and sent to the corresponding port.

- Timer events:

    The timer is not selectable by ioctl. The control rate is fixed to 100 regardless of HZ. That is, even on Alpha system, a tick is always 1/100 second. The base rate and tempo can be changed in `/dev/music`.

- Patch loading:

    It purely depends on the synth drivers whether it's supported since the patch loading is realized by callback to the synth driver.

- I/O controls:

    Most of controls are accepted. Some controls are dependent on the synth driver, as well as even on original OSS.

Furthermore, you can find the following advanced features:

- Better queue mechanism:

    The events are queued before processing them.

- Multiple applications:

    You can run two or more applications simultaneously (even for OSS sequencer)! However, each MIDI device is exclusive - that is, if a MIDI device is opened once by some application, other applications can't use it. No such a restriction in synth devices.

- Real-time event processing:

    The events can be processed in real time without using out of bound ioctl. To switch to real-time mode, send ABSTIME 0 event. The followed events will be processed in real-time without queued. To switch off the real-time mode, send RELTIME 0 event.

- `/proc` interface:

    The status of applications and devices can be shown via `/proc/asound/seq/oss` at any time. In the later version, configuration will be changed via `/proc` interface, too.

2.10.2 Installation

Run configure script with both sequencer support (`--with-sequencer=yes`) and OSS emulation (`--with-oss=yes`) options. A module `snd-seq-oss.o` will be created. If the synth module of your sound card supports for OSS emulation (so far, only Emu8000 driver), this module will be loaded automatically. Otherwise, you need to load this module manually.

At beginning, this module probes all the MIDI ports which have been already connected to the sequencer. Once after that, the creation and deletion of ports are watched by announcement mechanism of ALSA sequencer.

The available synth and MIDI devices can be found in proc interface. Run `cat /proc/asound/seq/oss`, and check the devices. For example, if you use an AWE64 card, you'll see like the following:

```
OSS sequencer emulation version 0.1.8
ALSA client number 63
ALSA receiver port 0

Number of applications: 0

Number of synth devices: 1
synth 0: [EMU8000]
  type 0x1 : subtype 0x20 : voices 32
  capabilities : ioctl enabled / load_patch enabled

Number of MIDI devices: 3
midi 0: [Emu8000 Port-0] ALSA port 65:0
  capability write / opened none

midi 1: [Emu8000 Port-1] ALSA port 65:1
  capability write / opened none

midi 2: [0: MPU-401 (UART)] ALSA port 64:0
  capability read/write / opened none
```

Note that the device number may be different from the information of `/proc/asound/oss-devices` or ones of the original OSS driver. Use the device number listed in `/proc/asound/seq/oss` to play via OSS sequencer emulation.

2.10.3 Using Synthesizer Devices

Run your favorite program. I've tested playmidi-2.4, awemidi-0.4.3, gmod-3.1 and xmp-1.1.5. You can load samples via `/dev/sequencer` like sfxload, too.

If the lowlevel driver supports multiple access to synth devices (like Emu8000 driver), two or more applications are allowed to run at the same time.

2.10.4 Using MIDI Devices

So far, only MIDI output was tested. MIDI input was not checked at all, but hopefully it will work. Use the device number listed in `/proc/asound/seq/oss`. Be aware that these numbers are mostly different from the list in `/proc/asound/oss-devices`.

2.10.5 Module Options

The following module options are available:

maxqlen
  specifies the maximum read/write queue length. This queue is private for OSS sequencer, so that it is independent from the queue length of ALSA sequencer. Default value is 1024.

seq_oss_debug
  specifies the debug level and accepts zero (= no debug message) or positive integer. Default value is 0.

2.10.6 Queue Mechanism

OSS sequencer emulation uses an ALSA priority queue. The events from `/dev/sequencer` are processed and put onto the queue specified by module option.

All the events from `/dev/sequencer` are parsed at beginning. The timing events are also parsed at this moment, so that the events may be processed in real-time. Sending an event ABSTIME 0 switches the operation mode to real-time mode, and sending an event RELTIME 0 switches it off. In the real-time mode, all events are dispatched immediately.

The queued events are dispatched to the corresponding ALSA sequencer ports after scheduled time by ALSA sequencer dispatcher.

If the write-queue is full, the application sleeps until a certain amount (as default one half) becomes empty in blocking mode. The synchronization to write timing was implemented, too.

The input from MIDI devices or echo-back events are stored on read FIFO queue. If application reads `/dev/sequencer` in blocking mode, the process will be awaked.

2.10.7 Interface to Synthesizer Device

Registration

To register an OSS synthesizer device, use snd_seq_oss_synth_register() function:

```
int snd_seq_oss_synth_register(char *name, int type, int subtype, int nvoices,
        snd_seq_oss_callback_t *oper, void *private_data)
```

The arguments `name`, `type`, `subtype` and `nvoices` are used for making the appropriate synth_info structure for ioctl. The return value is an index number of this device. This index must be remembered for unregister. If registration is failed, -errno will be returned.

To release this device, call snd_seq_oss_synth_unregister() function:

```
int snd_seq_oss_synth_unregister(int index)
```

where the `index` is the index number returned by register function.

Callbacks

OSS synthesizer devices have capability for sample downloading and ioctls like sample reset. In OSS emulation, these special features are realized by using callbacks. The registration argument oper is used to specify these callbacks. The following callback functions must be defined:

```
snd_seq_oss_callback_t:
 int (*open)(snd_seq_oss_arg_t *p, void *closure);
 int (*close)(snd_seq_oss_arg_t *p);
 int (*ioctl)(snd_seq_oss_arg_t *p, unsigned int cmd, unsigned long arg);
 int (*load_patch)(snd_seq_oss_arg_t *p, int format, const char *buf, int offs,
↪ int count);
 int (*reset)(snd_seq_oss_arg_t *p);
```

Except for `open` and `close` callbacks, they are allowed to be NULL.

Each callback function takes the argument type `snd_seq_oss_arg_t` as the first argument.

```
struct snd_seq_oss_arg_t {
    int app_index;
    int file_mode;
    int seq_mode;
    snd_seq_addr_t addr;
    void *private_data;
    int event_passing;
};
```

The first three fields, `app_index`, `file_mode` and `seq_mode` are initialized by OSS sequencer. The `app_index` is the application index which is unique to each application opening OSS sequencer. The `file_mode` is bit-flags indicating the file operation mode. See `seq_oss.h` for its meaning. The `seq_mode` is sequencer operation mode. In the current version, only `SND_OSSSEQ_MODE_SYNTH` is used.

The next two fields, `addr` and `private_data`, must be filled by the synth driver at open callback. The `addr` contains the address of ALSA sequencer port which is assigned to this device. If the driver allocates memory for `private_data`, it must be released in close callback by itself.

The last field, `event_passing`, indicates how to translate note-on / off events. In `PROCESS_EVENTS` mode, the note 255 is regarded as velocity change, and key pressure event is passed to the port. In `PASS_EVENTS` mode, all note on/off events are passed to the port without modified. `PROCESS_KEYPRESS` mode checks the note above 128 and regards it as key pressure event (mainly for Emu8000 driver).

Open Callback

The `open` is called at each time this device is opened by an application using OSS sequencer. This must not be NULL. Typically, the open callback does the following procedure:

1. Allocate private data record.

2. Create an ALSA sequencer port.

3. Set the new port address on `arg->addr`.

4. Set the private data record pointer on `arg->private_data`.

Note that the type bit-flags in port_info of this synth port must NOT contain `TYPE_MIDI_GENERIC` bit. Instead, `TYPE_SPECIFIC` should be used. Also, `CAP_SUBSCRIPTION` bit should NOT be included, too. This is necessary to tell it from other normal MIDI devices. If the open procedure succeeded, return zero. Otherwise, return -errno.

Ioctl Callback

The `ioctl` callback is called when the sequencer receives device-specific ioctls. The following two ioctls should be processed by this callback:

IOCTL_SEQ_RESET_SAMPLES
    reset all samples on memory -- return 0

IOCTL_SYNTH_MEMAVL
    return the available memory size

FM_4OP_ENABLE
    can be ignored usually

The other ioctls are processed inside the sequencer without passing to the lowlevel driver.

Load_Patch Callback

The `load_patch` callback is used for sample-downloading. This callback must read the data on user-space and transfer to each device. Return 0 if succeeded, and -errno if failed. The format argument is the patch key in patch_info record. The buf is user-space pointer where patch_info record is stored. The offs can be ignored. The count is total data size of this sample data.

Close Callback

The `close` callback is called when this device is closed by the application. If any private data was allocated in open callback, it must be released in the close callback. The deletion of ALSA port should be done here, too. This callback must not be NULL.

Reset Callback

The `reset` callback is called when sequencer device is reset or closed by applications. The callback should turn off the sounds on the relevant port immediately, and initialize the status of the port. If this callback is undefined, OSS seq sends a `HEARTBEAT` event to the port.

2.10.8 Events

Most of the events are processed by sequencer and translated to the adequate ALSA sequencer events, so that each synth device can receive by input_event callback of ALSA sequencer port. The following ALSA events should be implemented by the driver:

ALSA event	Original OSS events
NOTEON	SEQ_NOTEON, MIDI_NOTEON
NOTE	SEQ_NOTEOFF, MIDI_NOTEOFF
KEYPRESS	MIDI_KEY_PRESSURE
CHANPRESS	SEQ_AFTERTOUCH, MIDI_CHN_PRESSURE
PGMCHANGE	SEQ_PGMCHANGE, MIDI_PGM_CHANGE
PITCHBEND	SEQ_CONTROLLER(CTRL_PITCH_BENDER), MIDI_PITCH_BEND
CONTROLLER	MIDI_CTL_CHANGE, SEQ_BALANCE (with CTL_PAN)
CONTROL14	SEQ_CONTROLLER
REGPARAM	SEQ_CONTROLLER(CTRL_PITCH_BENDER_RANGE)
SYSEX	SEQ_SYSEX

The most of these behavior can be realized by MIDI emulation driver included in the Emu8000 lowlevel driver. In the future release, this module will be independent.

Some OSS events (`SEQ_PRIVATE` and `SEQ_VOLUME` events) are passed as event type SND_SEQ_OSS_PRIVATE. The OSS sequencer passes these event 8 byte packets without any modification. The lowlevel driver should process these events appropriately.

2.10.9 Interface to MIDI Device

Since the OSS emulation probes the creation and deletion of ALSA MIDI sequencer ports automatically by receiving announcement from ALSA sequencer, the MIDI devices don't need to be registered explicitly like synth devices. However, the MIDI port_info registered to ALSA sequencer must include a group name `SND_SEQ_GROUP_DEVICE` and a capability-bit `CAP_READ` or `CAP_WRITE`. Also, subscription capabilities, `CAP_SUBS_READ` or `CAP_SUBS_WRITE`, must be defined, too. If these conditions are not satisfied, the port is not registered as OSS sequencer MIDI device.

The events via MIDI devices are parsed in OSS sequencer and converted to the corresponding ALSA sequencer events. The input from MIDI sequencer is also converted to MIDI byte events by OSS sequencer. This works just a reverse way of seq_midi module.

2.10.10 Known Problems / TODO's

- Patch loading via ALSA instrument layer is not implemented yet.

2.11 ALSA Jack Software Injection

2.11.1 Simple Introduction On Jack Injection

Here jack injection means users could inject plugin or plugout events to the audio jacks through debugfs interface, it is helpful to validate ALSA userspace changes. For example, we change the audio profile switching code in the pulseaudio, and we want to verify if the change works as expected and if the change introduce the regression, in this case, we could inject plugin or plugout events to an audio jack or to some audio jacks, we don't need to physically access the machine and plug/unplug physical devices to the audio jack.

In this design, an audio jack doesn't equal to a physical audio jack. Sometimes a physical audio jack contains multi functions, and the ALSA driver creates multi `jack_kctl` for a `snd_jack`, here the `snd_jack` represents a physical audio jack and the `jack_kctl` represents a function, for example a physical jack has two functions: headphone and mic_in, the ALSA ASoC driver will build 2 `jack_kctl` for this jack. The jack injection is implemented based on the `jack_kctl` instead of `snd_jack`.

To inject events to audio jacks, we need to enable the jack injection via `sw_inject_enable` first, once it is enabled, this jack will not change the state by hardware events anymore, we could inject plugin or plugout events via `jackin_inject` and check the jack state via `status`, after we finish our test, we need to disable the jack injection via `sw_inject_enable` too, once it is disabled, the jack state will be restored according to the last reported hardware events and will change by future hardware events.

2.11.2 The Layout of Jack Injection Interface

If users enable the SND_JACK_INJECTION_DEBUG in the kernel, the audio jack injection interface will be created as below:

```
$debugfs_mount_dir/sound
|-- card0
|-- |-- HDMI_DP_pcm_10_Jack
|-- |-- |-- jackin_inject
|-- |-- |-- kctl_id
|-- |-- |-- mask_bits
|-- |-- |-- status
|-- |-- |-- sw_inject_enable
|-- |-- |-- type
...
|-- |-- HDMI_DP_pcm_9_Jack
|--     |-- jackin_inject
|--     |-- kctl_id
|--     |-- mask_bits
|--     |-- status
|--     |-- sw_inject_enable
|--     |-- type
|-- card1
    |-- HDMI_DP_pcm_5_Jack
    |-- |-- jackin_inject
    |-- |-- kctl_id
    |-- |-- mask_bits
    |-- |-- status
    |-- |-- sw_inject_enable
    |-- |-- type
    ...
    |-- Headphone_Jack
    |-- |-- jackin_inject
    |-- |-- kctl_id
    |-- |-- mask_bits
    |-- |-- status
    |-- |-- sw_inject_enable
```

```
    |-- |-- type
    |-- Headset_Mic_Jack
        |-- jackin_inject
        |-- kctl_id
        |-- mask_bits
        |-- status
        |-- sw_inject_enable
        |-- type
```

2.11.3 The Explanation Of The Nodes

kctl_id
>   read-only, get jack_kctl->kctl's id

```
sound/card1/Headphone_Jack# cat kctl_id
Headphone Jack
```

mask_bits
>   read-only, get jack_kctl's supported events mask_bits

```
sound/card1/Headphone_Jack# cat mask_bits
0x0001 HEADPHONE(0x0001)
```

status
>   read-only, get jack_kctl's current status

- headphone unplugged:

```
sound/card1/Headphone_Jack# cat status
Unplugged
```

- headphone plugged:

```
sound/card1/Headphone_Jack# cat status
Plugged
```

type
>   read-only, get snd_jack's supported events from type (all supported events on the physical
>   audio jack)

```
sound/card1/Headphone_Jack# cat type
0x7803 HEADPHONE(0x0001) MICROPHONE(0x0002) BTN_3(0x0800) BTN_2(0x1000)␣
 →BTN_1(0x2000) BTN_0(0x4000)
```

sw_inject_enable
>   read-write, enable or disable injection

- injection disabled:

```
sound/card1/Headphone_Jack# cat sw_inject_enable
Jack: Headphone Jack                 Inject Enabled: 0
```

- injection enabled:

```
sound/card1/Headphone_Jack# cat sw_inject_enable
Jack: Headphone Jack            Inject Enabled: 1
```

- to enable jack injection:

```
sound/card1/Headphone_Jack# echo 1 > sw_inject_enable
```

- to disable jack injection:

```
sound/card1/Headphone_Jack# echo 0 > sw_inject_enable
```

jackin_inject
> write-only, inject plugin or plugout

- to inject plugin:

```
sound/card1/Headphone_Jack# echo 1 > jackin_inject
```

- to inject plugout:

```
sound/card1/Headphone_Jack# echo 0 > jackin_inject
```

2.12 MIDI 2.0 on Linux

2.12.1 General

MIDI 2.0 is an extended protocol for providing higher resolutions and more fine controls over the legacy MIDI 1.0. The fundamental changes introduced for supporting MIDI 2.0 are:

- Support of Universal MIDI Packet (UMP)
- Support of MIDI 2.0 protocol messages
- Transparent conversions between UMP and legacy MIDI 1.0 byte stream
- MIDI-CI for property and profile configurations

UMP is a new container format to hold all MIDI protocol 1.0 and MIDI 2.0 protocol messages. Unlike the former byte stream, it's 32bit aligned, and each message can be put in a single packet. UMP can send the events up to 16 "UMP Groups", where each UMP Group contain up to 16 MIDI channels.

MIDI 2.0 protocol is an extended protocol to achieve the higher resolution and more controls over the old MIDI 1.0 protocol.

MIDI-CI is a high-level protocol that can talk with the MIDI device for the flexible profiles and configurations. It's represented in the form of special SysEx.

For Linux implementations, the kernel supports the UMP transport and the encoding/decoding of MIDI protocols on UMP, while MIDI-CI is supported in user-space over the standard SysEx.

As of this writing, only USB MIDI device supports the UMP and Linux 2.0 natively. The UMP support itself is pretty generic, hence it could be used by other transport layers, although it could be implemented differently (e.g. as a ALSA sequencer client), too.

The access to UMP devices are provided in two ways: the access via rawmidi device and the access via ALSA sequencer API.

ALSA sequencer API was extended to allow the payload of UMP packets. It's allowed to connect freely between MIDI 1.0 and MIDI 2.0 sequencer clients, and the events are converted transparently.

2.12.2 Kernel Configuration

The following new configs are added for supporting MIDI 2.0: *CONFIG_SND_UMP*, *CONFIG_SND_UMP_LEGACY_RAWMIDI*, *CONFIG_SND_SEQ_UMP*, *CONFIG_SND_SEQ_UMP_CLIENT*, and *CONFIG_SND_USB_AUDIO_MIDI_V2*. The first visible one is *CONFIG_SND_USB_AUDIO_MIDI_V2*, and when you choose it (to set =*y*), the core support for UMP (*CONFIG_SND_UMP*) and the sequencer binding (*CONFIG_SND_SEQ_UMP_CLIENT*) will be automatically selected.

Additionally, *CONFIG_SND_UMP_LEGACY_RAWMIDI=y* will enable the support for the legacy raw MIDI device for UMP Endpoints.

2.12.3 Rawmidi Device with USB MIDI 2.0

When a device supports MIDI 2.0, the USB-audio driver probes and uses the MIDI 2.0 interface (that is found always at the altset 1) as default instead of the MIDI 1.0 interface (at altset 0). You can switch back to the binding with the old MIDI 1.0 interface by passing *midi2_enable=0* option to snd-usb-audio driver module, too.

The USB audio driver tries to query the UMP Endpoint and UMP Function Block information that are provided since UMP v1.1, and builds up the topology based on those information. When the device is older and doesn't respond to the new UMP inquiries, the driver falls back and builds the topology based on Group Terminal Block (GTB) information from the USB descriptor. Some device might be screwed up by the unexpected UMP command; in such a case, pass *midi2_ump_probe=0* option to snd-usb-audio driver for skipping the UMP v1.1 inquiries.

When the MIDI 2.0 device is probed, the kernel creates a rawmidi device for each UMP Endpoint of the device. Its device name is */dev/snd/umpC*D** and different from the standard rawmidi device name */dev/snd/midiC*D** for MIDI 1.0, in order to avoid confusing the legacy applications accessing mistakenly to UMP devices.

You can read and write UMP packet data directly from/to this UMP rawmidi device. For example, reading via *hexdump* like below will show the incoming UMP packets of the card 0 device 0 in the hex format:

```
% hexdump -C /dev/snd/umpC0D0
00000000  01 07 b0 20 00 07 b0 20  64 3c 90 20 64 3c 80 20  |... ... d<. d<. |
```

Unlike the MIDI 1.0 byte stream, UMP is a 32bit packet, and the size for reading or writing the device is also aligned to 32bit (which is 4 bytes).

The 32-bit words in the UMP packet payload are always in CPU native endianness. Transport drivers are responsible to convert UMP words from / to system endianness to required transport endianness / byte order.

When *CONFIG_SND_UMP_LEGACY_RAWMIDI* is set, the driver creates another standard raw MIDI device additionally as */dev/snd/midiC*D**. This contains 16 substreams, and each sub-

stream corresponds to a (0-based) UMP Group. Legacy applications can access to the specified group via each substream in MIDI 1.0 byte stream format. With the ALSA rawmidi API, you can open the arbitrary substream, while just opening */dev/snd/midiC*D** will end up with opening the first substream.

Each UMP Endpoint can provide the additional information, constructed from the information inquired via UMP 1.1 Stream messages or USB MIDI 2.0 descriptors. And a UMP Endpoint may contain one or more UMP Blocks, where UMP Block is an abstraction introduced in the ALSA UMP implementations to represent the associations among UMP Groups. UMP Block corresponds to Function Block in UMP 1.1 specification. When UMP 1.1 Function Block information isn't available, it's filled partially from Group Terminal Block (GTB) as defined in USB MIDI 2.0 specifications.

The information of UMP Endpoints and UMP Blocks are found in the proc file */proc/asound/card*/midi**. For example:

```
% cat /proc/asound/card1/midi0
ProtoZOA MIDI

Type: UMP
EP Name: ProtoZOA
EP Product ID: ABCD12345678
UMP Version: 0x0000
Protocol Caps: 0x00000100
Protocol: 0x00000100
Num Blocks: 3

Block 0 (ProtoZOA Main)
  Direction: bidirection
  Active: Yes
  Groups: 1-1
  Is MIDI1: No

Block 1 (ProtoZOA Ext IN)
  Direction: output
  Active: Yes
  Groups: 2-2
  Is MIDI1: Yes (Low Speed)
....
```

Note that *Groups* field shown in the proc file above indicates the 1-based UMP Group numbers (from-to).

Those additional UMP Endpoint and UMP Block information can be obtained via the new ioctls *SNDRV_UMP_IOCTL_ENDPOINT_INFO* and *SNDRV_UMP_IOCTL_BLOCK_INFO*, respectively.

The rawmidi name and the UMP Endpoint name are usually identical, and in the case of USB MIDI, it's taken from *iInterface* of the corresponding USB MIDI interface descriptor. If it's not provided, it's copied from *iProduct* of the USB device descriptor as a fallback.

The Endpoint Product ID is a string field and supposed to be unique. It's copied from *iSerialNumber* of the device for USB MIDI.

The protocol capabilities and the actual protocol bits are defined in *asound.h*.

2.12.4 ALSA Sequencer with USB MIDI 2.0

In addition to the rawmidi interfaces, ALSA sequencer interface supports the new UMP MIDI 2.0 device, too. Now, each ALSA sequencer client may set its MIDI version (0, 1 or 2) to declare itself being either the legacy, UMP MIDI 1.0 or UMP MIDI 2.0 device, respectively. The first, legacy client is the one that sends/receives the old sequencer event as was. Meanwhile, UMP MIDI 1.0 and 2.0 clients send and receive in the extended event record for UMP. The MIDI version is seen in the new *midi_version* field of *snd_seq_client_info*.

A UMP packet can be sent/received in a sequencer event embedded by specifying the new event flag bit *SNDRV_SEQ_EVENT_UMP*. When this flag is set, the event has 16 byte (128 bit) data payload for holding the UMP packet. Without the *SNDRV_SEQ_EVENT_UMP* bit flag, the event is treated as a legacy event as it was (with max 12 byte data payload).

With *SNDRV_SEQ_EVENT_UMP* flag set, the type field of a UMP sequencer event is ignored (but it should be set to 0 as default).

The type of each client can be seen in */proc/asound/seq/clients*. For example:

```
% cat /proc/asound/seq/clients
Client info
  cur  clients : 3
....
Client  14 : "Midi Through" [Kernel Legacy]
  Port   0 : "Midi Through Port-0" (RWe-)
Client  20 : "ProtoZOA" [Kernel UMP MIDI1]
  UMP Endpoint: ProtoZOA
  UMP Block 0: ProtoZOA Main [Active]
    Groups: 1-1
  UMP Block 1: ProtoZOA Ext IN [Active]
    Groups: 2-2
  UMP Block 2: ProtoZOA Ext OUT [Active]
    Groups: 3-3
  Port   0 : "MIDI 2.0" (RWeX) [In/Out]
  Port   1 : "ProtoZOA Main" (RWeX) [In/Out]
  Port   2 : "ProtoZOA Ext IN" (-We-) [Out]
  Port   3 : "ProtoZOA Ext OUT" (R-e-) [In]
```

Here you can find two types of kernel clients, "Legacy" for client 14, and "UMP MIDI1" for client 20, which is a USB MIDI 2.0 device. A USB MIDI 2.0 client gives always the port 0 as "MIDI 2.0" and the rest ports from 1 for each UMP Group (e.g. port 1 for Group 1). In this example, the device has three active groups (Main, Ext IN and Ext OUT), and those are exposed as sequencer ports from 1 to 3. The "MIDI 2.0" port is for a UMP Endpoint, and its difference from other UMP Group ports is that UMP Endpoint port sends the events from the all ports on the device ("catch-all"), while each UMP Group port sends only the events from the given UMP Group. Also, UMP groupless messages (such as the UMP message type 0x0f) are sent only to the UMP Endpoint port.

Note that, although each UMP sequencer client usually creates 16 ports, those ports that don't belong to any UMP Blocks (or belonging to inactive UMP Blocks) are marked as inactive, and they don't appear in the proc outputs. In the example above, the sequencer ports from 4 to 16 are present but not shown there.

The proc file above shows the UMP Block information, too. The same entry (but with more

---

detailed information) is found in the rawmidi proc output.

When clients are connected between different MIDI versions, the events are translated automatically depending on the client's version, not only between the legacy and the UMP MIDI 1.0/2.0 types, but also between UMP MIDI 1.0 and 2.0 types, too. For example, running *aseqdump* program on the ProtoZOA Main port in the legacy mode will give you the output like:

```
% aseqdump -p 20:1
Waiting for data. Press Ctrl+C to end.
Source  Event                  Ch  Data
 20:1   Note on                 0, note 60, velocity 100
 20:1   Note off                0, note 60, velocity 100
 20:1   Control change          0, controller 11, value 4
```

When you run *aseqdump* in MIDI 2.0 mode, it'll receive the high precision data like:

```
% aseqdump -u 2 -p 20:1
Waiting for data. Press Ctrl+C to end.
Source  Event                  Ch  Data
 20:1   Note on                 0, note 60, velocity 0xc924, attr type = 0,␣
↪data = 0x0
 20:1   Note off                0, note 60, velocity 0xc924, attr type = 0,␣
↪data = 0x0
 20:1   Control change          0, controller 11, value 0x2000000
```

while the data is automatically converted by ALSA sequencer core.

2.12.5 Rawmidi API Extensions

- The additional UMP Endpoint information can be obtained via the new ioctl *SNDRV_UMP_IOCTL_ENDPOINT_INFO*. It contains the associated card and device numbers, the bit flags, the protocols, the number of UMP Blocks, the name string of the endpoint, etc.

  The protocols are specified in two field, the protocol capabilities and the current protocol. Both contain the bit flags specifying the MIDI protocol version (*SNDRV_UMP_EP_INFO_PROTO_MIDI1* or *SNDRV_UMP_EP_INFO_PROTO_MIDI2*) in the upper byte and the jitter reduction timestamp (*SNDRV_UMP_EP_INFO_PROTO_JRTS_TX* and *SNDRV_UMP_EP_INFO_PROTO_JRTS_RX*) in the lower byte.

  A UMP Endpoint may contain up to 32 UMP Blocks, and the number of the currently assigned blocks are shown in the Endpoint information.

- Each UMP Block information can be obtained via another new ioctl *SNDRV_UMP_IOCTL_BLOCK_INFO*. The block ID number (0-based) has to be passed for the block to query. The received data contains the associated the direction of the block, the first associated group ID (0-based) and the number of groups, the name string of the block, etc.

  The direction is either *SNDRV_UMP_DIR_INPUT*, *SNDRV_UMP_DIR_OUTPUT* or *SNDRV_UMP_DIR_BIDIRECTION*.

- For the device supports UMP v1.1, the UMP MIDI protocol can be switched via "Stream Configuration Request" message (UMP type 0x0f, status 0x05). When UMP core receives

such a message, it updates the UMP EP info and the corresponding sequencer clients as well.

2.12.6 Control API Extensions

- The new ioctl *SNDRV_CTL_IOCTL_UMP_NEXT_DEVICE* is introduced for querying the next UMP rawmidi device, while the existing ioctl *SNDRV_CTL_IOCTL_RAWMIDI_NEXT_DEVICE* queries only the legacy rawmidi devices.

  For setting the subdevice (substream number) to be opened, use the ioctl *SNDRV_CTL_IOCTL_RAWMIDI_PREFER_SUBDEVICE* like the normal rawmidi.

- Two new ioctls *SNDRV_CTL_IOCTL_UMP_ENDPOINT_INFO* and *SNDRV_CTL_IOCTL_UMP_BLOCK_INFO* provide the UMP Endpoint and UMP Block information of the specified UMP device via ALSA control API without opening the actual (UMP) rawmidi device. The *card* field is ignored upon inquiry, always tied with the card of the control interface.

2.12.7 Sequencer API Extensions

- *midi_version* field is added to *snd_seq_client_info* to indicate the current MIDI version (either 0, 1 or 2) of each client. When *midi_version* is 1 or 2, the alignment of read from a UMP sequencer client is also changed from the former 28 bytes to 32 bytes for the extended payload. The alignment size for the write isn't changed, but each event size may differ depending on the new bit flag below.

- *SNDRV_SEQ_EVENT_UMP* flag bit is added for each sequencer event flags. When this bit flag is set, the sequencer event is extended to have a larger payload of 16 bytes instead of the legacy 12 bytes, and the event contains the UMP packet in the payload.

- The new sequencer port type bit (*SNDRV_SEQ_PORT_TYPE_MIDI_UMP*) indicates the port being UMP-capable.

- The sequencer ports have new capability bits to indicate the inactive ports (*SNDRV_SEQ_PORT_CAP_INACTIVE*) and the UMP Endpoint port (*SNDRV_SEQ_PORT_CAP_UMP_ENDPOINT*).

- The event conversion of ALSA sequencer clients can be suppressed the new filter bit *SNDRV_SEQ_FILTER_NO_CONVERT* set to the client info. For example, the kernel pass-through client (*snd-seq-dummy*) sets this flag internally.

- The port information gained the new field *direction* to indicate the direction of the port (either *SNDRV_SEQ_PORT_DIR_INPUT*, *SNDRV_SEQ_PORT_DIR_OUTPUT* or *SNDRV_SEQ_PORT_DIR_BIDIRECTION*).

- Another additional field for the port information is *ump_group* which specifies the associated UMP Group Number (1-based). When it's non-zero, the UMP group field in the UMP packet updated upon delivery to the specified group (corrected to be 0-based). Each sequencer port is supposed to set this field if it's a port to specific to a certain UMP group.

- Each client may set the additional event filter for UMP Groups in *group_filter* bitmap. The filter consists of bitmap from 1-based Group numbers. For example, when the bit 1 is set, messages from Group 1 (i.e. the very first group) are filtered and not delivered. The bit 0 is used for filtering UMP groupless messages.

- Two new ioctls are added for UMP-capable clients: *SNDRV_SEQ_IOCTL_GET_CLIENT_UMP_INFO* and *SNDRV_SEQ_IOCTL_SET_CLIENT_UMP_INFO*. They are used to get and set either *snd_ump_endpoint_info* or *snd_ump_block_info* data associated with the sequencer client. The USB MIDI driver provides those information from the underlying UMP rawmidi, while a user-space client may provide its own data via *_SET* ioctl. For an Endpoint data, pass 0 to the *type* field, while for a Block data, pass the block number + 1 to the *type* field. Setting the data for a kernel client shall result in an error.

- With UMP 1.1, Function Block information may be changed dynamically. When the update of Function Block is received from the device, ALSA sequencer core changes the corresponding sequencer port name and attributes accordingly, and notifies the changes via the announcement to the ALSA sequencer system port, similarly like the normal port change notification.

2.12.8 MIDI2 USB Gadget Function Driver

The latest kernel contains the support for USB MIDI 2.0 gadget function driver, which can be used for prototyping and debugging MIDI 2.0 features.

CONFIG_USB_GADGET, *CONFIG_USB_CONFIGFS* and *CONFIG_USB_CONFIGFS_F_MIDI2* need to be enabled for the MIDI2 gadget driver.

In addition, for using a gadget driver, you need a working UDC driver. In the example below, we use *dummy_hcd* driver (enabled via *CONFIG_USB_DUMMY_HCD*) that is available on PC and VM for debugging purpose. There are other UDC drivers depending on the platform, and those can be used for a real device, instead, too.

At first, on a system to run the gadget, load *libcomposite* module:

```
% modprobe libcomposite
```

and you'll have *usb_gadget* subdirectory under configfs space (typically */sys/kernel/config* on modern OS). Then create a gadget instance and add configurations there, for example:

```
% cd /sys/kernel/config
% mkdir usb_gadget/g1

% cd usb_gadget/g1
% mkdir configs/c.1
% mkdir functions/midi2.usb0

% echo 0x0004 > idProduct
% echo 0x17b3 > idVendor
% mkdir strings/0x409
% echo "ACME Enterprises" > strings/0x409/manufacturer
% echo "ACMESynth" > strings/0x409/product
% echo "ABCD12345" > strings/0x409/serialnumber

% mkdir configs/c.1/strings/0x409
% echo "Monosynth" > configs/c.1/strings/0x409/configuration
% echo 120 > configs/c.1/MaxPower
```

At this point, there must be a subdirectory *ep.0*, and that is the configuration for a UMP Endpoint. You can fill the Endpoint information like:

```
% echo "ACMESynth" > functions/midi2.usb0/iface_name
% echo "ACMESynth" > functions/midi2.usb0/ep.0/ep_name
% echo "ABCD12345" > functions/midi2.usb0/ep.0/product_id
% echo 0x0123 > functions/midi2.usb0/ep.0/family
% echo 0x4567 > functions/midi2.usb0/ep.0/model
% echo 0x123456 > functions/midi2.usb0/ep.0/manufacturer
% echo 0x12345678 > functions/midi2.usb0/ep.0/sw_revision
```

The default MIDI protocol can be set either 1 or 2:

```
% echo 2 > functions/midi2.usb0/ep.0/protocol
```

And, you can find a subdirectory *block.0* under this Endpoint subdirectory. This defines the Function Block information:

```
% echo "Monosynth" > functions/midi2.usb0/ep.0/block.0/name
% echo 0 > functions/midi2.usb0/ep.0/block.0/first_group
% echo 1 > functions/midi2.usb0/ep.0/block.0/num_groups
```

Finally, link the configuration and enable it:

```
% ln -s functions/midi2.usb0 configs/c.1
% echo dummy_udc.0 > UDC
```

where *dummy_udc.0* is an example case and it differs depending on the system. You can find the UDC instances in */sys/class/udc* and pass the found name instead:

```
% ls /sys/class/udc
dummy_udc.0
```

Now, the MIDI 2.0 gadget device is enabled, and the gadget host creates a new sound card instance containing a UMP rawmidi device by *f_midi2* driver:

```
% cat /proc/asound/cards
....
1 [Gadget         ]: f_midi2 - MIDI 2.0 Gadget
                     MIDI 2.0 Gadget
```

And on the connected host, a similar card should appear, too, but with the card and device names given in the configfs above:

```
% cat /proc/asound/cards
....
2 [ACMESynth      ]: USB-Audio - ACMESynth
                     ACME Enterprises ACMESynth at usb-dummy_hcd.0-1, high␣
→speed
```

You can play a MIDI file on the gadget side:

```
% aplaymidi -p 20:1 to_host.mid
```

and this will appear as an input from a MIDI device on the connected host:

```
% aseqdump -p 20:0 -u 2
```

Vice versa, a playback on the connected host will work as an input on the gadget, too.

Each Function Block may have different direction and UI-hint, specified via *direction* and *ui_hint* attributes. Passing *1* is for input-only, *2* for out-only and *3* for bidirectional (the default value). For example:

```
% echo 2 > functions/midi2.usb0/ep.0/block.0/direction
% echo 2 > functions/midi2.usb0/ep.0/block.0/ui_hint
```

When you need more than one Function Blocks, you can create subdirectories *block.1*, *block.2*, etc dynamically, and configure them in the configuration procedure above before linking. For example, to create a second Function Block for a keyboard:

```
% mkdir functions/midi2.usb0/ep.0/block.1
% echo "Keyboard" > functions/midi2.usb0/ep.0/block.1/name
% echo 1 > functions/midi2.usb0/ep.0/block.1/first_group
% echo 1 > functions/midi2.usb0/ep.0/block.1/num_groups
% echo 1 > functions/midi2.usb0/ep.0/block.1/direction
% echo 1 > functions/midi2.usb0/ep.0/block.1/ui_hint
```

The *block.** subdirectories can be removed dynamically, too (except for *block.0* which is persistent).

For assigning a Function Block for MIDI 1.0 I/O, set up in *is_midi1* attribute. 1 is for MIDI 1.0, and 2 is for MIDI 1.0 with low speed connection:

```
% echo 2 > functions/midi2.usb0/ep.0/block.1/is_midi1
```

For disabling the processing of UMP Stream messages in the gadget driver, pass *0* to *process_ump* attribute in the top-level config:

```
% echo 0 > functions/midi2.usb0/process_ump
```

The MIDI 1.0 interface at altset 0 is supported by the gadget driver, too. When MIDI 1.0 interface is selected by the connected host, the UMP I/O on the gadget is translated from/to USB MIDI 1.0 packets accordingly while the gadget driver keeps communicating with the user-space over UMP rawmidi.

MIDI 1.0 ports are set up from the config in each Function Block. For example:

```
% echo 0 > functions/midi2.usb0/ep.0/block.0/midi1_first_group
% echo 1 > functions/midi2.usb0/ep.0/block.0/midi1_num_groups
```

The configuration above will enable the Group 1 (the index 0) for MIDI 1.0 interface. Note that those groups must be in the groups defined for the Function Block itself.

The gadget driver supports more than one UMP Endpoints, too. Similarly like the Function Blocks, you can create a new subdirectory *ep.1* (but under the card top-level config) to enable

a new Endpoint:

```
% mkdir functions/midi2.usb0/ep.1
```

and create a new Function Block there. For example, to create 4 Groups for the Function Block of this new Endpoint:

```
% mkdir functions/midi2.usb0/ep.1/block.0
% echo 4 > functions/midi2.usb0/ep.1/block.0/num_groups
```

Now, you'll have 4 rawmidi devices in total: the first two are UMP rawmidi devices for Endpoint 0 and Endpoint 1, and other two for the legacy MIDI 1.0 rawmidi devices corresponding to both EP 0 and EP 1.

The current altsetting on the gadget can be informed via a control element "Operation Mode" with *RAWMIDI* iface. e.g. you can read it via *amixer* program running on the gadget host like:

```
% amixer -c1 cget iface=RAWMIDI,name='Operation Mode'
; type=INTEGER,access=r--v----,values=1,min=0,max=2,step=0
: values=2
```

The value (shown in the second returned line with *: values=*) indicates 1 for MIDI 1.0 (altset 0), 2 for MIDI 2.0 (altset 1) and 0 for unset.

As of now, the configurations can't be changed after binding.

ALSA SOC LAYER

The documentation is spilt into the following sections:-

3.1 ALSA SoC Layer Overview

The overall project goal of the ALSA System on Chip (ASoC) layer is to provide better ALSA support for embedded system-on-chip processors (e.g. pxa2xx, au1x00, iMX, etc) and portable audio codecs. Prior to the ASoC subsystem there was some support in the kernel for SoC audio, however it had some limitations:-

- Codec drivers were often tightly coupled to the underlying SoC CPU. This is not ideal and leads to code duplication - for example, Linux had different wm8731 drivers for 4 different SoC platforms.

- There was no standard method to signal user initiated audio events (e.g. Headphone/Mic insertion, Headphone/Mic detection after an insertion event). These are quite common events on portable devices and often require machine specific code to re-route audio, enable amps, etc., after such an event.

- Drivers tended to power up the entire codec when playing (or recording) audio. This is fine for a PC, but tends to waste a lot of power on portable devices. There was also no support for saving power via changing codec oversampling rates, bias currents, etc.

3.1.1 ASoC Design

The ASoC layer is designed to address these issues and provide the following features :-

- Codec independence. Allows reuse of codec drivers on other platforms and machines.

- Easy I2S/PCM audio interface setup between codec and SoC. Each SoC interface and codec registers its audio interface capabilities with the core and are subsequently matched and configured when the application hardware parameters are known.

- Dynamic Audio Power Management (DAPM). DAPM automatically sets the codec to its minimum power state at all times. This includes powering up/down internal power blocks depending on the internal codec audio routing and any active streams.

- Pop and click reduction. Pops and clicks can be reduced by powering the codec up/down in the correct sequence (including using digital mute). ASoC signals the codec when to change power states.

- Machine specific controls: Allow machines to add controls to the sound card (e.g. volume control for speaker amplifier).

To achieve all this, ASoC basically splits an embedded audio system into multiple re-usable component drivers :-

- Codec class drivers: The codec class driver is platform independent and contains audio controls, audio interface capabilities, codec DAPM definition and codec IO functions. This class extends to BT, FM and MODEM ICs if required. Codec class drivers should be generic code that can run on any architecture and machine.

- Platform class drivers: The platform class driver includes the audio DMA engine driver, digital audio interface (DAI) drivers (e.g. I2S, AC97, PCM) and any audio DSP drivers for that platform.

- Machine class driver: The machine driver class acts as the glue that describes and binds the other component drivers together to form an ALSA "sound card device". It handles any machine specific controls and machine level audio events (e.g. turning on an amp at start of playback).

3.2 ASoC Codec Class Driver

The codec class driver is generic and hardware independent code that configures the codec, FM, MODEM, BT or external DSP to provide audio capture and playback. It should contain no code that is specific to the target platform or machine. All platform and machine specific code should be added to the platform and machine drivers respectively.

Each codec class driver *must* provide the following features:-

1. Codec DAI and PCM configuration
2. Codec control IO - using RegMap API
3. Mixers and audio controls
4. Codec audio operations
5. DAPM description.
6. DAPM event handler.

Optionally, codec drivers can also provide:-

7. DAC Digital mute control.

Its probably best to use this guide in conjunction with the existing codec driver code in sound/soc/codecs/

3.2.1 ASoC Codec driver breakdown

Codec DAI and PCM configuration

Each codec driver must have a struct snd_soc_dai_driver to define its DAI and PCM capabilities and operations. This struct is exported so that it can be registered with the core by your machine driver.

e.g.

```
static struct snd_soc_dai_ops wm8731_dai_ops = {
        .prepare        = wm8731_pcm_prepare,
        .hw_params      = wm8731_hw_params,
        .shutdown       = wm8731_shutdown,
        .mute_stream    = wm8731_mute,
        .set_sysclk     = wm8731_set_dai_sysclk,
        .set_fmt        = wm8731_set_dai_fmt,
};

struct snd_soc_dai_driver wm8731_dai = {
        .name = "wm8731-hifi",
        .playback = {
                .stream_name = "Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = WM8731_RATES,
                .formats = WM8731_FORMATS,},
        .capture = {
                .stream_name = "Capture",
                .channels_min = 1,
                .channels_max = 2,
                .rates = WM8731_RATES,
                .formats = WM8731_FORMATS,},
        .ops = &wm8731_dai_ops,
        .symmetric_rate = 1,
};
```

Codec control IO

The codec can usually be controlled via an I2C or SPI style interface (AC97 combines control with data in the DAI). The codec driver should use the Regmap API for all codec IO. Please see include/linux/regmap.h and existing codec drivers for example regmap usage.

Mixers and audio controls

All the codec mixers and audio controls can be defined using the convenience macros defined in soc.h.

```
#define SOC_SINGLE(xname, reg, shift, mask, invert)
```

Defines a single control as follows:-

```
xname = Control name e.g. "Playback Volume"
reg = codec register
shift = control bit(s) offset in register
mask = control bit size(s) e.g. mask of 7 = 3 bits
invert = the control is inverted
```

Other macros include:-

```
#define SOC_DOUBLE(xname, reg, shift_left, shift_right, mask, invert)
```

A stereo control

```
#define SOC_DOUBLE_R(xname, reg_left, reg_right, shift, mask, invert)
```

A stereo control spanning 2 registers

```
#define SOC_ENUM_SINGLE(xreg, xshift, xmask, xtexts)
```

Defines an single enumerated control as follows:-

```
xreg = register
xshift = control bit(s) offset in register
xmask = control bit(s) size
xtexts = pointer to array of strings that describe each setting

#define SOC_ENUM_DOUBLE(xreg, xshift_l, xshift_r, xmask, xtexts)
```

Defines a stereo enumerated control

Codec Audio Operations

The codec driver also supports the following ALSA PCM operations:-

```
/* SoC audio ops */
struct snd_soc_ops {
      int (*startup)(struct snd_pcm_substream *);
      void (*shutdown)(struct snd_pcm_substream *);
      int (*hw_params)(struct snd_pcm_substream *, struct snd_pcm_hw_params *);
      int (*hw_free)(struct snd_pcm_substream *);
      int (*prepare)(struct snd_pcm_substream *);
};
```

Please refer to the ALSA driver PCM documentation for details. https://www.kernel.org/doc/html/latest/sound/kernel-api/writing-an-alsa-driver.html

DAPM description

The Dynamic Audio Power Management description describes the codec power components and their relationships and registers to the ASoC core. Please read dapm.rst for details of building the description.

Please also see the examples in other codec drivers.

DAPM event handler

This function is a callback that handles codec domain PM calls and system domain PM calls (e.g. suspend and resume). It is used to put the codec to sleep when not in use.

Power states:-

```
SNDRV_CTL_POWER_D0: /* full On */
/* vref/mid, clk and osc on, active */

SNDRV_CTL_POWER_D1: /* partial On */
SNDRV_CTL_POWER_D2: /* partial On */

SNDRV_CTL_POWER_D3hot: /* Off, with power */
/* everything off except vref/vmid, inactive */

SNDRV_CTL_POWER_D3cold: /* Everything Off, without power */
```

Codec DAC digital mute control

Most codecs have a digital mute before the DACs that can be used to minimise any system noise. The mute stops any digital data from entering the DAC.

A callback can be created that is called by the core for each codec DAI when the mute is applied or freed.

i.e.

```
static int wm8974_mute(struct snd_soc_dai *dai, int mute, int direction)
{
        struct snd_soc_component *component = dai->component;
        u16 mute_reg = snd_soc_component_read(component, WM8974_DAC) & 0xffbf;

        if (mute)
                snd_soc_component_write(component, WM8974_DAC, mute_reg | 0x40);
        else
                snd_soc_component_write(component, WM8974_DAC, mute_reg);
        return 0;
}
```

3.3 ASoC Digital Audio Interface (DAI)

ASoC currently supports the three main Digital Audio Interfaces (DAI) found on SoC controllers and portable audio CODECs today, namely AC97, I2S and PCM.

3.3.1 AC97

AC97 is a five wire interface commonly found on many PC sound cards. It is now also popular in many portable devices. This DAI has a RESET line and time multiplexes its data on its SDATA_OUT (playback) and SDATA_IN (capture) lines. The bit clock (BCLK) is always driven by the CODEC (usually 12.288MHz) and the frame (FRAME) (usually 48kHz) is always driven by the controller. Each AC97 frame is 21uS long and is divided into 13 time slots.

The AC97 specification can be found at : https://www.intel.com/p/en_US/business/design

3.3.2 I2S

I2S is a common 4 wire DAI used in HiFi, STB and portable devices. The Tx and Rx lines are used for audio transmission, while the bit clock (BCLK) and left/right clock (LRC) synchronise the link. I2S is flexible in that either the controller or CODEC can drive (master) the BCLK and LRC clock lines. Bit clock usually varies depending on the sample rate and the master system clock (SYSCLK). LRCLK is the same as the sample rate. A few devices support separate ADC and DAC LRCLKs, this allows for simultaneous capture and playback at different sample rates.

I2S has several different operating modes:-

I2S
> MSB is transmitted on the falling edge of the first BCLK after LRC transition.

Left Justified
> MSB is transmitted on transition of LRC.

Right Justified
> MSB is transmitted sample size BCLKs before LRC transition.

3.3.3 PCM

PCM is another 4 wire interface, very similar to I2S, which can support a more flexible protocol. It has bit clock (BCLK) and sync (SYNC) lines that are used to synchronise the link while the Tx and Rx lines are used to transmit and receive the audio data. Bit clock usually varies depending on sample rate while sync runs at the sample rate. PCM also supports Time Division Multiplexing (TDM) in that several devices can use the bus simultaneously (this is sometimes referred to as network mode).

Common PCM operating modes:-

Mode A
> MSB is transmitted on falling edge of first BCLK after FRAME/SYNC.

Mode B
> MSB is transmitted on rising edge of FRAME/SYNC.

3.4 Dynamic Audio Power Management for Portable Devices

3.4.1 Description

Dynamic Audio Power Management (DAPM) is designed to allow portable Linux devices to use the minimum amount of power within the audio subsystem at all times. It is independent of other kernel PM and as such, can easily co-exist with the other PM systems.

DAPM is also completely transparent to all user space applications as all power switching is done within the ASoC core. No code changes or recompiling are required for user space applications. DAPM makes power switching decisions based upon any audio stream (capture/playback) activity and audio mixer settings within the device.

DAPM spans the whole machine. It covers power control within the entire audio subsystem, this includes internal codec power blocks and machine level power systems.

There are 4 power domains within DAPM

Codec bias domain
VREF, VMID (core codec and audio power)

Usually controlled at codec probe/remove and suspend/resume, although can be set at stream time if power is not needed for sidetone, etc.

Platform/Machine domain
physically connected inputs and outputs

Is platform/machine and user action specific, is configured by the machine driver and responds to asynchronous events e.g when HP are inserted

Path domain
audio subsystem signal paths

Automatically set when mixer and mux settings are changed by the user. e.g. alsamixer, amixer.

Stream domain
DACs and ADCs.

Enabled and disabled when stream playback/capture is started and stopped respectively. e.g. aplay, arecord.

All DAPM power switching decisions are made automatically by consulting an audio routing map of the whole machine. This map is specific to each machine and consists of the interconnections between every audio component (including internal codec components). All audio components that effect power are called widgets hereafter.

3.4.2 DAPM Widgets

Audio DAPM widgets fall into a number of types:-

Mixer
> Mixes several analog signals into a single analog signal.

Mux
> An analog switch that outputs only one of many inputs.

PGA
> A programmable gain amplifier or attenuation widget.

ADC
> Analog to Digital Converter

DAC
> Digital to Analog Converter

Switch
> An analog switch

Input
> A codec input pin

Output
> A codec output pin

Headphone
> Headphone (and optional Jack)

Mic
> Mic (and optional Jack)

Line
> Line Input/Output (and optional Jack)

Speaker
> Speaker

Supply
> Power or clock supply widget used by other widgets.

Regulator
> External regulator that supplies power to audio components.

Clock
> External clock that supplies clock to audio components.

AIF IN
> Audio Interface Input (with TDM slot mask).

AIF OUT
> Audio Interface Output (with TDM slot mask).

Siggen
> Signal Generator.

DAI IN
> Digital Audio Interface Input.

DAI OUT
    Digital Audio Interface Output.

DAI Link
    DAI Link between two DAI structures

Pre
    Special PRE widget (exec before all others)

Post
    Special POST widget (exec after all others)

Buffer
    Inter widget audio data buffer within a DSP.

Scheduler
    DSP internal scheduler that schedules component/pipeline processing work.

Effect
    Widget that performs an audio processing effect.

SRC
    Sample Rate Converter within DSP or CODEC

ASRC
    Asynchronous Sample Rate Converter within DSP or CODEC

Encoder
    Widget that encodes audio data from one format (usually PCM) to another usually more
    compressed format.

Decoder
    Widget that decodes audio data from a compressed format to an uncompressed format like
    PCM.

(Widgets are defined in include/sound/soc-dapm.h)

Widgets can be added to the sound card by any of the component driver types. There are
convenience macros defined in soc-dapm.h that can be used to quickly build a list of widgets of
the codecs and machines DAPM widgets.

Most widgets have a name, register, shift and invert. Some widgets have extra parameters for
stream name and kcontrols.

Stream Domain Widgets

Stream Widgets relate to the stream power domain and only consist of ADCs (analog to digital
converters), DACs (digital to analog converters), AIF IN and AIF OUT.

Stream widgets have the following format:-

```
SND_SOC_DAPM_DAC(name, stream name, reg, shift, invert),
SND_SOC_DAPM_AIF_IN(name, stream, slot, reg, shift, invert)
```

NOTE: the stream name must match the corresponding stream name in your codec
snd_soc_codec_dai.

e.g. stream widgets for HiFi playback and capture

```
SND_SOC_DAPM_DAC("HiFi DAC", "HiFi Playback", REG, 3, 1),
SND_SOC_DAPM_ADC("HiFi ADC", "HiFi Capture", REG, 2, 1),
```

e.g. stream widgets for AIF

```
SND_SOC_DAPM_AIF_IN("AIF1RX", "AIF1 Playback", 0, SND_SOC_NOPM, 0, 0),
SND_SOC_DAPM_AIF_OUT("AIF1TX", "AIF1 Capture", 0, SND_SOC_NOPM, 0, 0),
```

Path Domain Widgets

Path domain widgets have a ability to control or affect the audio signal or audio paths within the audio subsystem. They have the following form:-

```
SND_SOC_DAPM_PGA(name, reg, shift, invert, controls, num_controls)
```

Any widget kcontrols can be set using the controls and num_controls members.

e.g. Mixer widget (the kcontrols are declared first)

```
/* Output Mixer */
static const snd_kcontrol_new_t wm8731_output_mixer_controls[] = {
SOC_DAPM_SINGLE("Line Bypass Switch", WM8731_APANA, 3, 1, 0),
SOC_DAPM_SINGLE("Mic Sidetone Switch", WM8731_APANA, 5, 1, 0),
SOC_DAPM_SINGLE("HiFi Playback Switch", WM8731_APANA, 4, 1, 0),
};

SND_SOC_DAPM_MIXER("Output Mixer", WM8731_PWR, 4, 1, wm8731_output_mixer_
↪controls,
      ARRAY_SIZE(wm8731_output_mixer_controls)),
```

If you don't want the mixer elements prefixed with the name of the mixer widget, you can use SND_SOC_DAPM_MIXER_NAMED_CTL instead. the parameters are the same as for SND_SOC_DAPM_MIXER.

Machine domain Widgets

Machine widgets are different from codec widgets in that they don't have a codec register bit associated with them. A machine widget is assigned to each machine audio component (non codec or DSP) that can be independently powered. e.g.

- Speaker Amp

- Microphone Bias

- Jack connectors

A machine widget can have an optional call back.

e.g. Jack connector widget for an external Mic that enables Mic Bias when the Mic is inserted:-:

```
static int spitz_mic_bias(struct snd_soc_dapm_widget* w, int event)
{
      gpio_set_value(SPITZ_GPIO_MIC_BIAS, SND_SOC_DAPM_EVENT_ON(event));
```

```
      return 0;
}

SND_SOC_DAPM_MIC("Mic Jack", spitz_mic_bias),
```

Codec (BIAS) Domain

The codec bias power domain has no widgets and is handled by the codecs DAPM event handler. This handler is called when the codec powerstate is changed wrt to any stream event or by kernel PM events.

Virtual Widgets

Sometimes widgets exist in the codec or machine audio map that don't have any corresponding soft power control. In this case it is necessary to create a virtual widget - a widget with no control bits e.g.

```
SND_SOC_DAPM_MIXER("AC97 Mixer", SND_SOC_NOPM, 0, 0, NULL, 0),
```

This can be used to merge to signal paths together in software.

After all the widgets have been defined, they can then be added to the DAPM subsystem individually with a call to *snd_soc_dapm_new_control()*.

3.4.3 Codec/DSP Widget Interconnections

Widgets are connected to each other within the codec, platform and machine by audio paths (called interconnections). Each interconnection must be defined in order to create a map of all audio paths between widgets.

This is easiest with a diagram of the codec or DSP (and schematic of the machine audio system), as it requires joining widgets together via their audio signal paths.

e.g., from the WM8731 output mixer (wm8731.c)

The WM8731 output mixer has 3 inputs (sources)

1. Line Bypass Input
2. DAC (HiFi playback)
3. Mic Sidetone Input

Each input in this example has a kcontrol associated with it (defined in example above) and is connected to the output mixer via its kcontrol name. We can now connect the destination widget (wrt audio signal) with its source widgets.

```
/* output mixer */
{"Output Mixer", "Line Bypass Switch", "Line Input"},
{"Output Mixer", "HiFi Playback Switch", "DAC"},
{"Output Mixer", "Mic Sidetone Switch", "Mic Bias"},
```

So we have :-

- Destination Widget <=== Path Name <=== Source Widget, or

- Sink, Path, Source, or

- `Output Mixer` is connected to the DAC via the `HiFi Playback Switch`.

When there is no path name connecting widgets (e.g. a direct connection) we pass NULL for the path name.

Interconnections are created with a call to:-

```
snd_soc_dapm_connect_input(codec, sink, path, source);
```

Finally, snd_soc_dapm_new_widgets(codec) must be called after all widgets and interconnections have been registered with the core. This causes the core to scan the codec and machine so that the internal DAPM state matches the physical state of the machine.

Machine Widget Interconnections

Machine widget interconnections are created in the same way as codec ones and directly connect the codec pins to machine level widgets.

e.g. connects the speaker out codec pins to the internal speaker.

```
/* ext speaker connected to codec pins LOUT2, ROUT2  */
{"Ext Spk", NULL , "ROUT2"},
{"Ext Spk", NULL , "LOUT2"},
```

This allows the DAPM to power on and off pins that are connected (and in use) and pins that are NC respectively.

3.4.4 Endpoint Widgets

An endpoint is a start or end point (widget) of an audio signal within the machine and includes the codec. e.g.

- Headphone Jack

- Internal Speaker

- Internal Mic

- Mic Jack

- Codec Pins

Endpoints are added to the DAPM graph so that their usage can be determined in order to save power. e.g. NC codecs pins will be switched OFF, unconnected jacks can also be switched OFF.

3.4.5 DAPM Widget Events

Some widgets can register their interest with the DAPM core in PM events. e.g. A Speaker with an amplifier registers a widget so the amplifier can be powered only when the spk is in use.

```
/* turn speaker amplifier on/off depending on use */
static int corgi_amp_event(struct snd_soc_dapm_widget *w, int event)
{
        gpio_set_value(CORGI_GPIO_APM_ON, SND_SOC_DAPM_EVENT_ON(event));
        return 0;
}

/* corgi machine dapm widgets */
static const struct snd_soc_dapm_widget wm8731_dapm_widgets =
        SND_SOC_DAPM_SPK("Ext Spk", corgi_amp_event);
```

Please see soc-dapm.h for all other widgets that support events.

Event types

The following event types are supported by event widgets.

```
/* dapm event types */
#define SND_SOC_DAPM_PRE_PMU  0x1       /* before widget power up */
#define SND_SOC_DAPM_POST_PMU 0x2           /* after widget power up */
#define SND_SOC_DAPM_PRE_PMD  0x4       /* before widget power down */
#define SND_SOC_DAPM_POST_PMD 0x8           /* after widget power down */
#define SND_SOC_DAPM_PRE_REG  0x10      /* before audio path setup */
#define SND_SOC_DAPM_POST_REG 0x20      /* after audio path setup */
```

3.5 ASoC Platform Driver

An ASoC platform driver class can be divided into audio DMA drivers, SoC DAI drivers and DSP drivers. The platform drivers only target the SoC CPU and must have no board specific code.

3.5.1 Audio DMA

The platform DMA driver optionally supports the following ALSA operations:-

```
/* SoC audio ops */
struct snd_soc_ops {
        int (*startup)(struct snd_pcm_substream *);
        void (*shutdown)(struct snd_pcm_substream *);
        int (*hw_params)(struct snd_pcm_substream *, struct snd_pcm_hw_params *);
        int (*hw_free)(struct snd_pcm_substream *);
        int (*prepare)(struct snd_pcm_substream *);
        int (*trigger)(struct snd_pcm_substream *, int);
};
```

The platform driver exports its DMA functionality via struct snd_soc_component_driver:-

```
struct snd_soc_component_driver {
      const char *name;

      ...
      int (*probe)(struct snd_soc_component *);
      void (*remove)(struct snd_soc_component *);
      int (*suspend)(struct snd_soc_component *);
      int (*resume)(struct snd_soc_component *);

      /* pcm creation and destruction */
      int (*pcm_new)(struct snd_soc_pcm_runtime *);
      void (*pcm_free)(struct snd_pcm *);

      ...
      const struct snd_pcm_ops *ops;
      const struct snd_compr_ops *compr_ops;
      ...
};
```

Please refer to the ALSA driver documentation for details of audio DMA. https://www.kernel.org/doc/html/latest/sound/kernel-api/writing-an-alsa-driver.html

An example DMA driver is soc/pxa/pxa2xx-pcm.c

3.5.2 SoC DAI Drivers

Each SoC DAI driver must provide the following features:-

1. Digital audio interface (DAI) description
2. Digital audio interface configuration
3. PCM's description
4. SYSCLK configuration
5. Suspend and resume (optional)

Please see codec.rst for a description of items 1 - 4.

3.5.3 SoC DSP Drivers

Each SoC DSP driver usually supplies the following features :-

1. DAPM graph
2. Mixer controls
3. DMA IO to/from DSP buffers (if applicable)
4. Definition of DSP front end (FE) PCM devices.

Please see DPCM.txt for a description of item 4.

3.6 ASoC Machine Driver

The ASoC machine (or board) driver is the code that glues together all the component drivers (e.g. codecs, platforms and DAIs). It also describes the relationships between each component which include audio paths, GPIOs, interrupts, clocking, jacks and voltage regulators.

The machine driver can contain codec and platform specific code. It registers the audio subsystem with the kernel as a platform device and is represented by the following struct:-

```
/* SoC machine */
struct snd_soc_card {
        char *name;

        ...

        int (*probe)(struct platform_device *pdev);
        int (*remove)(struct platform_device *pdev);

        /* the pre and post PM functions are used to do any PM work before and
         * after the codec and DAIs do any PM work. */
        int (*suspend_pre)(struct platform_device *pdev, pm_message_t state);
        int (*suspend_post)(struct platform_device *pdev, pm_message_t state);
        int (*resume_pre)(struct platform_device *pdev);
        int (*resume_post)(struct platform_device *pdev);

        ...

        /* CPU <--> Codec DAI links  */
        struct snd_soc_dai_link *dai_link;
        int num_links;

        ...
};
```

3.6.1 probe()/remove()

probe/remove are optional. Do any machine specific probe here.

3.6.2 suspend()/resume()

The machine driver has pre and post versions of suspend and resume to take care of any machine audio tasks that have to be done before or after the codec, DAIs and DMA is suspended and resumed. Optional.

3.6.3 Machine DAI Configuration

The machine DAI configuration glues all the codec and CPU DAIs together. It can also be used to set up the DAI system clock and for any machine related DAI initialisation e.g. the machine audio map can be connected to the codec audio map, unconnected codec pins can be set as such.

struct snd_soc_dai_link is used to set up each DAI in your machine. e.g.

```
/* corgi digital audio interface glue - connects codec <--> CPU */
static struct snd_soc_dai_link corgi_dai = {
        .name = "WM8731",
        .stream_name = "WM8731",
        .cpu_dai_name = "pxa-is2-dai",
        .codec_dai_name = "wm8731-hifi",
        .platform_name = "pxa-pcm-audio",
        .codec_name = "wm8713-codec.0-001a",
        .init = corgi_wm8731_init,
        .ops = &corgi_ops,
};
```

struct snd_soc_card then sets up the machine with its DAIs. e.g.

```
/* corgi audio machine driver */
static struct snd_soc_card snd_soc_corgi = {
        .name = "Corgi",
        .dai_link = &corgi_dai,
        .num_links = 1,
};
```

3.6.4 Machine Power Map

The machine driver can optionally extend the codec power map and to become an audio power map of the audio subsystem. This allows for automatic power up/down of speaker/HP amplifiers, etc. Codec pins can be connected to the machines jack sockets in the machine init function.

3.6.5 Machine Controls

Machine specific audio mixer controls can be added in the DAI init function.

3.7 Audio Pops and Clicks

Pops and clicks are unwanted audio artifacts caused by the powering up and down of components within the audio subsystem. This is noticeable on PCs when an audio module is either loaded or unloaded (at module load time the sound card is powered up and causes a popping noise on the speakers).

Pops and clicks can be more frequent on portable systems with DAPM. This is because the components within the subsystem are being dynamically powered depending on the audio usage

and this can subsequently cause a small pop or click every time a component power state is changed.

3.7.1 Minimising Playback Pops and Clicks

Playback pops in portable audio subsystems cannot be completely eliminated currently, however future audio codec hardware will have better pop and click suppression. Pops can be reduced within playback by powering the audio components in a specific order. This order is different for startup and shutdown and follows some basic rules:-

```
Startup Order :- DAC --> Mixers --> Output PGA --> Digital Unmute

Shutdown Order :- Digital Mute --> Output PGA --> Mixers --> DAC
```

This assumes that the codec PCM output path from the DAC is via a mixer and then a PGA (programmable gain amplifier) before being output to the speakers.

3.7.2 Minimising Capture Pops and Clicks

Capture artifacts are somewhat easier to get rid as we can delay activating the ADC until all the pops have occurred. This follows similar power rules to playback in that components are powered in a sequence depending upon stream startup or shutdown.

```
Startup Order - Input PGA --> Mixers --> ADC

Shutdown Order - ADC --> Mixers --> Input PGA
```

3.7.3 Zipper Noise

An unwanted zipper noise can occur within the audio playback or capture stream when a volume control is changed near its maximum gain value. The zipper noise is heard when the gain increase or decrease changes the mean audio signal amplitude too quickly. It can be minimised by enabling the zero cross setting for each volume control. The ZC forces the gain change to occur when the signal crosses the zero amplitude line.

3.8 Audio Clocking

This text describes the audio clocking terms in ASoC and digital audio in general. Note: Audio clocking can be complex!

3.8.1 Master Clock

Every audio subsystem is driven by a master clock (sometimes referred to as MCLK or SYSCLK). This audio master clock can be derived from a number of sources (e.g. crystal, PLL, CPU clock) and is responsible for producing the correct audio playback and capture sample rates.

Some master clocks (e.g. PLLs and CPU based clocks) are configurable in that their speed can be altered by software (depending on the system use and to save power). Other master clocks are fixed at a set frequency (i.e. crystals).

3.8.2 DAI Clocks

The Digital Audio Interface is usually driven by a Bit Clock (often referred to as BCLK). This clock is used to drive the digital audio data across the link between the codec and CPU.

The DAI also has a frame clock to signal the start of each audio frame. This clock is sometimes referred to as LRC (left right clock) or FRAME. This clock runs at exactly the sample rate (LRC = Rate).

Bit Clock can be generated as follows:-

- BCLK = MCLK / x, or
- BCLK = LRC * x, or
- BCLK = LRC * Channels * Word Size

This relationship depends on the codec or SoC CPU in particular. In general it is best to configure BCLK to the lowest possible speed (depending on your rate, number of channels and word size) to save on power.

It is also desirable to use the codec (if possible) to drive (or master) the audio clocks as it usually gives more accurate sample rates than the CPU.

3.9 ASoC jack detection

ALSA has a standard API for representing physical jacks to user space, the kernel side of which can be seen in include/sound/jack.h. ASoC provides a version of this API adding two additional features:

- It allows more than one jack detection method to work together on one user visible jack. In embedded systems it is common for multiple to be present on a single jack but handled by separate bits of hardware.
- Integration with DAPM, allowing DAPM endpoints to be updated automatically based on the detected jack status (eg, turning off the headphone outputs if no headphones are present).

This is done by splitting the jacks up into three things working together: the jack itself represented by a struct snd_soc_jack, sets of snd_soc_jack_pins representing DAPM endpoints to update and blocks of code providing jack reporting mechanisms.

For example, a system may have a stereo headset jack with two reporting mechanisms, one for the headphone and one for the microphone. Some systems won't be able to use their speaker

output while a headphone is connected and so will want to make sure to update both speaker and headphone when the headphone jack status changes.

3.9.1 The jack - struct snd_soc_jack

This represents a physical jack on the system and is what is visible to user space. The jack itself is completely passive, it is set up by the machine driver and updated by jack detection methods.

Jacks are created by the machine driver calling snd_soc_jack_new().

3.9.2 snd_soc_jack_pin

These represent a DAPM pin to update depending on some of the status bits supported by the jack. Each snd_soc_jack has zero or more of these which are updated automatically. They are created by the machine driver and associated with the jack using *snd_soc_jack_add_pins()*. The status of the endpoint may configured to be the opposite of the jack status if required (eg, enabling a built in microphone if a microphone is not connected via a jack).

3.9.3 Jack detection methods

Actual jack detection is done by code which is able to monitor some input to the system and update a jack by calling *snd_soc_jack_report()*, specifying a subset of bits to update. The jack detection code should be set up by the machine driver, taking configuration for the jack to update and the set of things to report when the jack is connected.

Often this is done based on the status of a GPIO - a handler for this is provided by the snd_soc_jack_add_gpio() function. Other methods are also available, for example integrated into CODECs. One example of CODEC integrated jack detection can be see in the WM8350 driver.

Each jack may have multiple reporting mechanisms, though it will need at least one to be useful.

3.9.4 Machine drivers

These are all hooked together by the machine driver depending on the system hardware. The machine driver will set up the snd_soc_jack and the list of pins to update then set up one or more jack detection mechanisms to update that jack based on their current status.

3.10 Dynamic PCM

3.10.1 Description

Dynamic PCM allows an ALSA PCM device to digitally route its PCM audio to various digital endpoints during the PCM stream runtime. e.g. PCM0 can route digital audio to I2S DAI0, I2S DAI1 or PDM DAI2. This is useful for on SoC DSP drivers that expose several ALSA PCMs and can route to multiple DAIs.
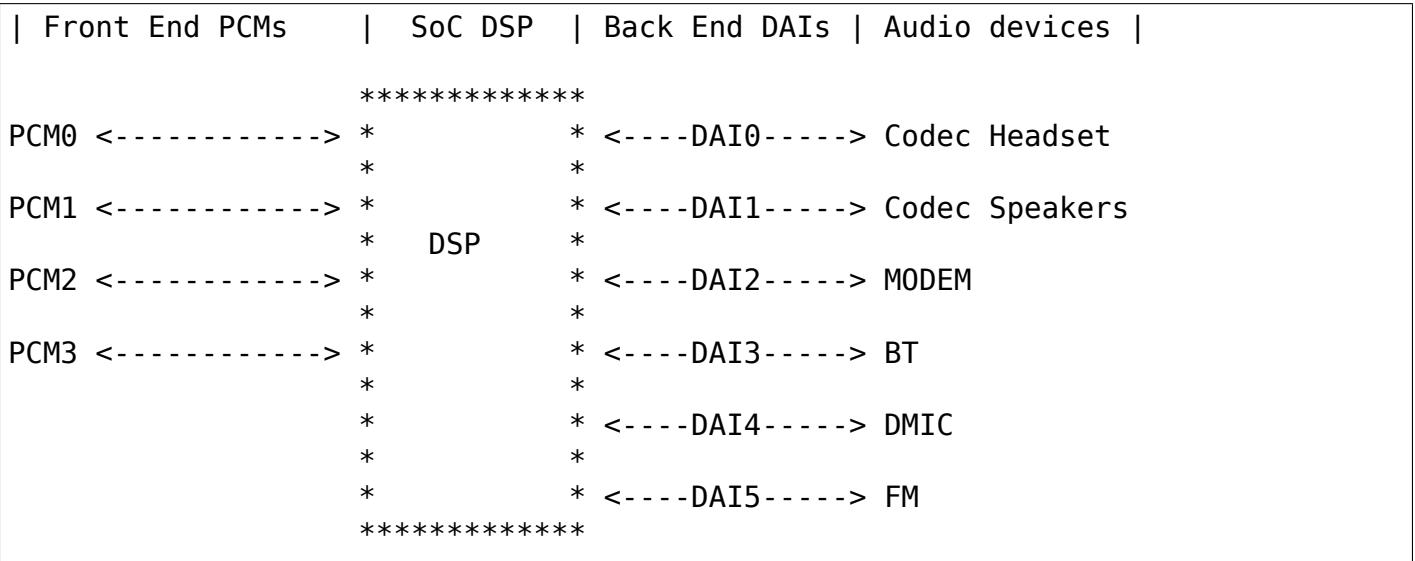
The DPCM runtime routing is determined by the ALSA mixer settings in the same way as the analog signal is routed in an ASoC codec driver. DPCM uses a DAPM graph representing the

DSP internal audio paths and uses the mixer settings to determine the path used by each ALSA PCM.

DPCM re-uses all the existing component codec, platform and DAI drivers without any modifications.

Phone Audio System with SoC based DSP

Consider the following phone audio subsystem. This will be used in this document for all examples :-

```
| Front End PCMs    |   SoC DSP  | Back End DAIs | Audio devices |

                    *************
PCM0 <------------> *           * <----DAI0-----> Codec Headset
                    *           *
PCM1 <------------> *           * <----DAI1-----> Codec Speakers
                    *    DSP    *
PCM2 <------------> *           * <----DAI2-----> MODEM
                    *           *
PCM3 <------------> *           * <----DAI3-----> BT
                    *           *
                    *           * <----DAI4-----> DMIC
                    *           *
                    *           * <----DAI5-----> FM
                    *************
```
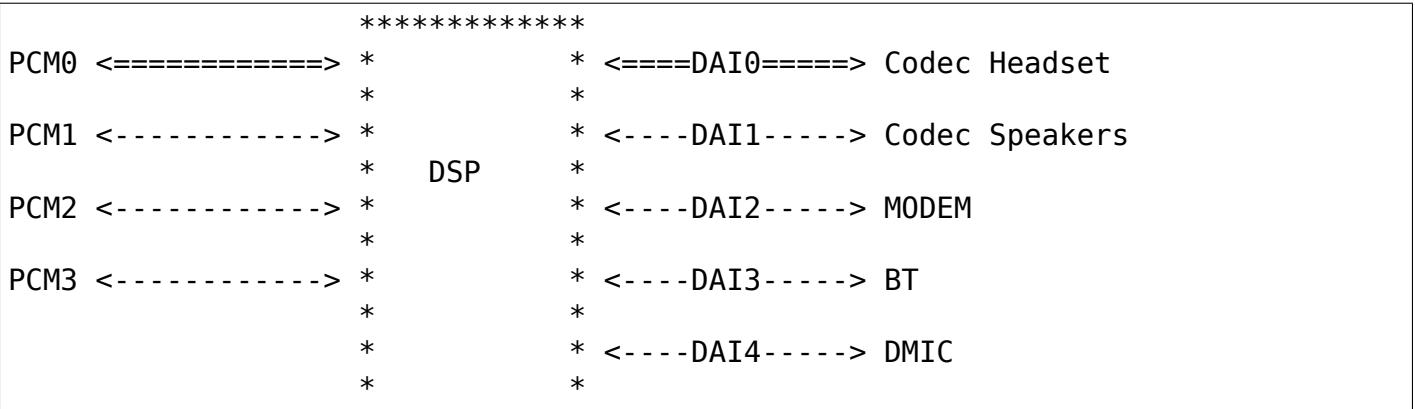
This diagram shows a simple smart phone audio subsystem. It supports Bluetooth, FM digital radio, Speakers, Headset Jack, digital microphones and cellular modem. This sound card exposes 4 DSP front end (FE) ALSA PCM devices and supports 6 back end (BE) DAIs. Each FE PCM can digitally route audio data to any of the BE DAIs. The FE PCM devices can also route audio to more than 1 BE DAI.

Example - DPCM Switching playback from DAI0 to DAI1

Audio is being played to the Headset. After a while the user removes the headset and audio continues playing on the speakers.

Playback on PCM0 to Headset would look like :-

```
                    *************
PCM0 <============> *           * <====DAI0====> Codec Headset
                    *           *
PCM1 <------------> *           * <----DAI1-----> Codec Speakers
                    *    DSP    *
PCM2 <------------> *           * <----DAI2-----> MODEM
                    *           *
PCM3 <------------> *           * <----DAI3-----> BT
                    *           *
                    *           * <----DAI4-----> DMIC
                    *           *
```

```
              *            * <----DAI5-----> FM
              *************
```

The headset is removed from the jack by user so the speakers must now be used :-

```
                   *************
PCM0 <============> *          * <----DAI0-----> Codec Headset
                   *          *
PCM1 <------------> *          * <====DAI1=====> Codec Speakers
                   *    DSP   *
PCM2 <------------> *          * <----DAI2-----> MODEM
                   *          *
PCM3 <------------> *          * <----DAI3-----> BT
                   *          *
                   *          * <----DAI4-----> DMIC
                   *          *
                   *          * <----DAI5-----> FM
                   *************
```

The audio driver processes this as follows :-

1. Machine driver receives Jack removal event.

2. Machine driver OR audio HAL disables the Headset path.

3. DPCM runs the PCM trigger(stop), hw_free(), shutdown() operations on DAI0 for headset since the path is now disabled.

4. Machine driver or audio HAL enables the speaker path.

5. DPCM runs the PCM ops for startup(), hw_params(), prepare() and trigger(start) for DAI1 Speakers since the path is enabled.
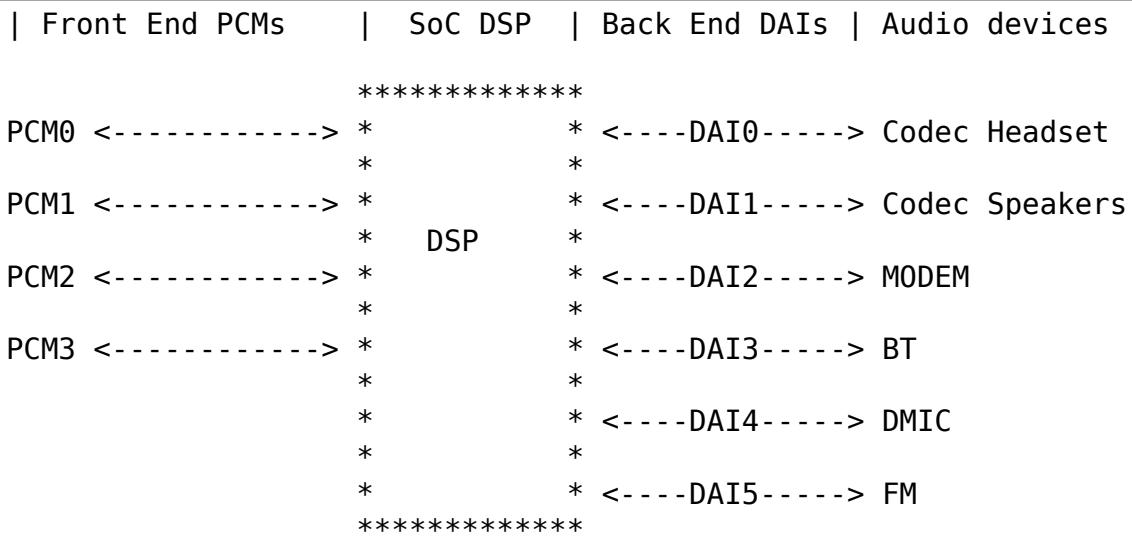
In this example, the machine driver or userspace audio HAL can alter the routing and then DPCM will take care of managing the DAI PCM operations to either bring the link up or down. Audio playback does not stop during this transition.

3.10.2 DPCM machine driver

The DPCM enabled ASoC machine driver is similar to normal machine drivers except that we also have to :-

1. Define the FE and BE DAI links.

2. Define any FE/BE PCM operations.

3. Define widget graph connections.

FE and BE DAI links

```
| Front End PCMs    |   SoC DSP  | Back End DAIs | Audio devices |

                    *************
PCM0 <------------> *          * <----DAI0-----> Codec Headset
                    *          *
PCM1 <-----------> *          * <----DAI1-----> Codec Speakers
                    *    DSP    *
PCM2 <-----------> *          * <----DAI2-----> MODEM
                    *          *
PCM3 <-----------> *          * <----DAI3-----> BT
                    *          *
                    *          * <----DAI4-----> DMIC
                    *          *
                    *          * <----DAI5-----> FM
                    *************
```

For the example above we have to define 4 FE DAI links and 6 BE DAI links. The FE DAI links are defined as follows :-

```
static struct snd_soc_dai_link machine_dais[] = {
    {
            .name = "PCM0 System",
            .stream_name = "System Playback",
            .cpu_dai_name = "System Pin",
            .platform_name = "dsp-audio",
            .codec_name = "snd-soc-dummy",
            .codec_dai_name = "snd-soc-dummy-dai",
            .dynamic = 1,
            .trigger = {SND_SOC_DPCM_TRIGGER_POST, SND_SOC_DPCM_TRIGGER_POST}
↪,
            .dpcm_playback = 1,
    },
    .....< other FE and BE DAI links here >
};
```

This FE DAI link is pretty similar to a regular DAI link except that we also set the DAI link to a DPCM FE with the `dynamic = 1`. The supported FE stream directions should also be set with the `dpcm_playback` and `dpcm_capture` flags. There is also an option to specify the ordering of the trigger call for each FE. This allows the ASoC core to trigger the DSP before or after the other components (as some DSPs have strong requirements for the ordering DAI/DSP start and stop sequences).

The FE DAI above sets the codec and code DAIs to dummy devices since the BE is dynamic and will change depending on runtime config.

The BE DAIs are configured as follows :-

```
static struct snd_soc_dai_link machine_dais[] = {
    .....< FE DAI links here >
    {
```

```
                    .name = "Codec Headset",
                    .cpu_dai_name = "ssp-dai.0",
                    .platform_name = "snd-soc-dummy",
                    .no_pcm = 1,
                    .codec_name = "rt5640.0-001c",
                    .codec_dai_name = "rt5640-aif1",
                    .ignore_suspend = 1,
                    .ignore_pmdown_time = 1,
                    .be_hw_params_fixup = hswult_ssp0_fixup,
                    .ops = &haswell_ops,
                    .dpcm_playback = 1,
                    .dpcm_capture = 1,
            },
            .....< other BE DAI links here >
};
```

This BE DAI link connects DAI0 to the codec (in this case RT5460 AIF1). It sets the `no_pcm` flag to mark it has a BE and sets flags for supported stream directions using `dpcm_playback` and `dpcm_capture` above.

The BE has also flags set for ignoring suspend and PM down time. This allows the BE to work in a hostless mode where the host CPU is not transferring data like a BT phone call :-

```
                    *************
PCM0 <------------> *           * <----DAI0-----> Codec Headset
                    *           *
PCM1 <------------> *           * <----DAI1-----> Codec Speakers
                    *    DSP    *
PCM2 <------------> *           * <====DAI2=====> MODEM
                    *           *
PCM3 <------------> *           * <====DAI3=====> BT
                    *           *
                    *           * <----DAI4-----> DMIC
                    *           *
                    *           * <----DAI5-----> FM
                    *************
```

This allows the host CPU to sleep while the DSP, MODEM DAI and the BT DAI are still in operation.

A BE DAI link can also set the codec to a dummy device if the codec is a device that is managed externally.

Likewise a BE DAI can also set a dummy cpu DAI if the CPU DAI is managed by the DSP firmware.

FE/BE PCM operations

The BE above also exports some PCM operations and a `fixup` callback. The fixup callback is used by the machine driver to (re)configure the DAI based upon the FE hw params. i.e. the DSP may perform SRC or ASRC from the FE to BE.

e.g. DSP converts all FE hw params to run at fixed rate of 48k, 16bit, stereo for DAI0. This means all FE hw_params have to be fixed in the machine driver for DAI0 so that the DAI is running at desired configuration regardless of the FE configuration.

```
static int dai0_fixup(struct snd_soc_pcm_runtime *rtd,
                      struct snd_pcm_hw_params *params)
{
      struct snd_interval *rate = hw_param_interval(params,
                      SNDRV_PCM_HW_PARAM_RATE);
      struct snd_interval *channels = hw_param_interval(params,
                                      SNDRV_PCM_HW_PARAM_CHANNELS);

      /* The DSP will convert the FE rate to 48k, stereo */
      rate->min = rate->max = 48000;
      channels->min = channels->max = 2;

      /* set DAI0 to 16 bit */
      params_set_format(params, SNDRV_PCM_FORMAT_S16_LE);
      return 0;
}
```

The other PCM operation are the same as for regular DAI links. Use as necessary.

Widget graph connections

The BE DAI links will normally be connected to the graph at initialisation time by the ASoC DAPM core. However, if the BE codec or BE DAI is a dummy then this has to be set explicitly in the driver :-

```
/* BE for codec Headset -  DAI0 is dummy and managed by DSP FW */
{"DAI0 CODEC IN", NULL, "AIF1 Capture"},
{"AIF1 Playback", NULL, "DAI0 CODEC OUT"},
```

3.10.3 Writing a DPCM DSP driver

The DPCM DSP driver looks much like a standard platform class ASoC driver combined with elements from a codec class driver. A DSP platform driver must implement :-

1. Front End PCM DAIs - i.e. struct snd_soc_dai_driver.

2. DAPM graph showing DSP audio routing from FE DAIs to BEs.

3. DAPM widgets from DSP graph.

4. Mixers for gains, routing, etc.

5. DMA configuration.

6. BE AIF widgets.

Items 6 is important for routing the audio outside of the DSP. AIF need to be defined for each BE and each stream direction. e.g for BE DAI0 above we would have :-

```
SND_SOC_DAPM_AIF_IN("DAI0 RX", NULL, 0, SND_SOC_NOPM, 0, 0),
SND_SOC_DAPM_AIF_OUT("DAI0 TX", NULL, 0, SND_SOC_NOPM, 0, 0),
```

The BE AIF are used to connect the DSP graph to the graphs for the other component drivers (e.g. codec graph).

3.10.4 Hostless PCM streams

A hostless PCM stream is a stream that is not routed through the host CPU. An example of this would be a phone call from handset to modem.

```
                        ************
PCM0 <------------> *          * <----DAI0-----> Codec Headset
                    *          *
PCM1 <----------->  *          * <====DAI1=====> Codec Speakers/Mic
                    *    DSP    *
PCM2 <------------> *          * <====DAI2=====> MODEM
                    *          *
PCM3 <------------> *          * <----DAI3-----> BT
                    *          *
                    *          * <----DAI4-----> DMIC
                    *          *
                    *          * <----DAI5-----> FM
                        ************
```

In this case the PCM data is routed via the DSP. The host CPU in this use case is only used for control and can sleep during the runtime of the stream.

The host can control the hostless link either by :-

1. Configuring the link as a CODEC <-> CODEC style link. In this case the link is enabled or disabled by the state of the DAPM graph. This usually means there is a mixer control that can be used to connect or disconnect the path between both DAIs.

2. Hostless FE. This FE has a virtual connection to the BE DAI links on the DAPM graph. Control is then carried out by the FE as regular PCM operations. This method gives more control over the DAI links, but requires much more userspace code to control the link. Its recommended to use CODEC<->CODEC unless your HW needs more fine grained sequencing of the PCM ops.

CODEC <-> CODEC link

This DAI link is enabled when DAPM detects a valid path within the DAPM graph. The machine driver sets some additional parameters to the DAI link i.e.

```
static const struct snd_soc_pcm_stream dai_params = {
        .formats = SNDRV_PCM_FMTBIT_S32_LE,
        .rate_min = 8000,
        .rate_max = 8000,
        .channels_min = 2,
        .channels_max = 2,
};

static struct snd_soc_dai_link dais[] = {
        < ... more DAI links above ... >
        {
                .name = "MODEM",
                .stream_name = "MODEM",
                .cpu_dai_name = "dai2",
                .codec_dai_name = "modem-aif1",
                .codec_name = "modem",
                .dai_fmt = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF
                                | SND_SOC_DAIFMT_CBM_CFM,
                .c2c_params = &dai_params,
                .num_c2c_params = 1,
        }
        < ... more DAI links here ... >
```
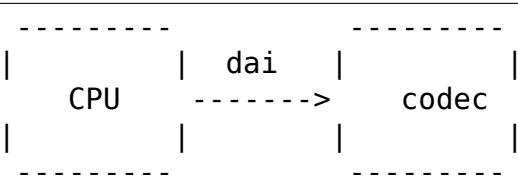
These parameters are used to configure the DAI hw_params() when DAPM detects a valid path and then calls the PCM operations to start the link. DAPM will also call the appropriate PCM operations to disable the DAI when the path is no longer valid.

Hostless FE
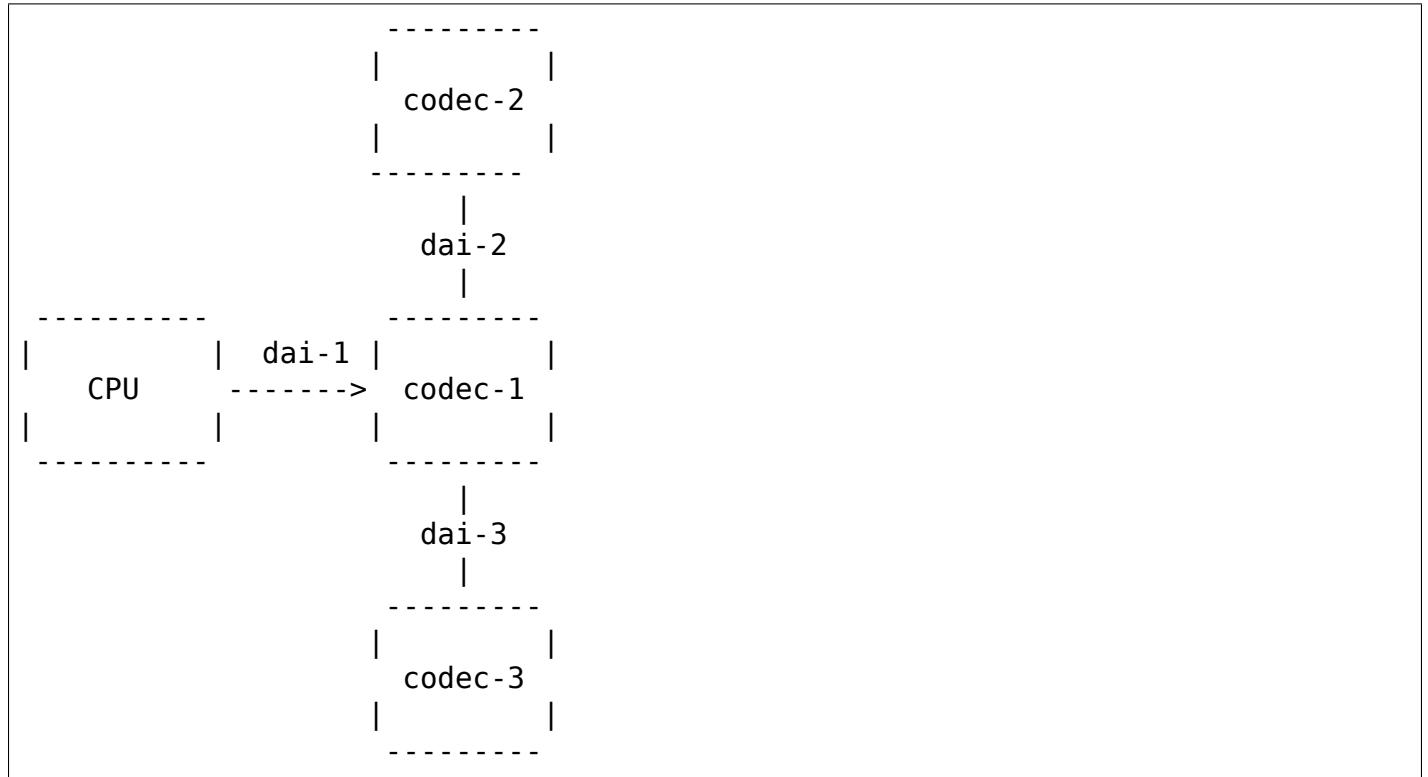
The DAI link(s) are enabled by a FE that does not read or write any PCM data. This means creating a new FE that is connected with a virtual path to both DAI links. The DAI links will be started when the FE PCM is started and stopped when the FE PCM is stopped. Note that the FE PCM cannot read or write data in this configuration.

3.11 Creating codec to codec dai link for ALSA dapm

Mostly the flow of audio is always from CPU to codec so your system will look as below:

```
 ---------              ---------
|         |   dai   |           |
    CPU      ------->     codec
|         |         |           |
 ---------              ---------
```

In case your system looks as below:

```
                          ---------
                      |           |
                        codec-2
                      |           |
                       ---------
                           |
                         dai-2
                           |
  ----------           ---------
 |          | dai-1 |           |
      CPU      ------->  codec-1
 |          |       |           |
  ----------           ---------
                           |
                         dai-3
                           |
                       ---------
                      |           |
                        codec-3
                      |           |
                       ---------
```

Suppose codec-2 is a bluetooth chip and codec-3 is connected to a speaker and you have a below scenario: codec-2 will receive the audio data and the user wants to play that audio through codec-3 without involving the CPU.This aforementioned case is the ideal case when codec to codec connection should be used.

Your dai_link should appear as below in your machine file:

```
/*
 * this pcm stream only supports 24 bit, 2 channel and
 * 48k sampling rate.
 */
static const struct snd_soc_pcm_stream dsp_codec_params = {
        .formats = SNDRV_PCM_FMTBIT_S24_LE,
        .rate_min = 48000,
        .rate_max = 48000,
        .channels_min = 2,
        .channels_max = 2,
};

{
    .name = "CPU-DSP",
    .stream_name = "CPU-DSP",
    .cpu_dai_name = "samsung-i2s.0",
    .codec_name = "codec-2,
    .codec_dai_name = "codec-2-dai_name",
    .platform_name = "samsung-i2s.0",
    .dai_fmt = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF
            | SND_SOC_DAIFMT_CBM_CFM,
    .ignore_suspend = 1,
```

```
    .c2c_params = &dsp_codec_params,
    .num_c2c_params = 1,
},
{
    .name = "DSP-CODEC",
    .stream_name = "DSP-CODEC",
    .cpu_dai_name = "wm0010-sdi2",
    .codec_name = "codec-3,
    .codec_dai_name = "codec-3-dai_name",
    .dai_fmt = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF
            | SND_SOC_DAIFMT_CBM_CFM,
    .ignore_suspend = 1,
    .c2c_params = &dsp_codec_params,
    .num_c2c_params = 1,
},
```

Above code snippet is motivated from sound/soc/samsung/speyside.c.

Note the "c2c_params" callback which lets the dapm know that this dai_link is a codec to codec connection.

In dapm core a route is created between cpu_dai playback widget and codec_dai capture widget for playback path and vice-versa is true for capture path. In order for this aforementioned route to get triggered, DAPM needs to find a valid endpoint which could be either a sink or source widget corresponding to playback and capture path respectively.

In order to trigger this dai_link widget, a thin codec driver for the speaker amp can be created as demonstrated in wm8727.c file, it sets appropriate constraints for the device even if it needs no control.

Make sure to name your corresponding cpu and codec playback and capture dai names ending with "Playback" and "Capture" respectively as dapm core will link and power those dais based on the name.

A dai_link in a "simple-audio-card" will automatically be detected as codec to codec when all DAIs on the link belong to codec components. The dai_link will be initialized with the subset of stream parameters (channels, format, sample rate) supported by all DAIs on the link. Since there is no way to provide these parameters in the device tree, this is mostly useful for communication with simple fixed-function codecs, such as a Bluetooth controller or cellular modem.

ADVANCED LINUX SOUND ARCHITECTURE - DRIVER CONFIGURATION GUIDE

4.1 Kernel Configuration

To enable ALSA support you need at least to build the kernel with primary sound card support (`CONFIG_SOUND`). Since ALSA can emulate OSS, you don't have to choose any of the OSS modules.

Enable "OSS API emulation" (`CONFIG_SND_OSSEMUL`) and both OSS mixer and PCM supports if you want to run OSS applications with ALSA.

If you want to support the WaveTable functionality on cards such as SB Live! then you need to enable "Sequencer support" (`CONFIG_SND_SEQUENCER`).

To make ALSA debug messages more verbose, enable the "Verbose printk" and "Debug" options. To check for memory leaks, turn on "Debug memory" too. "Debug detection" will add checks for the detection of cards.

Please note that all the ALSA ISA drivers support the Linux isapnp API (if the card supports ISA PnP). You don't need to configure the cards using isapnptools.

4.2 Module parameters

The user can load modules with options. If the module supports more than one card and you have more than one card of the same type then you can specify multiple values for the option separated by commas.

4.2.1 Module snd

The core ALSA module. It is used by all ALSA card drivers. It takes the following options which have global effects.

major
> major number for sound driver; Default: 116

cards_limit
> limiting card index for auto-loading (1-8); Default: 1; For auto-loading more than one card, specify this option together with snd-card-X aliases.

slots
> Reserve the slot index for the given driver; This option takes multiple strings. See *Module Autoloading Support* section for details.

debug

Specifies the debug message level; (0 = disable debug prints, 1 = normal debug messages, 2 = verbose debug messages); This option appears only when `CONFIG_SND_DEBUG=y`. This option can be dynamically changed via sysfs /sys/modules/snd/parameters/debug file.

4.2.2 Module snd-pcm-oss

The PCM OSS emulation module. This module takes options which change the mapping of devices.

dsp_map

PCM device number maps assigned to the 1st OSS device; Default: 0

adsp_map

PCM device number maps assigned to the 2nd OSS device; Default: 1

nonblock_open

Don't block opening busy PCM devices; Default: 1

For example, when `dsp_map=2`, /dev/dsp will be mapped to PCM #2 of the card #0. Similarly, when `adsp_map=0`, /dev/adsp will be mapped to PCM #0 of the card #0. For changing the second or later card, specify the option with commas, such like `dsp_map=0,1`.

`nonblock_open` option is used to change the behavior of the PCM regarding opening the device. When this option is non-zero, opening a busy OSS PCM device won't be blocked but return immediately with EAGAIN (just like O_NONBLOCK flag).

4.2.3 Module snd-rawmidi

This module takes options which change the mapping of devices. similar to those of the snd-pcm-oss module.

midi_map

MIDI device number maps assigned to the 1st OSS device; Default: 0

amidi_map

MIDI device number maps assigned to the 2nd OSS device; Default: 1

4.2.4 Module snd-soc-core

The soc core module. It is used by all ALSA card drivers. It takes the following options which have global effects.

prealloc_buffer_size_kbytes

Specify prealloc buffer size in kbytes (default: 512).

4.2.5 Common parameters for top sound card modules

Each of top level sound card module takes the following options.

index
> index (slot #) of sound card; Values: 0 through 31 or negative; If nonnegative, assign that index number; if negative, interpret as a bitmask of permissible indices; the first free permitted index is assigned; Default: -1

id
> card ID (identifier or name); Can be up to 15 characters long; Default: the card type; A directory by this name is created under /proc/asound/ containing information about the card; This ID can be used instead of the index number in identifying the card

enable
> enable card; Default: enabled, for PCI and ISA PnP cards

These options are used for either specifying the order of instances or controlling enabling and disabling of each one of the devices if there are multiple devices bound with the same driver. For example, there are many machines which have two HD-audio controllers (one for HDMI/DP audio and another for onboard analog). In most cases, the second one is in primary usage, and people would like to assign it as the first appearing card. They can do it by specifying "index=1,0" module parameter, which will swap the assignment slots.

Today, with the sound backend like PulseAudio and PipeWire which supports dynamic configuration, it's of little use, but that was a help for static configuration in the past.

4.2.6 Module snd-adlib

Module for AdLib FM cards.

port
> port # for OPL chip

This module supports multiple cards. It does not support autoprobe, so the port must be specified. For actual AdLib FM cards it will be 0x388. Note that this card does not have PCM support and no mixer; only FM synthesis.

Make sure you have `sbiload` from the alsa-tools package available and, after loading the module, find out the assigned ALSA sequencer port number through `sbiload -l`.

Example output:

```
Port       Client name                      Port name
64:0       OPL2 FM synth                    OPL2 FM Port
```

Load the `std.sb` and `drums.sb` patches also supplied by `sbiload`:

```
sbiload -p 64:0 std.sb drums.sb
```

If you use this driver to drive an OPL3, you can use `std.o3` and `drums.o3` instead. To have the card produce sound, use `aplaymidi` from alsa-utils:

```
aplaymidi -p 64:0 foo.mid
```

4.2.7 Module snd-ad1816a

Module for sound cards based on Analog Devices AD1816A/AD1815 ISA chips.

clockfreq
    Clock frequency for AD1816A chip (default = 0, 33000Hz)

This module supports multiple cards, autoprobe and PnP.

4.2.8 Module snd-ad1848

Module for sound cards based on AD1848/AD1847/CS4248 ISA chips.

port
    port # for AD1848 chip

irq
    IRQ # for AD1848 chip

dma1
    DMA # for AD1848 chip (0,1,3)

This module supports multiple cards. It does not support autoprobe thus main port must be specified!!! Other ports are optional.

The power-management is supported.

4.2.9 Module snd-ad1889

Module for Analog Devices AD1889 chips.

ac97_quirk
    AC'97 workaround for strange hardware; See the description of intel8x0 module for details.

This module supports multiple cards.

4.2.10 Module snd-ali5451

Module for ALi M5451 PCI chip.

pcm_channels
    Number of hardware channels assigned for PCM

spdif
    Support SPDIF I/O; Default: disabled

This module supports one chip and autoprobe.

The power-management is supported.

4.2.11 Module snd-als100

Module for sound cards based on Avance Logic ALS100/ALS120 ISA chips.

This module supports multiple cards, autoprobe and PnP.

The power-management is supported.

4.2.12 Module snd-als300

Module for Avance Logic ALS300 and ALS300+

This module supports multiple cards.

The power-management is supported.

4.2.13 Module snd-als4000

Module for sound cards based on Avance Logic ALS4000 PCI chip.

joystick_port
> port # for legacy joystick support; 0 = disabled (default), 1 = auto-detect

This module supports multiple cards, autoprobe and PnP.

The power-management is supported.

4.2.14 Module snd-asihpi

Module for AudioScience ASI soundcards

enable_hpi_hwdep
> enable HPI hwdep for AudioScience soundcard

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.15 Module snd-atiixp

Module for ATI IXP 150/200/250/400 AC97 controllers.

ac97_clock
> AC'97 clock (default = 48000)

ac97_quirk
> AC'97 workaround for strange hardware; See *AC97 Quirk Option* section below.

ac97_codec
> Workaround to specify which AC'97 codec instead of probing. If this works for you file a bug with your *lspci -vn* output. (-2 = Force probing, -1 = Default behavior, 0-2 = Use the specified codec.)

spdif_aclink
> S/PDIF transfer over AC-link (default = 1)

This module supports one card and autoprobe.

ATI IXP has two different methods to control SPDIF output. One is over AC-link and another is over the "direct" SPDIF output. The implementation depends on the motherboard, and you'll need to choose the correct one via spdif_aclink module option.

The power-management is supported.

4.2.16 Module snd-atiixp-modem

Module for ATI IXP 150/200/250 AC97 modem controllers.

This module supports one card and autoprobe.

Note: The default index value of this module is -2, i.e. the first slot is excluded.

The power-management is supported.

4.2.17 Module snd-au8810, snd-au8820, snd-au8830

Module for Aureal Vortex, Vortex2 and Advantage device.

pcifix
    Control PCI workarounds; 0 = Disable all workarounds, 1 = Force the PCI latency of the Aureal card to 0xff, 2 = Force the Extend PCI#2 Internal Master for Efficient Handling of Dummy Requests on the VIA KT133 AGP Bridge, 3 = Force both settings, 255 = Autodetect what is required (default)

This module supports all ADB PCM channels, ac97 mixer, SPDIF, hardware EQ, mpu401, gameport. A3D and wavetable support are still in development. Development and reverse engineering work is being coordinated at https://savannah.nongnu.org/projects/openvortex/ SPDIF output has a copy of the AC97 codec output, unless you use the spdif pcm device, which allows raw data passthru. The hardware EQ hardware and SPDIF is only present in the Vortex2 and Advantage.

Note: Some ALSA mixer applications don't handle the SPDIF sample rate control correctly. If you have problems regarding this, try another ALSA compliant mixer (alsamixer works).

4.2.18 Module snd-azt1605

Module for Aztech Sound Galaxy soundcards based on the Aztech AZT1605 chipset.

port
    port # for BASE (0x220,0x240,0x260,0x280)

wss_port
    port # for WSS (0x530,0x604,0xe80,0xf40)

irq
    IRQ # for WSS (7,9,10,11)

dma1
    DMA # for WSS playback (0,1,3)

dma2
    DMA # for WSS capture (0,1), -1 = disabled (default)

mpu_port
> port # for MPU-401 UART (0x300,0x330), -1 = disabled (default)

mpu_irq
> IRQ # for MPU-401 UART (3,5,7,9), -1 = disabled (default)

fm_port
> port # for OPL3 (0x388), -1 = disabled (default)

This module supports multiple cards. It does not support autoprobe: `port`, `wss_port`, `irq` and `dma1` have to be specified. The other values are optional.

`port` needs to match the BASE ADDRESS jumper on the card (0x220 or 0x240) or the value stored in the card's EEPROM for cards that have an EEPROM and their "CONFIG MODE" jumper set to "EEPROM SETTING". The other values can be chosen freely from the options enumerated above.

If `dma2` is specified and different from `dma1`, the card will operate in full-duplex mode. When `dma1=3`, only `dma2=0` is valid and the only way to enable capture since only channels 0 and 1 are available for capture.

Generic settings are `port=0x220 wss_port=0x530 irq=10 dma1=1 dma2=0 mpu_port=0x330 mpu_irq=9 fm_port=0x388`.

Whatever IRQ and DMA channels you pick, be sure to reserve them for legacy ISA in your BIOS.

4.2.19 Module snd-azt2316

Module for Aztech Sound Galaxy soundcards based on the Aztech AZT2316 chipset.

port
> port # for BASE (0x220,0x240,0x260,0x280)

wss_port
> port # for WSS (0x530,0x604,0xe80,0xf40)

irq
> IRQ # for WSS (7,9,10,11)

dma1
> DMA # for WSS playback (0,1,3)

dma2
> DMA # for WSS capture (0,1), -1 = disabled (default)

mpu_port
> port # for MPU-401 UART (0x300,0x330), -1 = disabled (default)

mpu_irq
> IRQ # for MPU-401 UART (5,7,9,10), -1 = disabled (default)

fm_port
> port # for OPL3 (0x388), -1 = disabled (default)

This module supports multiple cards. It does not support autoprobe: `port`, `wss_port`, `irq` and `dma1` have to be specified. The other values are optional.

`port` needs to match the BASE ADDRESS jumper on the card (0x220 or 0x240) or the value stored in the card's EEPROM for cards that have an EEPROM and their "CONFIG MODE"

jumper set to "EEPROM SETTING". The other values can be chosen freely from the options enumerated above.

If `dma2` is specified and different from `dma1`, the card will operate in full-duplex mode. When `dma1=3`, only `dma2=0` is valid and the only way to enable capture since only channels 0 and 1 are available for capture.

Generic settings are `port=0x220 wss_port=0x530 irq=10 dma1=1 dma2=0 mpu_port=0x330 mpu_irq=9 fm_port=0x388`.

Whatever IRQ and DMA channels you pick, be sure to reserve them for legacy ISA in your BIOS.

4.2.20 Module snd-aw2

Module for Audiowerk2 sound card

This module supports multiple cards.

4.2.21 Module snd-azt2320

Module for sound cards based on Aztech System AZT2320 ISA chip (PnP only).

This module supports multiple cards, PnP and autoprobe.

The power-management is supported.

4.2.22 Module snd-azt3328

Module for sound cards based on Aztech AZF3328 PCI chip.

joystick
    Enable joystick (default off)

This module supports multiple cards.

4.2.23 Module snd-bt87x

Module for video cards based on Bt87x chips.

digital_rate
    Override the default digital rate (Hz)

load_all
    Load the driver even if the card model isn't known

This module supports multiple cards.

Note: The default index value of this module is -2, i.e. the first slot is excluded.

4.2.24 Module snd-ca0106

Module for Creative Audigy LS and SB Live 24bit

This module supports multiple cards.

4.2.25 Module snd-cmi8330

Module for sound cards based on C-Media CMI8330 ISA chips.

isapnp
   ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

wssport
   port # for CMI8330 chip (WSS)

wssirq
   IRQ # for CMI8330 chip (WSS)

wssdma
   first DMA # for CMI8330 chip (WSS)

sbport
   port # for CMI8330 chip (SB16)

sbirq
   IRQ # for CMI8330 chip (SB16)

sbdma8
   8bit DMA # for CMI8330 chip (SB16)

sbdma16
   16bit DMA # for CMI8330 chip (SB16)

fmport
   (optional) OPL3 I/O port

mpuport
   (optional) MPU401 I/O port

mpuirq
   (optional) MPU401 irq #

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.26 Module snd-cmipci

Module for C-Media CMI8338/8738/8768/8770 PCI sound cards.

mpu_port
    port address of MIDI interface (8338 only): 0x300,0x310,0x320,0x330 = legacy port, 1 = integrated PCI port (default on 8738), 0 = disable

fm_port
    port address of OPL-3 FM synthesizer (8x38 only): 0x388 = legacy port, 1 = integrated PCI port (default on 8738), 0 = disable

soft_ac3
    Software-conversion of raw SPDIF packets (model 033 only) (default = 1)

joystick_port
    Joystick port address (0 = disable, 1 = auto-detect)

This module supports autoprobe and multiple cards.

The power-management is supported.

4.2.27 Module snd-cs4231

Module for sound cards based on CS4231 ISA chips.

port
    port # for CS4231 chip

mpu_port
    port # for MPU-401 UART (optional), -1 = disable

irq
    IRQ # for CS4231 chip

mpu_irq
    IRQ # for MPU-401 UART

dma1
    first DMA # for CS4231 chip

dma2
    second DMA # for CS4231 chip

This module supports multiple cards. This module does not support autoprobe thus main port must be specified!!! Other ports are optional.

The power-management is supported.

4.2.28 Module snd-cs4236

Module for sound cards based on CS4232/CS4232A, CS4235/CS4236/CS4236B/CS4237B/CS4238B/CS
ISA chips.

isapnp
    ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port
    port # for CS4236 chip (PnP setup - 0x534)

cport
    control port # for CS4236 chip (PnP setup - 0x120,0x210,0xf00)

mpu_port
    port # for MPU-401 UART (PnP setup - 0x300), -1 = disable

fm_port
    FM port # for CS4236 chip (PnP setup - 0x388), -1 = disable

irq
    IRQ # for CS4236 chip (5,7,9,11,12,15)

mpu_irq
    IRQ # for MPU-401 UART (9,11,12,15)

dma1
    first DMA # for CS4236 chip (0,1,3)

dma2
    second DMA # for CS4236 chip (0,1,3), -1 = disable

This module supports multiple cards. This module does not support autoprobe (if ISA PnP is not used) thus main port and control port must be specified!!! Other ports are optional.

The power-management is supported.

This module is aliased as snd-cs4232 since it provides the old snd-cs4232 functionality, too.

4.2.29 Module snd-cs4281

Module for Cirrus Logic CS4281 soundchip.

dual_codec
    Secondary codec ID (0 = disable, default)

This module supports multiple cards.

The power-management is supported.

4.2.30 Module snd-cs46xx

Module for PCI sound cards based on CS4610/CS4612/CS4614/CS4615/CS4622/CS4624/CS4630/CS4280 PCI chips.

external_amp
> Force to enable external amplifier.

thinkpad
> Force to enable Thinkpad's CLKRUN control.

mmap_valid
> Support OSS mmap mode (default = 0).

This module supports multiple cards and autoprobe. Usually external amp and CLKRUN controls are detected automatically from PCI sub vendor/device ids. If they don't work, give the options above explicitly.

The power-management is supported.

4.2.31 Module snd-cs5530

Module for Cyrix/NatSemi Geode 5530 chip.

4.2.32 Module snd-cs5535audio

Module for multifunction CS5535 companion PCI device

The power-management is supported.

4.2.33 Module snd-ctxfi

Module for Creative Sound Blaster X-Fi boards (20k1 / 20k2 chips)

- Creative Sound Blaster X-Fi Titanium Fatal1ty Champion Series
- Creative Sound Blaster X-Fi Titanium Fatal1ty Professional Series
- Creative Sound Blaster X-Fi Titanium Professional Audio
- Creative Sound Blaster X-Fi Titanium
- Creative Sound Blaster X-Fi Elite Pro
- Creative Sound Blaster X-Fi Platinum
- Creative Sound Blaster X-Fi Fatal1ty
- Creative Sound Blaster X-Fi XtremeGamer
- Creative Sound Blaster X-Fi XtremeMusic

reference_rate
> reference sample rate, 44100 or 48000 (default)

multiple
> multiple to ref. sample rate, 1 or 2 (default)

subsystem
> override the PCI SSID for probing; the value consists of SSVID << 16 | SSDID. The default is zero, which means no override.

This module supports multiple cards.

4.2.34 Module snd-darla20

Module for Echoaudio Darla20

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.35 Module snd-darla24

Module for Echoaudio Darla24

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.36 Module snd-dt019x

Module for Diamond Technologies DT-019X / Avance Logic ALS-007 (PnP only)

This module supports multiple cards. This module is enabled only with ISA PnP support.

The power-management is supported.

4.2.37 Module snd-dummy

Module for the dummy sound card. This "card" doesn't do any output or input, but you may use this module for any application which requires a sound card (like RealPlayer).

pcm_devs
> Number of PCM devices assigned to each card (default = 1, up to 4)

pcm_substreams
> Number of PCM substreams assigned to each PCM (default = 8, up to 128)

hrtimer
> Use hrtimer (=1, default) or system timer (=0)

fake_buffer
> Fake buffer allocations (default = 1)

When multiple PCM devices are created, snd-dummy gives different behavior to each PCM device: * 0 = interleaved with mmap support * 1 = non-interleaved with mmap support * 2 = interleaved without mmap * 3 = non-interleaved without mmap

As default, snd-dummy drivers doesn't allocate the real buffers but either ignores read/write or mmap a single dummy page to all buffer pages, in order to save the resources. If your apps need the read/ written buffer data to be consistent, pass fake_buffer=0 option.

The power-management is supported.

4.2.38 Module snd-echo3g

Module for Echoaudio 3G cards (Gina3G/Layla3G)

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.39 Module snd-emu10k1

Module for EMU10K1/EMU10k2 based PCI sound cards.

- Sound Blaster Live!
- Sound Blaster PCI 512
- Sound Blaster Audigy
- E-MU APS (partially supported)
- E-MU DAS

extin
bitmap of available external inputs for FX8010 (see below)

extout
bitmap of available external outputs for FX8010 (see below)

seq_ports
allocated sequencer ports (4 by default)

max_synth_voices
limit of voices used for wavetable (64 by default)

max_buffer_size
specifies the maximum size of wavetable/pcm buffers given in MB unit. Default value is 128.

enable_ir
enable IR

This module supports multiple cards and autoprobe.

Input & Output configurations [extin/extout] * Creative Card wo/Digital out [0x0003/0x1f03] * Creative Card w/Digital out [0x0003/0x1f0f] * Creative Card w/Digital CD in [0x000f/0x1f0f] * Creative Card wo/Digital out + LiveDrive [0x3fc3/0x1fc3] * Creative Card w/Digital out + LiveDrive [0x3fc3/0x1fcf] * Creative Card w/Digital CD in + LiveDrive [0x3fcf/0x1fcf] * Creative Card wo/Digital out + Digital I/O 2 [0x0fc3/0x1f0f] * Creative Card w/Digital out + Digital I/O 2 [0x0fc3/0x1f0f] * Creative Card w/Digital CD in + Digital I/O 2 [0x0fcf/0x1f0f] * Creative Card 5.1/w Digital out + LiveDrive [0x3fc3/0x1fff] * Creative Card 5.1 (c) 2003 [0x3fc3/0x7cff] * Creative Card all ins and outs [0x3fff/0x7fff]

The power-management is supported.

4.2.40 Module snd-emu10k1x

Module for Creative Emu10k1X (SB Live Dell OEM version)

This module supports multiple cards.

4.2.41 Module snd-ens1370

Module for Ensoniq AudioPCI ES1370 PCI sound cards.

- SoundBlaster PCI 64
- SoundBlaster PCI 128

joystick
    Enable joystick (default off)

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.42 Module snd-ens1371

Module for Ensoniq AudioPCI ES1371 PCI sound cards.

- SoundBlaster PCI 64
- SoundBlaster PCI 128
- SoundBlaster Vibra PCI

joystick_port
    port # for joystick (0x200,0x208,0x210,0x218), 0 = disable (default), 1 = auto-detect

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.43 Module snd-es1688

Module for ESS AudioDrive ES-1688 and ES-688 sound cards.

isapnp
    ISA PnP detection - 0 = disable, 1 = enable (default)

mpu_port
    port # for MPU-401 port (0x300,0x310,0x320,0x330), -1 = disable (default)

mpu_irq
    IRQ # for MPU-401 port (5,7,9,10)

fm_port
    port # for OPL3 (option; share the same port as default)

with `isapnp=0`, the following additional options are available:

port
    port # for ES-1688 chip (0x220,0x240,0x260)

---

irq
    IRQ # for ES-1688 chip (5,7,9,10)

dma8
    DMA # for ES-1688 chip (0,1,3)

This module supports multiple cards and autoprobe (without MPU-401 port) and PnP with the ES968 chip.

4.2.44 Module snd-es18xx

Module for ESS AudioDrive ES-18xx sound cards.

isapnp
    ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port
    port # for ES-18xx chip (0x220,0x240,0x260)

mpu_port
    port # for MPU-401 port (0x300,0x310,0x320,0x330), -1 = disable (default)

fm_port
    port # for FM (optional, not used)

irq
    IRQ # for ES-18xx chip (5,7,9,10)

dma1
    first DMA # for ES-18xx chip (0,1,3)

dma2
    first DMA # for ES-18xx chip (0,1,3)

This module supports multiple cards, ISA PnP and autoprobe (without MPU-401 port if native ISA PnP routines are not used). When `dma2` is equal with `dma1`, the driver works as half-duplex.

The power-management is supported.

4.2.45 Module snd-es1938

Module for sound cards based on ESS Solo-1 (ES1938,ES1946) chips.

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.46 Module snd-es1968

Module for sound cards based on ESS Maestro-1/2/2E (ES1968/ES1978) chips.

total_bufsize
    total buffer size in kB (1-4096kB)

pcm_substreams_p
    playback channels (1-8, default=2)

pcm_substreams_c
    capture channels (1-8, default=0)

clock
    clock (0 = auto-detection)

use_pm
    support the power-management (0 = off, 1 = on, 2 = auto (default))

enable_mpu
    enable MPU401 (0 = off, 1 = on, 2 = auto (default))

joystick
    enable joystick (default off)

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.47 Module snd-fm801

Module for ForteMedia FM801 based PCI sound cards.

tea575x_tuner
    Enable TEA575x tuner; 1 = MediaForte 256-PCS, 2 = MediaForte 256-PCPR, 3 = MediaForte 64-PCR High 16-bits are video (radio) device number + 1; example: 0x10002 (MediaForte 256-PCPR, device 1)

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.48 Module snd-gina20

Module for Echoaudio Gina20

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.49 Module snd-gina24

Module for Echoaudio Gina24

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.50 Module snd-gusclassic

Module for Gravis UltraSound Classic sound card.

port
      port # for GF1 chip (0x220,0x230,0x240,0x250,0x260)

irq
      IRQ # for GF1 chip (3,5,9,11,12,15)

dma1
      DMA # for GF1 chip (1,3,5,6,7)

dma2
      DMA # for GF1 chip (1,3,5,6,7,-1=disable)

joystick_dac
      0 to 31, (0.59V-4.52V or 0.389V-2.98V)

voices
      GF1 voices limit (14-32)

pcm_voices
      reserved PCM voices

This module supports multiple cards and autoprobe.

4.2.51 Module snd-gusextreme

Module for Gravis UltraSound Extreme (Synergy ViperMax) sound card.

port
      port # for ES-1688 chip (0x220,0x230,0x240,0x250,0x260)

gf1_port
      port # for GF1 chip (0x210,0x220,0x230,0x240,0x250,0x260,0x270)

mpu_port
      port # for MPU-401 port (0x300,0x310,0x320,0x330), -1 = disable

irq
      IRQ # for ES-1688 chip (5,7,9,10)

gf1_irq
      IRQ # for GF1 chip (3,5,9,11,12,15)

mpu_irq
      IRQ # for MPU-401 port (5,7,9,10)

dma8
      DMA # for ES-1688 chip (0,1,3)

dma1
> DMA # for GF1 chip (1,3,5,6,7)

joystick_dac
> 0 to 31, (0.59V-4.52V or 0.389V-2.98V)

voices
> GF1 voices limit (14-32)

pcm_voices
> reserved PCM voices

This module supports multiple cards and autoprobe (without MPU-401 port).

4.2.52 Module snd-gusmax

Module for Gravis UltraSound MAX sound card.

port
> port # for GF1 chip (0x220,0x230,0x240,0x250,0x260)

irq
> IRQ # for GF1 chip (3,5,9,11,12,15)

dma1
> DMA # for GF1 chip (1,3,5,6,7)

dma2
> DMA # for GF1 chip (1,3,5,6,7,-1=disable)

joystick_dac
> 0 to 31, (0.59V-4.52V or 0.389V-2.98V)

voices
> GF1 voices limit (14-32)

pcm_voices
> reserved PCM voices

This module supports multiple cards and autoprobe.

4.2.53 Module snd-hda-intel

Module for Intel HD Audio (ICH6, ICH6M, ESB2, ICH7, ICH8, ICH9, ICH10, PCH, SCH), ATI SB450, SB600, R600, RS600, RS690, RS780, RV610, RV620, RV630, RV635, RV670, RV770, VIA VT8251/VT8237A, SIS966, ULI M5461

[Multiple options for each card instance]

model
> force the model name

position_fix
> Fix DMA pointer; -1 = system default: choose appropriate one per controller hardware, 0 = auto: falls back to LPIB when POSBUF doesn't work, 1 = use LPIB, 2 = POSBUF: use position buffer, 3 = VIACOMBO: VIA-specific workaround for capture, 4 = COMBO: use LPIB for playback, auto for capture stream 5 = SKL+: apply the delay calculation available

on recent Intel chips 6 = FIFO: correct the position with the fixed FIFO size, for recent AMD chips

probe_mask
Bitmask to probe codecs (default = -1, meaning all slots); When the bit 8 (0x100) is set, the lower 8 bits are used as the "fixed" codec slots; i.e. the driver probes the slots regardless what hardware reports back

probe_only
Only probing and no codec initialization (default=off); Useful to check the initial codec status for debugging

bdl_pos_adj
Specifies the DMA IRQ timing delay in samples. Passing -1 will make the driver to choose the appropriate value based on the controller chip.

patch
Specifies the early "patch" files to modify the HD-audio setup before initializing the codecs. This option is available only when `CONFIG_SND_HDA_PATCH_LOADER=y` is set. See *More Notes on HD-Audio Driver* for details.

beep_mode
Selects the beep registration mode (0=off, 1=on); default value is set via `CONFIG_SND_HDA_INPUT_BEEP_MODE` kconfig.

[Single (global) options]

single_cmd
Use single immediate commands to communicate with codecs (for debugging only)

enable_msi
Enable Message Signaled Interrupt (MSI) (default = off)

power_save
Automatic power-saving timeout (in second, 0 = disable)

power_save_controller
Reset HD-audio controller in power-saving mode (default = on)

align_buffer_size
Force rounding of buffer/period sizes to multiples of 128 bytes. This is more efficient in terms of memory access but isn't required by the HDA spec and prevents users from specifying exact period/buffer sizes. (default = on)

snoop
Enable/disable snooping (default = on)

This module supports multiple cards and autoprobe.

See *More Notes on HD-Audio Driver* for more details about HD-audio driver.

Each codec may have a model table for different configurations. If your machine isn't listed there, the default (usually minimal) configuration is set up. You can pass `model=<name>` option to specify a certain model in such a case. There are different models depending on the codec chip. The list of available models is found in *HD-Audio Codec-Specific Models*.

The model name `generic` is treated as a special case. When this model is given, the driver uses the generic codec parser without "codec-patch". It's sometimes good for testing and debugging.

The model option can be used also for aliasing to another PCI or codec SSID. When it's passed in the form of `model=XXXX:YYYY` where XXXX and YYYY are the sub-vendor and sub-device IDs in hex numbers, respectively, the driver will refer to that SSID as a reference to the quirk table.

If the default configuration doesn't work and one of the above matches with your device, report it together with alsa-info.sh output (with `--no-upload` option) to kernel bugzilla or alsa-devel ML (see the section *Links and Addresses*).

`power_save` and `power_save_controller` options are for power-saving mode. See power-save.rst for details.

Note 2: If you get click noises on output, try the module option `position_fix=1` or 2. `position_fix=1` will use the SD_LPIB register value without FIFO size correction as the current DMA pointer. `position_fix=2` will make the driver to use the position buffer instead of reading SD_LPIB register. (Usually SD_LPIB register is more accurate than the position buffer.)

`position_fix=3` is specific to VIA devices. The position of the capture stream is checked from both LPIB and POSBUF values. `position_fix=4` is a combination mode, using LPIB for playback and POSBUF for capture.

NB: If you get many `azx_get_response timeout` messages at loading, it's likely a problem of interrupts (e.g. ACPI irq routing). Try to boot with options like `pci=noacpi`. Also, you can try `single_cmd=1` module option. This will switch the communication method between HDA controller and codecs to the single immediate commands instead of CORB/RIRB. Basically, the single command mode is provided only for BIOS, and you won't get unsolicited events, too. But, at least, this works independently from the irq. Remember this is a last resort, and should be avoided as much as possible...

MORE NOTES ON `azx_get_response timeout` PROBLEMS: On some hardware, you may need to add a proper probe_mask option to avoid the `azx_get_response timeout` problem above, instead. This occurs when the access to non-existing or non-working codec slot (likely a modem one) causes a stall of the communication via HD-audio bus. You can see which codec slots are probed by enabling `CONFIG_SND_DEBUG_VERBOSE`, or simply from the file name of the codec proc files. Then limit the slots to probe by probe_mask option. For example, `probe_mask=1` means to probe only the first slot, and `probe_mask=4` means only the third slot.

The power-management is supported.

4.2.54 Module snd-hdsp

Module for RME Hammerfall DSP audio interface(s)

This module supports multiple cards.

Note: The firmware data can be automatically loaded via hotplug when `CONFIG_FW_LOADER` is set. Otherwise, you need to load the firmware via hdsploader utility included in alsa-tools package. The firmware data is found in alsa-firmware package.

Note: snd-page-alloc module does the job which snd-hammerfall-mem module did formerly. It will allocate the buffers in advance when any HDSP cards are found. To make the buffer allocation sure, load snd-page-alloc module in the early stage of boot sequence. See *Early Buffer Allocation* section.

4.2.55 Module snd-hdspm

Module for RME HDSP MADI board.

precise_ptr
    Enable precise pointer, or disable.

line_outs_monitor
    Send playback streams to analog outs by default.

enable_monitor
    Enable Analog Out on Channel 63/64 by default.

See hdspm.rst for details.

4.2.56 Module snd-ice1712

Module for Envy24 (ICE1712) based PCI sound cards.

- MidiMan M Audio Delta 1010
- MidiMan M Audio Delta 1010LT
- MidiMan M Audio Delta DiO 2496
- MidiMan M Audio Delta 66
- MidiMan M Audio Delta 44
- MidiMan M Audio Delta 410
- MidiMan M Audio Audiophile 2496
- TerraTec EWS 88MT
- TerraTec EWS 88D
- TerraTec EWX 24/96
- TerraTec DMX 6Fire
- TerraTec Phase 88
- Hoontech SoundTrack DSP 24
- Hoontech SoundTrack DSP 24 Value
- Hoontech SoundTrack DSP 24 Media 7.1
- Event Electronics, EZ8
- Digigram VX442
- Lionstracs, Mediastaton
- Terrasoniq TS 88

model
    Use the given board model, one of the following: delta1010, dio2496, delta66, delta44,
    audiophile, delta410, delta1010lt, vx442, ewx2496, ews88mt, ews88mt_new, ews88d,
    dmx6fire, dsp24, dsp24_value, dsp24_71, ez8, phase88, mediastation

omni
    Omni I/O support for MidiMan M-Audio Delta44/66

cs8427_timeout
    reset timeout for the CS8427 chip (S/PDIF transceiver) in msec resolution, default value
    is 500 (0.5 sec)

This module supports multiple cards and autoprobe. Note: The consumer part is not used with
all Envy24 based cards (for example in the MidiMan Delta siree).

Note: The supported board is detected by reading EEPROM or PCI SSID (if EEPROM isn't
available). You can override the model by passing `model` module option in case that the driver
isn't configured properly or you want to try another type for testing.

4.2.57 Module snd-ice1724

Module for Envy24HT (VT/ICE1724), Envy24PT (VT1720) based PCI sound cards.

- MidiMan M Audio Revolution 5.1
- MidiMan M Audio Revolution 7.1
- MidiMan M Audio Audiophile 192
- AMP Ltd AUDIO2000
- TerraTec Aureon 5.1 Sky
- TerraTec Aureon 7.1 Space
- TerraTec Aureon 7.1 Universe
- TerraTec Phase 22
- TerraTec Phase 28
- AudioTrak Prodigy 7.1
- AudioTrak Prodigy 7.1 LT
- AudioTrak Prodigy 7.1 XT
- AudioTrak Prodigy 7.1 HIFI
- AudioTrak Prodigy 7.1 HD2
- AudioTrak Prodigy 192
- Pontis MS300
- Albatron K8X800 Pro II
- Chaintech ZNF3-150
- Chaintech ZNF3-250
- Chaintech 9CJS
- Chaintech AV-710
- Shuttle SN25P
- Onkyo SE-90PCI

- Onkyo SE-200PCI

- ESI Juli@

- ESI Maya44

- Hercules Fortissimo IV

- EGO-SYS WaveTerminal 192M

model

> Use the given board model, one of the following: revo51, revo71, amp2000, prodigy71, prodigy71lt, prodigy71xt, prodigy71hifi, prodigyhd2, prodigy192, juli, aureon51, aureon71, universe, ap192, k8x800, phase22, phase28, ms300, av710, se200pci, se90pci, fortissimo4, sn25p, WT192M, maya44

This module supports multiple cards and autoprobe.

Note: The supported board is detected by reading EEPROM or PCI SSID (if EEPROM isn't available). You can override the model by passing `model` module option in case that the driver isn't configured properly or you want to try another type for testing.

4.2.58 Module snd-indigo

Module for Echoaudio Indigo

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.59 Module snd-indigodj

Module for Echoaudio Indigo DJ

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.60 Module snd-indigoio

Module for Echoaudio Indigo IO

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.61 Module snd-intel8x0

Module for AC'97 motherboards from Intel and compatibles.

- Intel i810/810E, i815, i820, i830, i84x, MX440 ICH5, ICH6, ICH7, 6300ESB, ESB2

- SiS 7012 (SiS 735)

- NVidia NForce, NForce2, NForce3, MCP04, CK804 CK8, CK8S, MCP501

- AMD AMD768, AMD8111

- ALi m5455

ac97_clock
   AC'97 codec clock base (0 = auto-detect)

ac97_quirk
   AC'97 workaround for strange hardware; See *AC97 Quirk Option* section below.

buggy_irq
   Enable workaround for buggy interrupts on some motherboards (default yes on nForce chips, otherwise off)

buggy_semaphore
   Enable workaround for hardware with buggy semaphores (e.g. on some ASUS laptops) (default off)

spdif_aclink
   Use S/PDIF over AC-link instead of direct connection from the controller chip (0 = off, 1 = on, -1 = default)

This module supports one chip and autoprobe.

Note: the latest driver supports auto-detection of chip clock. if you still encounter too fast playback, specify the clock explicitly via the module option `ac97_clock=41194`.

Joystick/MIDI ports are not supported by this driver. If your motherboard has these devices, use the ns558 or snd-mpu401 modules, respectively.

The power-management is supported.

4.2.62 Module snd-intel8x0m

Module for Intel ICH (i8x0) chipset MC97 modems.

   • Intel i810/810E, i815, i820, i830, i84x, MX440 ICH5, ICH6, ICH7

   • SiS 7013 (SiS 735)

   • NVidia NForce, NForce2, NForce2s, NForce3

   • AMD AMD8111

   • ALi m5455

ac97_clock
   AC'97 codec clock base (0 = auto-detect)

This module supports one card and autoprobe.

Note: The default index value of this module is -2, i.e. the first slot is excluded.

The power-management is supported.

4.2.63 Module snd-interwave

Module for Gravis UltraSound PnP, Dynasonic 3-D/Pro, STB Sound Rage 32 and other sound cards based on AMD InterWave (tm) chip.

joystick_dac
> 0 to 31, (0.59V-4.52V or 0.389V-2.98V)

midi
> 1 = MIDI UART enable, 0 = MIDI UART disable (default)

pcm_voices
> reserved PCM voices for the synthesizer (default 2)

effect
> 1 = InterWave effects enable (default 0); requires 8 voices

isapnp
> ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port
> port # for InterWave chip (0x210,0x220,0x230,0x240,0x250,0x260)

irq
> IRQ # for InterWave chip (3,5,9,11,12,15)

dma1
> DMA # for InterWave chip (0,1,3,5,6,7)

dma2
> DMA # for InterWave chip (0,1,3,5,6,7,-1=disable)

This module supports multiple cards, autoprobe and ISA PnP.

4.2.64 Module snd-interwave-stb

Module for UltraSound 32-Pro (sound card from STB used by Compaq) and other sound cards based on AMD InterWave (tm) chip with TEA6330T circuit for extended control of bass, treble and master volume.

joystick_dac
> 0 to 31, (0.59V-4.52V or 0.389V-2.98V)

midi
> 1 = MIDI UART enable, 0 = MIDI UART disable (default)

pcm_voices
> reserved PCM voices for the synthesizer (default 2)

effect
> 1 = InterWave effects enable (default 0); requires 8 voices

isapnp
> ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port

  port # for InterWave chip (0x210,0x220,0x230,0x240,0x250,0x260)

port_tc

  tone control (i2c bus) port # for TEA6330T chip (0x350,0x360,0x370,0x380)

irq

  IRQ # for InterWave chip (3,5,9,11,12,15)

dma1

  DMA # for InterWave chip (0,1,3,5,6,7)

dma2

  DMA # for InterWave chip (0,1,3,5,6,7,-1=disable)

This module supports multiple cards, autoprobe and ISA PnP.

4.2.65 Module snd-jazz16

Module for Media Vision Jazz16 chipset. The chipset consists of 3 chips: MVD1216 + MVA416 + MVA514.

port

  port # for SB DSP chip (0x210,0x220,0x230,0x240,0x250,0x260)

irq

  IRQ # for SB DSP chip (3,5,7,9,10,15)

dma8

  DMA # for SB DSP chip (1,3)

dma16

  DMA # for SB DSP chip (5,7)

mpu_port

  MPU-401 port # (0x300,0x310,0x320,0x330)

mpu_irq

  MPU-401 irq # (2,3,5,7)

This module supports multiple cards.

4.2.66 Module snd-korg1212

Module for Korg 1212 IO PCI card

This module supports multiple cards.

4.2.67 Module snd-layla20

Module for Echoaudio Layla20

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.68 Module snd-layla24

Module for Echoaudio Layla24

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.69 Module snd-lola

Module for Digigram Lola PCI-e boards

This module supports multiple cards.

4.2.70 Module snd-lx6464es

Module for Digigram LX6464ES boards

This module supports multiple cards.

4.2.71 Module snd-maestro3

Module for Allegro/Maestro3 chips

external_amp
    enable external amp (enabled by default)

amp_gpio
    GPIO pin number for external amp (0-15) or -1 for default pin (8 for allegro, 1 for others)

This module supports autoprobe and multiple chips.

Note: the binding of amplifier is dependent on hardware. If there is no sound even though all channels are unmuted, try to specify other gpio connection via amp_gpio option. For example, a Panasonic notebook might need `amp_gpio=0x0d` option.

The power-management is supported.

4.2.72 Module snd-mia

Module for Echoaudio Mia

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.73 Module snd-miro

Module for Miro soundcards: miroSOUND PCM 1 pro, miroSOUND PCM 12, miroSOUND PCM 20 Radio.

port
> Port # (0x530,0x604,0xe80,0xf40)

irq
> IRQ # (5,7,9,10,11)

dma1
> 1st dma # (0,1,3)

dma2
> 2nd dma # (0,1)

mpu_port
> MPU-401 port # (0x300,0x310,0x320,0x330)

mpu_irq
> MPU-401 irq # (5,7,9,10)

fm_port
> FM Port # (0x388)

wss
> enable WSS mode

ide
> enable onboard ide support

4.2.74 Module snd-mixart

Module for Digigram miXart8 sound cards.

This module supports multiple cards. Note: One miXart8 board will be represented as 4 alsa cards. See *Alsa driver for Digigram miXart8 and miXart8AES/EBU soundcards* for details.

When the driver is compiled as a module and the hotplug firmware is supported, the firmware data is loaded via hotplug automatically. Install the necessary firmware files in alsa-firmware package. When no hotplug fw loader is available, you need to load the firmware via mixartloader utility in alsa-tools package.

4.2.75 Module snd-mona

Module for Echoaudio Mona

This module supports multiple cards. The driver requires the firmware loader support on kernel.

4.2.76 Module snd-mpu401

Module for MPU-401 UART devices.

port
> port number or -1 (disable)

irq
> IRQ number or -1 (disable)

pnp
> PnP detection - 0 = disable, 1 = enable (default)

This module supports multiple devices and PnP.

4.2.77 Module snd-msnd-classic

Module for Turtle Beach MultiSound Classic, Tahiti or Monterey soundcards.

io
> Port # for msnd-classic card

irq
> IRQ # for msnd-classic card

mem
> Memory address (0xb0000, 0xc8000, 0xd0000, 0xd8000, 0xe0000 or 0xe8000)

write_ndelay
> enable write ndelay (default = 1)

calibrate_signal
> calibrate signal (default = 0)

isapnp
> ISA PnP detection - 0 = disable, 1 = enable (default)

digital
> Digital daughterboard present (default = 0)

cfg
> Config port (0x250, 0x260 or 0x270) default = PnP

reset
> Reset all devices

mpu_io
> MPU401 I/O port

mpu_irq
> MPU401 irq#

ide_io0
>   IDE port #0

ide_io1
>   IDE port #1

ide_irq
>   IDE irq#

joystick_io
>   Joystick I/O port

The driver requires firmware files `turtlebeach/msndinit.bin` and `turtlebeach/msndperm.bin` in the proper firmware directory.

See Documentation/sound/cards/multisound.sh for important information about this driver. Note that it has been discontinued, but the Voyetra Turtle Beach knowledge base entry for it is still available at https://www.turtlebeach.com

4.2.78 Module snd-msnd-pinnacle

Module for Turtle Beach MultiSound Pinnacle/Fiji soundcards.

io
>   Port # for pinnacle/fiji card

irq
>   IRQ # for pinnalce/fiji card

mem
>   Memory address (0xb0000, 0xc8000, 0xd0000, 0xd8000, 0xe0000 or 0xe8000)

write_ndelay
>   enable write ndelay (default = 1)

calibrate_signal
>   calibrate signal (default = 0)

isapnp
>   ISA PnP detection - 0 = disable, 1 = enable (default)

The driver requires firmware files `turtlebeach/pndspini.bin` and `turtlebeach/pndsperm.bin` in the proper firmware directory.

4.2.79 Module snd-mtpav

Module for MOTU MidiTimePiece AV multiport MIDI (on the parallel port).

port
>   I/O port # for MTPAV (0x378,0x278, default=0x378)

irq
>   IRQ # for MTPAV (7,5, default=7)

hwports
>   number of supported hardware ports, default=8.

Module supports only 1 card. This module has no enable option.

4.2.80 Module snd-mts64

Module for Ego Systems (ESI) Miditerminal 4140

This module supports multiple devices. Requires parport (`CONFIG_PARPORT`).

4.2.81 Module snd-nm256

Module for NeoMagic NM256AV/ZX chips

playback_bufsize
    max playback frame size in kB (4-128kB)

capture_bufsize
    max capture frame size in kB (4-128kB)

force_ac97
    0 or 1 (disabled by default)

buffer_top
    specify buffer top address

use_cache
    0 or 1 (disabled by default)

vaio_hack
    alias buffer_top=0x25a800

reset_workaround
    enable AC97 RESET workaround for some laptops

reset_workaround2
    enable extended AC97 RESET workaround for some other laptops

This module supports one chip and autoprobe.

The power-management is supported.

Note: on some notebooks the buffer address cannot be detected automatically, or causes hang-up during initialization. In such a case, specify the buffer top address explicitly via the buffer_top option. For example, Sony F250: buffer_top=0x25a800 Sony F270: buffer_top=0x272800 The driver supports only ac97 codec. It's possible to force to initialize/use ac97 although it's not detected. In such a case, use `force_ac97=1` option - but *NO* guarantee whether it works!

Note: The NM256 chip can be linked internally with non-AC97 codecs. This driver supports only the AC97 codec, and won't work with machines with other (most likely CS423x or OPL3SAx) chips, even though the device is detected in lspci. In such a case, try other drivers, e.g. snd-cs4232 or snd-opl3sa2. Some has ISA-PnP but some doesn't have ISA PnP. You'll need to specify `isapnp=0` and proper hardware parameters in the case without ISA PnP.

Note: some laptops need a workaround for AC97 RESET. For the known hardware like Dell Latitude LS and Sony PCG-F305, this workaround is enabled automatically. For other laptops with a hard freeze, you can try `reset_workaround=1` option.

Note: Dell Latitude CSx laptops have another problem regarding AC97 RESET. On these laptops, reset_workaround2 option is turned on as default. This option is worth to try if the previous reset_workaround option doesn't help.

Note: This driver is really crappy. It's a porting from the OSS driver, which is a result of black-magic reverse engineering. The detection of codec will fail if the driver is loaded *after* X-server as described above. You might be able to force to load the module, but it may result in hang-up. Hence, make sure that you load this module *before* X if you encounter this kind of problem.

4.2.82 Module snd-opl3sa2

Module for Yamaha OPL3-SA2/SA3 sound cards.

isapnp
> ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port
> control port # for OPL3-SA chip (0x370)

sb_port
> SB port # for OPL3-SA chip (0x220,0x240)

wss_port
> WSS port # for OPL3-SA chip (0x530,0xe80,0xf40,0x604)

midi_port
> port # for MPU-401 UART (0x300,0x330), -1 = disable

fm_port
> FM port # for OPL3-SA chip (0x388), -1 = disable

irq
> IRQ # for OPL3-SA chip (5,7,9,10)

dma1
> first DMA # for Yamaha OPL3-SA chip (0,1,3)

dma2
> second DMA # for Yamaha OPL3-SA chip (0,1,3), -1 = disable

This module supports multiple cards and ISA PnP. It does not support autoprobe (if ISA PnP is not used) thus all ports must be specified!!!

The power-management is supported.

4.2.83 Module snd-opti92x-ad1848

Module for sound cards based on OPTi 82c92x and Analog Devices AD1848 chips. Module works with OAK Mozart cards as well.

isapnp
> ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port
> port # for WSS chip (0x530,0xe80,0xf40,0x604)

mpu_port
> port # for MPU-401 UART (0x300,0x310,0x320,0x330)

fm_port
    port # for OPL3 device (0x388)

irq
    IRQ # for WSS chip (5,7,9,10,11)

mpu_irq
    IRQ # for MPU-401 UART (5,7,9,10)

dma1
    first DMA # for WSS chip (0,1,3)

This module supports only one card, autoprobe and PnP.

4.2.84 Module snd-opti92x-cs4231

Module for sound cards based on OPTi 82c92x and Crystal CS4231 chips.

isapnp
    ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port
    port # for WSS chip (0x530,0xe80,0xf40,0x604)

mpu_port
    port # for MPU-401 UART (0x300,0x310,0x320,0x330)

fm_port
    port # for OPL3 device (0x388)

irq
    IRQ # for WSS chip (5,7,9,10,11)

mpu_irq
    IRQ # for MPU-401 UART (5,7,9,10)

dma1
    first DMA # for WSS chip (0,1,3)

dma2
    second DMA # for WSS chip (0,1,3)

This module supports only one card, autoprobe and PnP.

4.2.85 Module snd-opti93x

Module for sound cards based on OPTi 82c93x chips.

isapnp
    ISA PnP detection - 0 = disable, 1 = enable (default)

with `isapnp=0`, the following options are available:

port
    port # for WSS chip (0x530,0xe80,0xf40,0x604)

mpu_port
　　port # for MPU-401 UART (0x300,0x310,0x320,0x330)

fm_port
　　port # for OPL3 device (0x388)

irq
　　IRQ # for WSS chip (5,7,9,10,11)

mpu_irq
　　IRQ # for MPU-401 UART (5,7,9,10)

dma1
　　first DMA # for WSS chip (0,1,3)

dma2
　　second DMA # for WSS chip (0,1,3)

This module supports only one card, autoprobe and PnP.

4.2.86 Module snd-oxygen

Module for sound cards based on the C-Media CMI8786/8787/8788 chip:

- Asound A-8788
- Asus Xonar DG/DGX
- AuzenTech X-Meridian
- AuzenTech X-Meridian 2G
- Bgears b-Enspirer
- Club3D Theatron DTS
- HT-Omega Claro (plus)
- HT-Omega Claro halo (XT)
- Kuroutoshikou CMI8787-HG2PCI
- Razer Barracuda AC-1
- Sondigo Inferno
- TempoTec HiFier Fantasia
- TempoTec HiFier Serenade

This module supports autoprobe and multiple cards.

4.2.87 Module snd-pcsp

Module for internal PC-Speaker.

nopcm
> Disable PC-Speaker PCM sound. Only beeps remain.

nforce_wa
> enable NForce chipset workaround. Expect bad sound.

This module supports system beeps, some kind of PCM playback and even a few mixer controls.

4.2.88 Module snd-pcxhr

Module for Digigram PCXHR boards

This module supports multiple cards.

4.2.89 Module snd-portman2x4

Module for Midiman Portman 2x4 parallel port MIDI interface

This module supports multiple cards.

4.2.90 Module snd-powermac (on ppc only)

Module for PowerMac, iMac and iBook on-board soundchips

enable_beep
> enable beep using PCM (enabled as default)

Module supports autoprobe a chip.

Note: the driver may have problems regarding endianness.

The power-management is supported.

4.2.91 Module snd-pxa2xx-ac97 (on arm only)

Module for AC97 driver for the Intel PXA2xx chip

For ARM architecture only.

The power-management is supported.

4.2.92 Module snd-riptide

Module for Conexant Riptide chip

joystick_port
Joystick port # (default: 0x200)

mpu_port
MPU401 port # (default: 0x330)

opl3_port
OPL3 port # (default: 0x388)

This module supports multiple cards. The driver requires the firmware loader support on kernel. You need to install the firmware file `riptide.hex` to the standard firmware path (e.g. /lib/firmware).

4.2.93 Module snd-rme32

Module for RME Digi32, Digi32 Pro and Digi32/8 (Sek'd Prodif32, Prodif96 and Prodif Gold) sound cards.

This module supports multiple cards.

4.2.94 Module snd-rme96

Module for RME Digi96, Digi96/8 and Digi96/8 PRO/PAD/PST sound cards.

This module supports multiple cards.

4.2.95 Module snd-rme9652

Module for RME Digi9652 (Hammerfall, Hammerfall-Light) sound cards.

precise_ptr
Enable precise pointer (doesn't work reliably). (default = 0)

This module supports multiple cards.

Note: snd-page-alloc module does the job which snd-hammerfall-mem module did formerly. It will allocate the buffers in advance when any RME9652 cards are found. To make the buffer allocation sure, load snd-page-alloc module in the early stage of boot sequence. See *Early Buffer Allocation* section.

4.2.96 Module snd-sa11xx-uda1341 (on arm only)

Module for Philips UDA1341TS on Compaq iPAQ H3600 sound card.

Module supports only one card. Module has no enable and index options.

The power-management is supported.

4.2.97 Module snd-sb8

Module for 8-bit SoundBlaster cards: SoundBlaster 1.0, SoundBlaster 2.0, SoundBlaster Pro

port
    port # for SB DSP chip (0x220,0x240,0x260)

irq
    IRQ # for SB DSP chip (5,7,9,10)

dma8
    DMA # for SB DSP chip (1,3)

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.98 Module snd-sb16 and snd-sbawe

Module for 16-bit SoundBlaster cards: SoundBlaster 16 (PnP), SoundBlaster AWE 32 (PnP), SoundBlaster AWE 64 PnP

mic_agc
    Mic Auto-Gain-Control - 0 = disable, 1 = enable (default)

csp
    ASP/CSP chip support - 0 = disable (default), 1 = enable

isapnp
    ISA PnP detection - 0 = disable, 1 = enable (default)

with isapnp=0, the following options are available:

port
    port # for SB DSP 4.x chip (0x220,0x240,0x260)

mpu_port
    port # for MPU-401 UART (0x300,0x330), -1 = disable

awe_port
    base port # for EMU8000 synthesizer (0x620,0x640,0x660) (snd-sbawe module only)

irq
    IRQ # for SB DSP 4.x chip (5,7,9,10)

dma8
    8-bit DMA # for SB DSP 4.x chip (0,1,3)

dma16
    16-bit DMA # for SB DSP 4.x chip (5,6,7)

This module supports multiple cards, autoprobe and ISA PnP.

Note: To use Vibra16X cards in 16-bit half duplex mode, you must disable 16bit DMA with dma16 = -1 module parameter. Also, all Sound Blaster 16 type cards can operate in 16-bit half duplex mode through 8-bit DMA channel by disabling their 16-bit DMA channel.

The power-management is supported.

4.2.99 Module snd-sc6000

Module for Gallant SC-6000 soundcard and later models: SC-6600 and SC-7000.

port
    Port # (0x220 or 0x240)

mss_port
    MSS Port # (0x530 or 0xe80)

irq
    IRQ # (5,7,9,10,11)

mpu_irq
    MPU-401 IRQ # (5,7,9,10) ,0 - no MPU-401 irq

dma
    DMA # (1,3,0)

joystick
    Enable gameport - 0 = disable (default), 1 = enable

This module supports multiple cards.

This card is also known as Audio Excel DSP 16 or Zoltrix AV302.

4.2.100 Module snd-sscape

Module for ENSONIQ SoundScape cards.

port
    Port # (PnP setup)

wss_port
    WSS Port # (PnP setup)

irq
    IRQ # (PnP setup)

mpu_irq
    MPU-401 IRQ # (PnP setup)

dma
    DMA # (PnP setup)

dma2
    2nd DMA # (PnP setup, -1 to disable)

joystick
    Enable gameport - 0 = disable (default), 1 = enable

This module supports multiple cards.

The driver requires the firmware loader support on kernel.

4.2.101 Module snd-sun-amd7930 (on sparc only)

Module for AMD7930 sound chips found on Sparcs.

This module supports multiple cards.

4.2.102 Module snd-sun-cs4231 (on sparc only)

Module for CS4231 sound chips found on Sparcs.

This module supports multiple cards.

4.2.103 Module snd-sun-dbri (on sparc only)

Module for DBRI sound chips found on Sparcs.

This module supports multiple cards.

4.2.104 Module snd-wavefront

Module for Turtle Beach Maui, Tropez and Tropez+ sound cards.

use_cs4232_midi
Use CS4232 MPU-401 interface (inaccessibly located inside your computer)

isapnp
ISA PnP detection - 0 = disable, 1 = enable (default)

with isapnp=0, the following options are available:

cs4232_pcm_port
Port # for CS4232 PCM interface.

cs4232_pcm_irq
IRQ # for CS4232 PCM interface (5,7,9,11,12,15).

cs4232_mpu_port
Port # for CS4232 MPU-401 interface.

cs4232_mpu_irq
IRQ # for CS4232 MPU-401 interface (9,11,12,15).

ics2115_port
Port # for ICS2115

ics2115_irq
IRQ # for ICS2115

fm_port
FM OPL-3 Port #

dma1
    DMA1 # for CS4232 PCM interface.

dma2
    DMA2 # for CS4232 PCM interface.

The below are options for wavefront_synth features:

wf_raw
    Assume that we need to boot the OS (default:no); If yes, then during driver loading, the state of the board is ignored, and we reset the board and load the firmware anyway.

fx_raw
    Assume that the FX process needs help (default:yes); If false, we'll leave the FX processor in whatever state it is when the driver is loaded. The default is to download the microprogram and associated coefficients to set it up for "default" operation, whatever that means.

debug_default
    Debug parameters for card initialization

wait_usecs
    How long to wait without sleeping, usecs (default:150); This magic number seems to give pretty optimal throughput based on my limited experimentation. If you want to play around with it and find a better value, be my guest. Remember, the idea is to get a number that causes us to just busy wait for as many WaveFront commands as possible, without coming up with a number so large that we hog the whole CPU. Specifically, with this number, out of about 134,000 status waits, only about 250 result in a sleep.

sleep_interval
    How long to sleep when waiting for reply (default: 100)

sleep_tries
    How many times to try sleeping during a wait (default: 50)

ospath
    Pathname to processed ICS2115 OS firmware (default:wavefront.os); The path name of the ISC2115 OS firmware. In the recent version, it's handled via firmware loader framework, so it must be installed in the proper path, typically, /lib/firmware.

reset_time
    How long to wait for a reset to take effect (default:2)

ramcheck_time
    How many seconds to wait for the RAM test (default:20)

osrun_time
    How many seconds to wait for the ICS2115 OS (default:10)

This module supports multiple cards and ISA PnP.

Note: the firmware file `wavefront.os` was located in the earlier version in /etc. Now it's loaded via firmware loader, and must be in the proper firmware path, such as /lib/firmware. Copy (or symlink) the file appropriately if you get an error regarding firmware downloading after upgrading the kernel.

4.2.105 Module snd-sonicvibes

Module for S3 SonicVibes PCI sound cards. * PINE Schubert 32 PCI

reverb
    Reverb Enable - 1 = enable, 0 = disable (default); SoundCard must have onboard SRAM
    for this.

mge
    Mic Gain Enable - 1 = enable, 0 = disable (default)

This module supports multiple cards and autoprobe.

4.2.106 Module snd-serial-u16550

Module for UART16550A serial MIDI ports.

port
    port # for UART16550A chip

irq
    IRQ # for UART16550A chip, -1 = poll mode

speed
    speed in bauds (9600,19200,38400,57600,115200) 38400 = default

base
    base for divisor in bauds (57600,115200,230400,460800) 115200 = default

outs
    number of MIDI ports in a serial port (1-4) 1 = default

adaptor

    Type of adaptor.
        0 = Soundcanvas, 1 = MS-124T, 2 = MS-124W S/A, 3 = MS-124W M/B, 4 = Generic

This module supports multiple cards. This module does not support autoprobe thus the main
port must be specified!!! Other options are optional.

4.2.107 Module snd-trident

Module for Trident 4DWave DX/NX sound cards. * Best Union Miss Melody 4DWave PCI * HIS
4DWave PCI * Warpspeed ONSpeed 4DWave PCI * AzTech PCI 64-Q3D * Addonics SV 750 *
CHIC True Sound 4Dwave * Shark Predator4D-PCI * Jaton SonicWave 4D * SiS SI7018 PCI
Audio * Hoontech SoundTrack Digital 4DWave NX

pcm_channels
    max channels (voices) reserved for PCM

wavetable_size
    max wavetable size in kB (4-?kb)

This module supports multiple cards and autoprobe.

The power-management is supported.

4.2.108 Module snd-ua101

Module for the Edirol UA-101/UA-1000 audio/MIDI interfaces.

This module supports multiple devices, autoprobe and hotplugging.

4.2.109 Module snd-usb-audio

Module for USB audio and USB MIDI devices.

vid
: Vendor ID for the device (optional)

pid
: Product ID for the device (optional)

nrpacks
: Max. number of packets per URB (default: 8)

device_setup
: Device specific magic number (optional); Influence depends on the device Default: 0x0000

ignore_ctl_error
: Ignore any USB-controller regarding mixer interface (default: no)

autoclock
: Enable auto-clock selection for UAC2 devices (default: yes)

quirk_alias
: Quirk alias list, pass strings like `0123abcd:5678beef`, which applies the existing quirk for the device 5678:beef to a new device 0123:abcd.

implicit_fb
: Apply the generic implicit feedback sync mode. When this is set and the playback stream sync mode is ASYNC, the driver tries to tie an adjacent ASYNC capture stream as the implicit feedback source. This is equivalent with quirk_flags bit 17.

use_vmalloc
: Use vmalloc() for allocations of the PCM buffers (default: yes). For architectures with non-coherent memory like ARM or MIPS, the mmap access may give inconsistent results with vmalloc'ed buffers. If mmap is used on such architectures, turn off this option, so that the DMA-coherent buffers are allocated and used instead.

delayed_register
: The option is needed for devices that have multiple streams defined in multiple USB interfaces. The driver may invoke registrations multiple times (once per interface) and this may lead to the insufficient device enumeration. This option receives an array of strings, and you can pass ID:INTERFACE like `0123abcd:4` for performing the delayed registration to the given device. In this example, when a USB device 0123:abcd is probed, the driver waits the registration until the USB interface 4 gets probed. The driver prints a message like "Found post-registration device assignment: 1234abcd:04" for such a device, so that user can notice the need.

quirk_flags
: Contains the bit flags for various device specific workarounds. Applied to the corresponding card index.

- bit 0: Skip reading sample rate for devices
- bit 1: Create Media Controller API entries
- bit 2: Allow alignment on audio sub-slot at transfer
- bit 3: Add length specifier to transfers
- bit 4: Start playback stream at first in implement feedback mode
- bit 5: Skip clock selector setup
- bit 6: Ignore errors from clock source search
- bit 7: Indicates ITF-USB DSD based DACs
- bit 8: Add a delay of 20ms at each control message handling
- bit 9: Add a delay of 1-2ms at each control message handling
- bit 10: Add a delay of 5-6ms at each control message handling
- bit 11: Add a delay of 50ms at each interface setup
- bit 12: Perform sample rate validations at probe
- bit 13: Disable runtime PM autosuspend
- bit 14: Ignore errors for mixer access
- bit 15: Support generic DSD raw U32_BE format
- bit 16: Set up the interface at first like UAC1
- bit 17: Apply the generic implicit feedback sync mode
- bit 18: Don't apply implicit feedback sync mode

This module supports multiple devices, autoprobe and hotplugging.

NB: `nrpacks` parameter can be modified dynamically via sysfs. Don't put the value over 20. Changing via sysfs has no sanity check.

NB: `ignore_ctl_error=1` may help when you get an error at accessing the mixer element such as URB error -22. This happens on some buggy USB device or the controller. This workaround corresponds to the `quirk_flags` bit 14, too.

NB: `quirk_alias` option is provided only for testing / development. If you want to have a proper support, contact to upstream for adding the matching quirk in the driver code statically. Ditto for `quirk_flags`. If a device is known to require specific workarounds, please report to the upstream.

4.2.110 Module snd-usb-caiaq

Module for caiaq UB audio interfaces,

- Native Instruments RigKontrol2
- Native Instruments Kore Controller
- Native Instruments Audio Kontrol 1
- Native Instruments Audio 8 DJ

This module supports multiple devices, autoprobe and hotplugging.

4.2.111 Module snd-usb-usx2y

Module for Tascam USB US-122, US-224 and US-428 devices.

This module supports multiple devices, autoprobe and hotplugging.

Note: you need to load the firmware via `usx2yloader` utility included in alsa-tools and alsa-firmware packages.

4.2.112 Module snd-via82xx

Module for AC'97 motherboards based on VIA 82C686A/686B, 8233, 8233A, 8233C, 8235, 8237 (south) bridge.

mpu_port
0x300,0x310,0x320,0x330, otherwise obtain BIOS setup [VIA686A/686B only]

joystick
Enable joystick (default off) [VIA686A/686B only]

ac97_clock
AC'97 codec clock base (default 48000Hz)

dxs_support
support DXS channels, 0 = auto (default), 1 = enable, 2 = disable, 3 = 48k only, 4 = no VRA, 5 = enable any sample rate and different sample rates on different channels [VIA8233/C, 8235, 8237 only]

ac97_quirk
AC'97 workaround for strange hardware; See *AC97 Quirk Option* section below.

This module supports one chip and autoprobe.

Note: on some SMP motherboards like MSI 694D the interrupts might not be generated properly. In such a case, please try to set the SMP (or MPS) version on BIOS to 1.1 instead of default value 1.4. Then the interrupt number will be assigned under 15. You might also upgrade your BIOS.

Note: VIA8233/5/7 (not VIA8233A) can support DXS (direct sound) channels as the first PCM. On these channels, up to 4 streams can be played at the same time, and the controller can perform sample rate conversion with separate rates for each channel. As default (`dxs_support` = 0), 48k fixed rate is chosen except for the known devices since the output is often noisy except for 48k on some mother boards due to the bug of BIOS. Please try once `dxs_support=5` and if it works on other sample rates (e.g. 44.1kHz of mp3 playback), please let us know the PCI subsystem vendor/device id's (output of `lspci -nv`). If `dxs_support=5` does not work, try `dxs_support=4`; if it doesn't work too, try dxs_support=1. (dxs_support=1 is usually for old motherboards. The correct implemented board should work with 4 or 5.) If it still doesn't work and the default setting is ok, `dxs_support=3` is the right choice. If the default setting doesn't work at all, try `dxs_support=2` to disable the DXS channels. In any cases, please let us know the result and the subsystem vendor/device ids. See *Links and Addresses* below.

Note: for the MPU401 on VIA823x, use snd-mpu401 driver additionally. The mpu_port option is for VIA686 chips only.

The power-management is supported.

4.2.113 Module snd-via82xx-modem

Module for VIA82xx AC97 modem

ac97_clock
    AC'97 codec clock base (default 48000Hz)

This module supports one card and autoprobe.

Note: The default index value of this module is -2, i.e. the first slot is excluded.

The power-management is supported.

4.2.114 Module snd-virmidi

Module for virtual rawmidi devices. This module creates virtual rawmidi devices which communicate to the corresponding ALSA sequencer ports.

midi_devs
    MIDI devices # (1-4, default=4)

This module supports multiple cards.

4.2.115 Module snd-virtuoso

Module for sound cards based on the Asus AV66/AV100/AV200 chips, i.e., Xonar D1, DX, D2, D2X, DS, DSX, Essence ST (Deluxe), Essence STX (II), HDAV1.3 (Deluxe), and HDAV1.3 Slim.

This module supports autoprobe and multiple cards.

4.2.116 Module snd-vx222

Module for Digigram VX-Pocket VX222, V222 v2 and Mic cards.

mic
    Enable Microphone on V222 Mic (NYI)

ibl
    Capture IBL size. (default = 0, minimum size)

This module supports multiple cards.

When the driver is compiled as a module and the hotplug firmware is supported, the firmware data is loaded via hotplug automatically. Install the necessary firmware files in alsa-firmware package. When no hotplug fw loader is available, you need to load the firmware via vxloader utility in alsa-tools package. To invoke vxloader automatically, add the following to /etc/modprobe.d/alsa.conf

```
install snd-vx222 /sbin/modprobe --first-time -i snd-vx222\
  && /usr/bin/vxloader
```

(for 2.2/2.4 kernels, add `post-install /usr/bin/vxloader` to /etc/modules.conf, instead.) IBL size defines the interrupts period for PCM. The smaller size gives smaller latency but leads to more CPU consumption, too. The size is usually aligned to 126. As default (=0), the smallest size is chosen. The possible IBL values can be found in /proc/asound/cardX/vx-status proc file.

The power-management is supported.

4.2.117 Module snd-vxpocket

Module for Digigram VX-Pocket VX2 and 440 PCMCIA cards.

ibl
Capture IBL size. (default = 0, minimum size)

This module supports multiple cards. The module is compiled only when PCMCIA is supported on kernel.

With the older 2.6.x kernel, to activate the driver via the card manager, you'll need to set up /etc/pcmcia/vxpocket.conf. See the sound/pcmcia/vx/vxpocket.c. 2.6.13 or later kernel requires no longer require a config file.

When the driver is compiled as a module and the hotplug firmware is supported, the firmware data is loaded via hotplug automatically. Install the necessary firmware files in alsa-firmware package. When no hotplug fw loader is available, you need to load the firmware via vxloader utility in alsa-tools package.

About capture IBL, see the description of snd-vx222 module.

Note: snd-vxp440 driver is merged to snd-vxpocket driver since ALSA 1.0.10.

The power-management is supported.

4.2.118 Module snd-ymfpci

Module for Yamaha PCI chips (YMF72x, YMF74x & YMF75x).

mpu_port
0x300,0x330,0x332,0x334, 0 (disable) by default, 1 (auto-detect for YMF744/754 only)

fm_port
0x388,0x398,0x3a0,0x3a8, 0 (disable) by default 1 (auto-detect for YMF744/754 only)

joystick_port
0x201,0x202,0x204,0x205, 0 (disable) by default, 1 (auto-detect)

rear_switch
enable shared rear/line-in switch (bool)

This module supports autoprobe and multiple chips.

The power-management is supported.

4.2.119 Module snd-pdaudiocf

Module for Sound Core PDAudioCF sound card.

The power-management is supported.

4.3 AC97 Quirk Option

The ac97_quirk option is used to enable/override the workaround for specific devices on drivers for on-board AC'97 controllers like snd-intel8x0. Some hardware have swapped output pins between Master and Headphone, or Surround (thanks to confusion of AC'97 specifications from version to version :-)

The driver provides the auto-detection of known problematic devices, but some might be unknown or wrongly detected. In such a case, pass the proper value with this option.

The following strings are accepted:

default
> Don't override the default setting

none
> Disable the quirk

hp_only
> Bind Master and Headphone controls as a single control

swap_hp
> Swap headphone and master controls

swap_surround
> Swap master and surround controls

ad_sharing
> For AD1985, turn on OMS bit and use headphone

alc_jack
> For ALC65x, turn on the jack sense mode

inv_eapd
> Inverted EAPD implementation

mute_led
> Bind EAPD bit for turning on/off mute LED

For backward compatibility, the corresponding integer value -1, 0, ... are accepted, too.

For example, if `Master` volume control has no effect on your device but only `Headphone` does, pass ac97_quirk=hp_only module option.

4.4 Configuring Non-ISAPNP Cards

When the kernel is configured with ISA-PnP support, the modules supporting the isapnp cards will have module options `isapnp`. If this option is set, *only* the ISA-PnP devices will be probed. For probing the non ISA-PnP cards, you have to pass `isapnp=0` option together with the proper i/o and irq configuration.

When the kernel is configured without ISA-PnP support, isapnp option will be not built in.

4.5 Module Autoloading Support

The ALSA drivers can be loaded automatically on demand by defining module aliases. The string `snd-card-%1` is requested for ALSA native devices where `%i` is sound card number from zero to seven.

To auto-load an ALSA driver for OSS services, define the string `sound-slot-%i` where `%i` means the slot number for OSS, which corresponds to the card index of ALSA. Usually, define this as the same card module.

An example configuration for a single emu10k1 card is like below:

```
----- /etc/modprobe.d/alsa.conf
alias snd-card-0 snd-emu10k1
alias sound-slot-0 snd-emu10k1
----- /etc/modprobe.d/alsa.conf
```

The available number of auto-loaded sound cards depends on the module option `cards_limit` of snd module. As default it's set to 1. To enable the auto-loading of multiple cards, specify the number of sound cards in that option.

When multiple cards are available, it'd better to specify the index number for each card via module option, too, so that the order of cards is kept consistent.

An example configuration for two sound cards is like below:

```
----- /etc/modprobe.d/alsa.conf
# ALSA portion
options snd cards_limit=2
alias snd-card-0 snd-interwave
alias snd-card-1 snd-ens1371
options snd-interwave index=0
options snd-ens1371 index=1
# OSS/Free portion
alias sound-slot-0 snd-interwave
alias sound-slot-1 snd-ens1371
----- /etc/modprobe.d/alsa.conf
```

In this example, the interwave card is always loaded as the first card (index 0) and ens1371 as the second (index 1).

Alternative (and new) way to fixate the slot assignment is to use `slots` option of snd module. In the case above, specify like the following:

```
options snd slots=snd-interwave,snd-ens1371
```

Then, the first slot (#0) is reserved for snd-interwave driver, and the second (#1) for snd-ens1371. You can omit index option in each driver if slots option is used (although you can still have them at the same time as long as they don't conflict).

The slots option is especially useful for avoiding the possible hot-plugging and the resultant slot conflict. For example, in the case above again, the first two slots are already reserved. If any other driver (e.g. snd-usb-audio) is loaded before snd-interwave or snd-ens1371, it will be assigned to the third or later slot.

When a module name is given with '!', the slot will be given for any modules but that name. For example, `slots=!snd-pcsp` will reserve the first slot for any modules but snd-pcsp.

4.6 ALSA PCM devices to OSS devices mapping

```
/dev/snd/pcmC0D0[c|p]  -> /dev/audio0 (/dev/audio) -> minor 4
/dev/snd/pcmC0D0[c|p]  -> /dev/dsp0 (/dev/dsp)     -> minor 3
/dev/snd/pcmC0D1[c|p]  -> /dev/adsp0 (/dev/adsp)   -> minor 12
/dev/snd/pcmC1D0[c|p]  -> /dev/audio1              -> minor 4+16 = 20
/dev/snd/pcmC1D0[c|p]  -> /dev/dsp1                -> minor 3+16 = 19
/dev/snd/pcmC1D1[c|p]  -> /dev/adsp1               -> minor 12+16 = 28
/dev/snd/pcmC2D0[c|p]  -> /dev/audio2              -> minor 4+32 = 36
/dev/snd/pcmC2D0[c|p]  -> /dev/dsp2                -> minor 3+32 = 39
/dev/snd/pcmC2D1[c|p]  -> /dev/adsp2               -> minor 12+32 = 44
```

The first number from `/dev/snd/pcmC{X}D{Y}[c|p]` expression means sound card number and second means device number. The ALSA devices have either `c` or `p` suffix indicating the direction, capture and playback, respectively.

Please note that the device mapping above may be varied via the module options of snd-pcm-oss module.

4.7 Proc interfaces (/proc/asound)

4.7.1 /proc/asound/card#/pcm#[cp]/oss

erase
   erase all additional information about OSS applications

<app_name> <fragments> <fragment_size> [<options>]

   <app_name>
      name of application with (higher priority) or without path

   <fragments>
      number of fragments or zero if auto

   <fragment_size>
      size of fragment in bytes or zero if auto

<options>
> optional parameters

> **disable**
>> the application tries to open a pcm device for this channel but does not want to use it. (Cause a bug or mmap needs) It's good for Quake etc...

> **direct**
>> don't use plugins

> **block**
>> force block mode (rvplayer)

> **non-block**
>> force non-block mode

> **whole-frag**
>> write only whole fragments (optimization affecting playback only)

> **no-silence**
>> do not fill silence ahead to avoid clicks

> **buggy-ptr**
>> Returns the whitespace blocks in GETOPTR ioctl instead of filled blocks

Example:

```
echo "x11amp 128 16384" > /proc/asound/card0/pcm0p/oss
echo "squake 0 0 disable" > /proc/asound/card0/pcm0c/oss
echo "rvplayer 0 0 block" > /proc/asound/card0/pcm0p/oss
```

4.8 Early Buffer Allocation

Some drivers (e.g. hdsp) require the large contiguous buffers, and sometimes it's too late to find such spaces when the driver module is actually loaded due to memory fragmentation. You can pre-allocate the PCM buffers by loading snd-page-alloc module and write commands to its proc file in prior, for example, in the early boot stage like `/etc/init.d/*.local` scripts.

Reading the proc file /proc/drivers/snd-page-alloc shows the current usage of page allocation. In writing, you can send the following commands to the snd-page-alloc driver:

- add VENDOR DEVICE MASK SIZE BUFFERS

VENDOR and DEVICE are PCI vendor and device IDs. They take integer numbers (0x prefix is needed for the hex). MASK is the PCI DMA mask. Pass 0 if not restricted. SIZE is the size of each buffer to allocate. You can pass k and m suffix for KB and MB. The max number is 16MB. BUFFERS is the number of buffers to allocate. It must be greater than 0. The max number is 4.

- erase

This will erase the all pre-allocated buffers which are not in use.

4.9 Links and Addresses

ALSA project homepage
> http://www.alsa-project.org

Kernel Bugzilla
> http://bugzilla.kernel.org/

ALSA Developers ML
> mailto:alsa-devel@alsa-project.org

alsa-info.sh script
> https://www.alsa-project.org/alsa-info.sh

HD-AUDIO

5.1 More Notes on HD-Audio Driver

Takashi Iwai <tiwai@suse.de>

5.1.1 General

HD-audio is the new standard on-board audio component on modern PCs after AC97. Although Linux has been supporting HD-audio since long time ago, there are often problems with new machines. A part of the problem is broken BIOS, and the rest is the driver implementation. This document explains the brief trouble-shooting and debugging methods for the HD-audio hardware.

The HD-audio component consists of two parts: the controller chip and the codec chips on the HD-audio bus. Linux provides a single driver for all controllers, snd-hda-intel. Although the driver name contains a word of a well-known hardware vendor, it's not specific to it but for all controller chips by other companies. Since the HD-audio controllers are supposed to be compatible, the single snd-hda-driver should work in most cases. But, not surprisingly, there are known bugs and issues specific to each controller type. The snd-hda-intel driver has a bunch of workarounds for these as described below.

A controller may have multiple codecs. Usually you have one audio codec and optionally one modem codec. In theory, there might be multiple audio codecs, e.g. for analog and digital outputs, and the driver might not work properly because of conflict of mixer elements. This should be fixed in future if such hardware really exists.

The snd-hda-intel driver has several different codec parsers depending on the codec. It has a generic parser as a fallback, but this functionality is fairly limited until now. Instead of the generic parser, usually the codec-specific parser (coded in patch_*.c) is used for the codec-specific implementations. The details about the codec-specific problems are explained in the later sections.

If you are interested in the deep debugging of HD-audio, read the HD-audio specification at first. The specification is found on Intel's web page, for example:

- https://www.intel.com/standards/hdaudio/

5.1.2 HD-Audio Controller

DMA-Position Problem

The most common problem of the controller is the inaccurate DMA pointer reporting. The DMA pointer for playback and capture can be read in two ways, either via a LPIB register or via a position-buffer map. As default the driver tries to read from the io-mapped position-buffer, and falls back to LPIB if the position-buffer appears dead. However, this detection isn't perfect on some devices. In such a case, you can change the default method via `position_fix` option.

`position_fix=1` means to use LPIB method explicitly. `position_fix=2` means to use the position-buffer. `position_fix=3` means to use a combination of both methods, needed for some VIA controllers. The capture stream position is corrected by comparing both LPIB and position-buffer values. `position_fix=4` is another combination available for all controllers, and uses LPIB for the playback and the position-buffer for the capture streams. `position_fix=5` is specific to Intel platforms, so far, for Skylake and onward. It applies the delay calculation for the precise position reporting. `position_fix=6` is to correct the position with the fixed FIFO size, mainly targeted for the recent AMD controllers. 0 is the default value for all other controllers, the automatic check and fallback to LPIB as described in the above. If you get a problem of repeated sounds, this option might help.

In addition to that, every controller is known to be broken regarding the wake-up timing. It wakes up a few samples before actually processing the data on the buffer. This caused a lot of problems, for example, with ALSA dmix or JACK. Since 2.6.27 kernel, the driver puts an artificial delay to the wake up timing. This delay is controlled via `bdl_pos_adj` option.

When `bdl_pos_adj` is a negative value (as default), it's assigned to an appropriate value depending on the controller chip. For Intel chips, it'd be 1 while it'd be 32 for others. Usually this works. Only in case it doesn't work and you get warning messages, you should change this parameter to other values.

Codec-Probing Problem

A less often but a more severe problem is the codec probing. When BIOS reports the available codec slots wrongly, the driver gets confused and tries to access the non-existing codec slot. This often results in the total screw-up, and destructs the further communication with the codec chips. The symptom appears usually as error messages like:

```
hda_intel: azx_get_response timeout, switching to polling mode:
      last cmd=0x12345678
hda_intel: azx_get_response timeout, switching to single_cmd mode:
      last cmd=0x12345678
```

The first line is a warning, and this is usually relatively harmless. It means that the codec response isn't notified via an IRQ. The driver uses explicit polling method to read the response. It gives very slight CPU overhead, but you'd unlikely notice it.

The second line is, however, a fatal error. If this happens, usually it means that something is really wrong. Most likely you are accessing a non-existing codec slot.

Thus, if the second error message appears, try to narrow the probed codec slots via `probe_mask` option. It's a bitmask, and each bit corresponds to the codec slot. For example, to probe only

the first slot, pass `probe_mask=1`. For the first and the third slots, pass `probe_mask=5` (where 5 = 1 | 4), and so on.

Since 2.6.29 kernel, the driver has a more robust probing method, so this error might happen rarely, though.

On a machine with a broken BIOS, sometimes you need to force the driver to probe the codec slots the hardware doesn't report for use. In such a case, turn the bit 8 (0x100) of `probe_mask` option on. Then the rest 8 bits are passed as the codec slots to probe unconditionally. For example, `probe_mask=0x103` will force to probe the codec slots 0 and 1 no matter what the hardware reports.

Interrupt Handling

HD-audio driver uses MSI as default (if available) since 2.6.33 kernel as MSI works better on some machines, and in general, it's better for performance. However, Nvidia controllers showed bad regressions with MSI (especially in a combination with AMD chipset), thus we disabled MSI for them.

There seem also still other devices that don't work with MSI. If you see a regression wrt the sound quality (stuttering, etc) or a lock-up in the recent kernel, try to pass `enable_msi=0` option to disable MSI. If it works, you can add the known bad device to the blacklist defined in hda_intel.c. In such a case, please report and give the patch back to the upstream developer.

5.1.3 HD-Audio Codec

Model Option

The most common problem regarding the HD-audio driver is the unsupported codec features or the mismatched device configuration. Most of codec-specific code has several preset models, either to override the BIOS setup or to provide more comprehensive features.

The driver checks PCI SSID and looks through the static configuration table until any matching entry is found. If you have a new machine, you may see a message like below:

```
hda_codec: ALC880: BIOS auto-probing.
```

Meanwhile, in the earlier versions, you would see a message like:

```
hda_codec: Unknown model for ALC880, trying auto-probe from BIOS...
```

Even if you see such a message, DON'T PANIC. Take a deep breath and keep your towel. First of all, it's an informational message, no warning, no error. This means that the PCI SSID of your device isn't listed in the known preset model (white-)list. But, this doesn't mean that the driver is broken. Many codec-drivers provide the automatic configuration mechanism based on the BIOS setup.

The HD-audio codec has usually "pin" widgets, and BIOS sets the default configuration of each pin, which indicates the location, the connection type, the jack color, etc. The HD-audio driver can guess the right connection judging from these default configuration values. However -- some codec-support codes, such as patch_analog.c, don't support the automatic probing (yet as of 2.6.28). And, BIOS is often, yes, pretty often broken. It sets up wrong values and screws up the driver.

The preset model (or recently called as "fix-up") is provided basically to overcome such a situation. When the matching preset model is found in the white-list, the driver assumes the static configuration of that preset with the correct pin setup, etc. Thus, if you have a newer machine with a slightly different PCI SSID (or codec SSID) from the existing one, you may have a good chance to re-use the same model. You can pass the `model` option to specify the preset model instead of PCI (and codec-) SSID look-up.

What `model` option values are available depends on the codec chip. Check your codec chip from the codec proc file (see "Codec Proc-File" section below). It will show the vendor/product name of your codec chip. Then, see *HD-Audio Codec-Specific Models* file, the section of HD-audio driver. You can find a list of codecs and `model` options belonging to each codec. For example, for Realtek ALC262 codec chip, pass `model=ultra` for devices that are compatible with Samsung Q1 Ultra.

Thus, the first thing you can do for any brand-new, unsupported and non-working HD-audio hardware is to check HD-audio codec and several different `model` option values. If you have any luck, some of them might suit with your device well.

There are a few special model option values:

- when 'nofixup' is passed, the device-specific fixups in the codec parser are skipped.
- when `generic` is passed, the codec-specific parser is skipped and only the generic parser is used.

A new style for the model option that was introduced since 5.15 kernel is to pass the PCI or codec SSID in the form of `model=XXXX:YYYY` where XXXX and YYYY are the sub-vendor and sub-device IDs in hex numbers, respectively. This is a kind of aliasing to another device; when this form is given, the driver will refer to that SSID as a reference to the quirk table. It'd be useful especially when the target quirk isn't listed in the model table. For example, passing model=103c:8862 will apply the quirk for HP ProBook 445 G8 (which isn't found in the model table as of writing) as long as the device is handled equivalently by the same driver.

Speaker and Headphone Output

One of the most frequent (and obvious) bugs with HD-audio is the silent output from either or both of a built-in speaker and a headphone jack. In general, you should try a headphone output at first. A speaker output often requires more additional controls like the external amplifier bits. Thus a headphone output has a slightly better chance.

Before making a bug report, double-check whether the mixer is set up correctly. The recent version of snd-hda-intel driver provides mostly "Master" volume control as well as "Front" volume (where Front indicates the front-channels). In addition, there can be individual "Headphone" and "Speaker" controls.

Ditto for the speaker output. There can be "External Amplifier" switch on some codecs. Turn on this if present.

Another related problem is the automatic mute of speaker output by headphone plugging. This feature is implemented in most cases, but not on every preset model or codec-support code.

In anyway, try a different model option if you have such a problem. Some other models may match better and give you more matching functionality. If none of the available models works, send a bug report. See the bug report section for details.

If you are masochistic enough to debug the driver problem, note the following:

- The speaker (and the headphone, too) output often requires the external amplifier. This can be set usually via EAPD verb or a certain GPIO. If the codec pin supports EAPD, you have a better chance via SET_EAPD_BTL verb (0x70c). On others, GPIO pin (mostly it's either GPIO0 or GPIO1) may turn on/off EAPD.

- Some Realtek codecs require special vendor-specific coefficients to turn on the amplifier. See patch_realtek.c.

- IDT codecs may have extra power-enable/disable controls on each analog pin. See patch_sigmatel.c.

- Very rare but some devices don't accept the pin-detection verb until triggered. Issuing GET_PIN_SENSE verb (0xf09) may result in the codec-communication stall. Some examples are found in patch_realtek.c.

Capture Problems

The capture problems are often because of missing setups of mixers. Thus, before submitting a bug report, make sure that you set up the mixer correctly. For example, both "Capture Volume" and "Capture Switch" have to be set properly in addition to the right "Capture Source" or "Input Source" selection. Some devices have "Mic Boost" volume or switch.

When the PCM device is opened via "default" PCM (without pulse-audio plugin), you'll likely have "Digital Capture Volume" control as well. This is provided for the extra gain/attenuation of the signal in software, especially for the inputs without the hardware volume control such as digital microphones. Unless really needed, this should be set to exactly 50%, corresponding to 0dB -- neither extra gain nor attenuation. When you use "hw" PCM, i.e., a raw access PCM, this control will have no influence, though.

It's known that some codecs / devices have fairly bad analog circuits, and the recorded sound contains a certain DC-offset. This is no bug of the driver.

Most of modern laptops have no analog CD-input connection. Thus, the recording from CD input won't work in many cases although the driver provides it as the capture source. Use CDDA instead.

The automatic switching of the built-in and external mic per plugging is implemented on some codec models but not on every model. Partly because of my laziness but mostly lack of testers. Feel free to submit the improvement patch to the author.

Direct Debugging

If no model option gives you a better result, and you are a tough guy to fight against evil, try debugging via hitting the raw HD-audio codec verbs to the device. Some tools are available: hda-emu and hda-analyzer. The detailed description is found in the sections below. You'd need to enable hwdep for using these tools. See "Kernel Configuration" section.

5.1.4 Other Issues

Kernel Configuration

In general, I recommend you to enable the sound debug option, `CONFIG_SND_DEBUG=y`, no matter whether you are debugging or not. This enables *snd_printd()* macro and others, and you'll get additional kernel messages at probing.

In addition, you can enable `CONFIG_SND_DEBUG_VERBOSE=y`. But this will give you far more messages. Thus turn this on only when you are sure to want it.

Don't forget to turn on the appropriate `CONFIG_SND_HDA_CODEC_*` options. Note that each of them corresponds to the codec chip, not the controller chip. Thus, even if lspci shows the Nvidia controller, you may need to choose the option for other vendors. If you are unsure, just select all yes.

`CONFIG_SND_HDA_HWDEP` is a useful option for debugging the driver. When this is enabled, the driver creates hardware-dependent devices (one per each codec), and you have a raw access to the device via these device files. For example, `hwC0D2` will be created for the codec slot #2 of the first card (#0). For debug-tools such as hda-verb and hda-analyzer, the hwdep device has to be enabled. Thus, it'd be better to turn this on always.

`CONFIG_SND_HDA_RECONFIG` is a new option, and this depends on the hwdep option above. When enabled, you'll have some sysfs files under the corresponding hwdep directory. See "HD-audio reconfiguration" section below.

`CONFIG_SND_HDA_POWER_SAVE` option enables the power-saving feature. See "Power-saving" section below.

Codec Proc-File

The codec proc-file is a treasure-chest for debugging HD-audio. It shows most of useful information of each codec widget.

The proc file is located in /proc/asound/card*/codec#*, one file per each codec slot. You can know the codec vendor, product id and names, the type of each widget, capabilities and so on. This file, however, doesn't show the jack sensing state, so far. This is because the jack-sensing might be depending on the trigger state.

This file will be picked up by the debug tools, and also it can be fed to the emulator as the primary codec information. See the debug tools section below.

This proc file can be also used to check whether the generic parser is used. When the generic parser is used, the vendor/product ID name will appear as "Realtek ID 0262", instead of "Realtek ALC262".

HD-Audio Reconfiguration

This is an experimental feature to allow you re-configure the HD-audio codec dynamically without reloading the driver. The following sysfs files are available under each codec-hwdep device directory (e.g. /sys/class/sound/hwC0D0):

vendor_id
> Shows the 32bit codec vendor-id hex number. You can change the vendor-id value by writing to this file.

subsystem_id
> Shows the 32bit codec subsystem-id hex number. You can change the subsystem-id value by writing to this file.

revision_id
> Shows the 32bit codec revision-id hex number. You can change the revision-id value by writing to this file.

afg
> Shows the AFG ID. This is read-only.

mfg
> Shows the MFG ID. This is read-only.

name
> Shows the codec name string. Can be changed by writing to this file.

modelname
> Shows the currently set `model` option. Can be changed by writing to this file.

init_verbs
> The extra verbs to execute at initialization. You can add a verb by writing to this file. Pass three numbers: nid, verb and parameter (separated with a space).

hints
> Shows / stores hint strings for codec parsers for any use. Its format is `key = value`. For example, passing `jack_detect = no` will disable the jack detection of the machine completely.

init_pin_configs
> Shows the initial pin default config values set by BIOS.

driver_pin_configs
> Shows the pin default values set by the codec parser explicitly. This doesn't show all pin values but only the changed values by the parser. That is, if the parser doesn't change the pin default config values by itself, this will contain nothing.

user_pin_configs
> Shows the pin default config values to override the BIOS setup. Writing this (with two numbers, NID and value) appends the new value. The given will be used instead of the initial BIOS value at the next reconfiguration time. Note that this config will override even the driver pin configs, too.

reconfig
> Triggers the codec re-configuration. When any value is written to this file, the driver re-initialize and parses the codec tree again. All the changes done by the sysfs entries above are taken into account.

clear

Resets the codec, removes the mixer elements and PCM stuff of the specified codec, and clear all init verbs and hints.

For example, when you want to change the pin default configuration value of the pin widget 0x14 to 0x9993013f, and let the driver re-configure based on that state, run like below:

```
# echo 0x14 0x9993013f > /sys/class/sound/hwC0D0/user_pin_configs
# echo 1 > /sys/class/sound/hwC0D0/reconfig
```

Hint Strings

The codec parser have several switches and adjustment knobs for matching better with the actual codec or device behavior. Many of them can be adjusted dynamically via "hints" strings as mentioned in the section above. For example, by passing `jack_detect = no` string via sysfs or a patch file, you can disable the jack detection, thus the codec parser will skip the features like auto-mute or mic auto-switch. As a boolean value, either `yes`, `no`, `true`, `false`, `1` or `0` can be passed.

The generic parser supports the following hints:

jack_detect (bool)

specify whether the jack detection is available at all on this machine; default true

inv_jack_detect (bool)

indicates that the jack detection logic is inverted

trigger_sense (bool)

indicates that the jack detection needs the explicit call of AC_VERB_SET_PIN_SENSE verb

inv_eapd (bool)

indicates that the EAPD is implemented in the inverted logic

pcm_format_first (bool)

sets the PCM format before the stream tag and channel ID

sticky_stream (bool)

keep the PCM format, stream tag and ID as long as possible; default true

spdif_status_reset (bool)

reset the SPDIF status bits at each time the SPDIF stream is set up

pin_amp_workaround (bool)

the output pin may have multiple amp values

single_adc_amp (bool)

ADCs can have only single input amps

auto_mute (bool)

enable/disable the headphone auto-mute feature; default true

auto_mic (bool)

enable/disable the mic auto-switch feature; default true

line_in_auto_switch (bool)

enable/disable the line-in auto-switch feature; default false

need_dac_fix (bool)
> limits the DACs depending on the channel count

primary_hp (bool)
> probe headphone jacks as the primary outputs; default true

multi_io (bool)
> try probing multi-I/O config (e.g. shared line-in/surround, mic/clfe jacks)

multi_cap_vol (bool)
> provide multiple capture volumes

inv_dmic_split (bool)
> provide split internal mic volume/switch for phase-inverted digital mics

indep_hp (bool)
> provide the independent headphone PCM stream and the corresponding mixer control, if available

add_stereo_mix_input (bool)
> add the stereo mix (analog-loopback mix) to the input mux if available

add_jack_modes (bool)
> add "xxx Jack Mode" enum controls to each I/O jack for allowing to change the headphone amp and mic bias VREF capabilities

power_save_node (bool)
> advanced power management for each widget, controlling the power state (D0/D3) of each widget node depending on the actual pin and stream states

power_down_unused (bool)
> power down the unused widgets, a subset of power_save_node, and will be dropped in future

add_hp_mic (bool)
> add the headphone to capture source if possible

hp_mic_detect (bool)
> enable/disable the hp/mic shared input for a single built-in mic case; default true

vmaster (bool)
> enable/disable the virtual Master control; default true

mixer_nid (int)
> specifies the widget NID of the analog-loopback mixer

Early Patching

When `CONFIG_SND_HDA_PATCH_LOADER=y` is set, you can pass a "patch" as a firmware file for modifying the HD-audio setup before initializing the codec. This can work basically like the reconfiguration via sysfs in the above, but it does it before the first codec configuration.

A patch file is a plain text file which looks like below:

```
[codec]
0x12345678 0xabcd1234 2

[model]
```

```
auto

[pincfg]
0x12 0x411111f0

[verb]
0x20 0x500 0x03
0x20 0x400 0xff

[hint]
jack_detect = no
```

The file needs to have a line `[codec]`. The next line should contain three numbers indicating the codec vendor-id (0x12345678 in the example), the codec subsystem-id (0xabcd1234) and the address (2) of the codec. The rest patch entries are applied to this specified codec until another codec entry is given. Passing 0 or a negative number to the first or the second value will make the check of the corresponding field be skipped. It'll be useful for really broken devices that don't initialize SSID properly.

The `[model]` line allows to change the model name of the each codec. In the example above, it will be changed to model=auto. Note that this overrides the module option.

After the `[pincfg]` line, the contents are parsed as the initial default pin-configurations just like `user_pin_configs` sysfs above. The values can be shown in user_pin_configs sysfs file, too.

Similarly, the lines after `[verb]` are parsed as `init_verbs` sysfs entries, and the lines after `[hint]` are parsed as `hints` sysfs entries, respectively.

Another example to override the codec vendor id from 0x12345678 to 0xdeadbeef is like below:

```
[codec]
0x12345678 0xabcd1234 2

[vendor_id]
0xdeadbeef
```

In the similar way, you can override the codec subsystem_id via `[subsystem_id]`, the revision id via `[revision_id]` line. Also, the codec chip name can be rewritten via `[chip_name]` line.

```
[codec]
0x12345678 0xabcd1234 2

[subsystem_id]
0xffff1111

[revision_id]
0x10

[chip_name]
My-own NEWS-0002
```

The hd-audio driver reads the file via request_firmware(). Thus, a patch file has to be located on the appropriate firmware path, typically, /lib/firmware. For example, when you pass the option

patch=hda-init.fw, the file /lib/firmware/hda-init.fw must be present.

The patch module option is specific to each card instance, and you need to give one file name for each instance, separated by commas. For example, if you have two cards, one for an on-board analog and one for an HDMI video board, you may pass patch option like below:

```
options snd-hda-intel patch=on-board-patch,hdmi-patch
```

Power-Saving

The power-saving is a kind of auto-suspend of the device. When the device is inactive for a certain time, the device is automatically turned off to save the power. The time to go down is specified via power_save module option, and this option can be changed dynamically via sysfs.

The power-saving won't work when the analog loopback is enabled on some codecs. Make sure that you mute all unneeded signal routes when you want the power-saving.

The power-saving feature might cause audible click noises at each power-down/up depending on the device. Some of them might be solvable, but some are hard, I'm afraid. Some distros such as openSUSE enables the power-saving feature automatically when the power cable is unplugged. Thus, if you hear noises, suspect first the power-saving. See /sys/module/snd_hda_intel/parameters/power_save to check the current value. If it's non-zero, the feature is turned on.

The recent kernel supports the runtime PM for the HD-audio controller chip, too. It means that the HD-audio controller is also powered up / down dynamically. The feature is enabled only for certain controller chips like Intel LynxPoint. You can enable/disable this feature forcibly by setting power_save_controller option, which is also available at /sys/module/snd_hda_intel/parameters directory.

Tracepoints

The hd-audio driver gives a few basic tracepoints. hda:hda_send_cmd traces each CORB write while hda:hda_get_response traces the response from RIRB (only when read from the codec driver). hda:hda_bus_reset traces the bus-reset due to fatal error, etc, hda:hda_unsol_event traces the unsolicited events, and hda:hda_power_down and hda:hda_power_up trace the power down/up via power-saving behavior.

Enabling all tracepoints can be done like

```
# echo 1 > /sys/kernel/tracing/events/hda/enable
```

then after some commands, you can traces from /sys/kernel/tracing/trace file. For example, when you want to trace what codec command is sent, enable the tracepoint like:

```
# cat /sys/kernel/tracing/trace
# tracer: nop
#
#        TASK-PID    CPU#      TIMESTAMP  FUNCTION
#           | |        |          |          |
       <...>-7807   [002] 105147.774889: hda_send_cmd: [0:0] val=e3a019
       <...>-7807   [002] 105147.774893: hda_send_cmd: [0:0] val=e39019
```

```
        <...>-7807  [002] 105147.999542: hda_send_cmd: [0:0] val=e3a01a
        <...>-7807  [002] 105147.999543: hda_send_cmd: [0:0] val=e3901a
        <...>-26764 [001] 349222.837143: hda_send_cmd: [0:0] val=e3a019
        <...>-26764 [001] 349222.837148: hda_send_cmd: [0:0] val=e39019
        <...>-26764 [001] 349223.058539: hda_send_cmd: [0:0] val=e3a01a
        <...>-26764 [001] 349223.058541: hda_send_cmd: [0:0] val=e3901a
```

Here `[0:0]` indicates the card number and the codec address, and `val` shows the value sent to the codec, respectively. The value is a packed value, and you can decode it via hda-decode-verb program included in hda-emu package below. For example, the value e3a019 is to set the left output-amp value to 25.

```
% hda-decode-verb 0xe3a019
raw value = 0x00e3a019
cid = 0, nid = 0x0e, verb = 0x3a0, parm = 0x19
raw value: verb = 0x3a0, parm = 0x19
verbname = set_amp_gain_mute
amp raw val = 0xa019
output, left, idx=0, mute=0, val=25
```

Development Tree

The latest development codes for HD-audio are found on sound git tree:

- git://git.kernel.org/pub/scm/linux/kernel/git/tiwai/sound.git

The master branch or for-next branches can be used as the main development branches in general while the development for the current and next kernels are found in for-linus and for-next branches, respectively.

Sending a Bug Report

If any model or module options don't work for your device, it's time to send a bug report to the developers. Give the following in your bug report:

- Hardware vendor, product and model names
- Kernel version (and ALSA-driver version if you built externally)
- `alsa-info.sh` output; run with `--no-upload` option. See the section below about alsa-info

If it's a regression, at best, send alsa-info outputs of both working and non-working kernels. This is really helpful because we can compare the codec registers directly.

Send a bug report either the following:

kernel-bugzilla
    https://bugzilla.kernel.org/

alsa-devel ML
    alsa-devel@alsa-project.org

5.1.5 Debug Tools

This section describes some tools available for debugging HD-audio problems.

alsa-info

The script `alsa-info.sh` is a very useful tool to gather the audio device information. It's included in alsa-utils package. The latest version can be found on git repository:

- git://git.alsa-project.org/alsa-utils.git

The script can be fetched directly from the following URL, too:

- https://www.alsa-project.org/alsa-info.sh

Run this script as root, and it will gather the important information such as the module lists, module parameters, proc file contents including the codec proc files, mixer outputs and the control elements. As default, it will store the information onto a web server on alsa-project.org. But, if you send a bug report, it'd be better to run with `--no-upload` option, and attach the generated file.

There are some other useful options. See `--help` option output for details.

When a probe error occurs or when the driver obviously assigns a mismatched model, it'd be helpful to load the driver with `probe_only=1` option (at best after the cold reboot) and run alsa-info at this state. With this option, the driver won't configure the mixer and PCM but just tries to probe the codec slot. After probing, the proc file is available, so you can get the raw codec information before modified by the driver. Of course, the driver isn't usable with `probe_only=1`. But you can continue the configuration via hwdep sysfs file if hda-reconfig option is enabled. Using `probe_only` mask 2 skips the reset of HDA codecs (use `probe_only=3` as module option). The hwdep interface can be used to determine the BIOS codec initialization.

hda-verb

hda-verb is a tiny program that allows you to access the HD-audio codec directly. You can execute a raw HD-audio codec verb with this. This program accesses the hwdep device, thus you need to enable the kernel config `CONFIG_SND_HDA_HWDEP=y` beforehand.

The hda-verb program takes four arguments: the hwdep device file, the widget NID, the verb and the parameter. When you access to the codec on the slot 2 of the card 0, pass /dev/snd/hwC0D2 to the first argument, typically. (However, the real path name depends on the system.)

The second parameter is the widget number-id to access. The third parameter can be either a hex/digit number or a string corresponding to a verb. Similarly, the last parameter is the value to write, or can be a string for the parameter type.

```
% hda-verb /dev/snd/hwC0D0 0x12 0x701 2
nid = 0x12, verb = 0x701, param = 0x2
value = 0x0

% hda-verb /dev/snd/hwC0D0 0x0 PARAMETERS VENDOR_ID
nid = 0x0, verb = 0xf00, param = 0x0
value = 0x10ec0262
```

```
% hda-verb /dev/snd/hwC0D0 2 set_a 0xb080
nid = 0x2, verb = 0x300, param = 0xb080
value = 0x0
```

Although you can issue any verbs with this program, the driver state won't be always updated. For example, the volume values are usually cached in the driver, and thus changing the widget amp value directly via hda-verb won't change the mixer value.

The hda-verb program is included now in alsa-tools:

- git://git.alsa-project.org/alsa-tools.git

Also, the old stand-alone package is found in the ftp directory:

- ftp://ftp.suse.com/pub/people/tiwai/misc/

Also a git repository is available:

- git://git.kernel.org/pub/scm/linux/kernel/git/tiwai/hda-verb.git

See README file in the tarball for more details about hda-verb program.

hda-analyzer

hda-analyzer provides a graphical interface to access the raw HD-audio control, based on pyGTK2 binding. It's a more powerful version of hda-verb. The program gives you an easy-to-use GUI stuff for showing the widget information and adjusting the amp values, as well as the proc-compatible output.

The hda-analyzer:

- https://git.alsa-project.org/?p=alsa.git;a=tree;f=hda-analyzer

is a part of alsa.git repository in alsa-project.org:

- git://git.alsa-project.org/alsa.git

Codecgraph

Codecgraph is a utility program to generate a graph and visualizes the codec-node connection of a codec chip. It's especially useful when you analyze or debug a codec without a proper datasheet. The program parses the given codec proc file and converts to SVG via graphiz program.

The tarball and GIT trees are found in the web page at:

- http://helllabs.org/codecgraph/

hda-emu

hda-emu is an HD-audio emulator. The main purpose of this program is to debug an HD-audio codec without the real hardware. Thus, it doesn't emulate the behavior with the real audio I/O, but it just dumps the codec register changes and the ALSA-driver internal changes at probing and operating the HD-audio driver.

The program requires a codec proc-file to simulate. Get a proc file for the target codec beforehand, or pick up an example codec from the codec proc collections in the tarball. Then, run the program with the proc file, and the hda-emu program will start parsing the codec file and simulates the HD-audio driver:

```
% hda-emu codecs/stac9200-dell-d820-laptop
# Parsing..
hda_codec: Unknown model for STAC9200, using BIOS defaults
hda_codec: pin nid 08 bios pin config 40c003fa
....
```

The program gives you only a very dumb command-line interface. You can get a proc-file dump at the current state, get a list of control (mixer) elements, set/get the control element value, simulate the PCM operation, the jack plugging simulation, etc.

The program is found in the git repository below:

- git://git.kernel.org/pub/scm/linux/kernel/git/tiwai/hda-emu.git

See README file in the repository for more details about hda-emu program.

hda-jack-retask

hda-jack-retask is a user-friendly GUI program to manipulate the HD-audio pin control for jack retasking. If you have a problem about the jack assignment, try this program and check whether you can get useful results. Once when you figure out the proper pin assignment, it can be fixed either in the driver code statically or via passing a firmware patch file (see "Early Patching" section).

The program is included in alsa-tools now:

- git://git.alsa-project.org/alsa-tools.git

5.2 HD-Audio Codec-Specific Models

5.2.1 ALC880

3stack
    3-jack in back and a headphone out

3stack-digout
    3-jack in back, a HP out and a SPDIF out

5stack
    5-jack in back, 2-jack in front

5stack-digout
    5-jack in back, 2-jack in front, a SPDIF out

6stack
    6-jack in back, 2-jack in front

6stack-digout
    6-jack with a SPDIF out

6stack-automute
    6-jack with headphone jack detection

5.2.2 ALC260

gpio1
    Enable GPIO1

coef
    Enable EAPD via COEF table

fujitsu
    Quirk for FSC S7020

fujitsu-jwse
    Quirk for FSC S7020 with jack modes and HP mic support

5.2.3 ALC262

inv-dmic
    Inverted internal mic workaround

fsc-h270
    Fixups for Fujitsu-Siemens Celsius H270

fsc-s7110
    Fixups for Fujitsu-Siemens Lifebook S7110

hp-z200
    Fixups for HP Z200

tyan
    Fixups for Tyan Thunder n6650W

lenovo-3000
    Fixups for Lenovo 3000

benq
    Fixups for Benq ED8

benq-t31
    Fixups for Benq T31

bayleybay
    Fixups for Intel BayleyBay

5.2.4 ALC267/268

inv-dmic
> Inverted internal mic workaround

hp-eapd
> Disable HP EAPD on NID 0x15

spdif
> Enable SPDIF output on NID 0x1e

5.2.5 ALC22x/23x/25x/269/27x/28x/29x (and vendor-specific ALC3xxx models)

laptop-amic
> Laptops with analog-mic input

laptop-dmic
> Laptops with digital-mic input

alc269-dmic
> Enable ALC269(VA) digital mic workaround

alc271-dmic
> Enable ALC271X digital mic workaround

inv-dmic
> Inverted internal mic workaround

headset-mic
> Indicates a combined headset (headphone+mic) jack

headset-mode
> More comprehensive headset support for ALC269 & co

headset-mode-no-hp-mic
> Headset mode support without headphone mic

lenovo-dock
> Enables docking station I/O for some Lenovos

hp-gpio-led
> GPIO LED support on HP laptops

hp-dock-gpio-mic1-led
> HP dock with mic LED support

dell-headset-multi
> Headset jack, which can also be used as mic-in

dell-headset-dock
> Headset jack (without mic-in), and also dock I/O

dell-headset3
> Headset jack (without mic-in), and also dock I/O, variant 3

dell-headset4
> Headset jack (without mic-in), and also dock I/O, variant 4

alc283-dac-wcaps
> Fixups for Chromebook with ALC283

alc283-sense-combo
> Combo jack sensing on ALC283

tpt440-dock
> Pin configs for Lenovo Thinkpad Dock support

tpt440
> Lenovo Thinkpad T440s setup

tpt460
> Lenovo Thinkpad T460/560 setup

tpt470-dock
> Lenovo Thinkpad T470 dock setup

dual-codecs
> Lenovo laptops with dual codecs

alc700-ref
> Intel reference board with ALC700 codec

vaio
> Pin fixups for Sony VAIO laptops

dell-m101z
> COEF setup for Dell M101z

asus-g73jw
> Subwoofer pin fixup for ASUS G73JW

lenovo-eapd
> Inversed EAPD setup for Lenovo laptops

sony-hweq
> H/W EQ COEF setup for Sony laptops

pcm44k
> Fixed PCM 44kHz constraints (for buggy devices)

lifebook
> Dock pin fixups for Fujitsu Lifebook

lifebook-extmic
> Headset mic fixup for Fujitsu Lifebook

lifebook-hp-pin
> Headphone pin fixup for Fujitsu Lifebook

lifebook-u7x7
> Lifebook U7x7 fixups

alc269vb-amic
> ALC269VB analog mic pin fixups

alc269vb-dmic
> ALC269VB digital mic pin fixups

hp-mute-led-mic1
   Mute LED via Mic1 pin on HP

hp-mute-led-mic2
   Mute LED via Mic2 pin on HP

hp-mute-led-mic3
   Mute LED via Mic3 pin on HP

hp-gpio-mic1
   GPIO + Mic1 pin LED on HP

hp-line1-mic1
   Mute LED via Line1 + Mic1 pins on HP

noshutup
   Skip shutup callback

sony-nomic
   Headset mic fixup for Sony laptops

aspire-headset-mic
   Headset pin fixup for Acer Aspire

asus-x101
   ASUS X101 fixups

acer-ao7xx
   Acer AO7xx fixups

acer-aspire-e1
   Acer Aspire E1 fixups

acer-ac700
   Acer AC700 fixups

limit-mic-boost
   Limit internal mic boost on Lenovo machines

asus-zenbook
   ASUS Zenbook fixups

asus-zenbook-ux31a
   ASUS Zenbook UX31A fixups

ordissimo
   Ordissimo EVE2 (or Malata PC-B1303) fixups

asus-tx300
   ASUS TX300 fixups

alc283-int-mic
   ALC283 COEF setup for Lenovo machines

mono-speakers
   Subwoofer and headset fixupes for Dell Inspiron

alc290-subwoofer
   Subwoofer fixups for Dell Vostro

thinkpad
Binding with thinkpad_acpi driver for Lenovo machines

dmic-thinkpad
thinkpad_acpi binding + digital mic support

alc255-acer
ALC255 fixups on Acer machines

alc255-asus
ALC255 fixups on ASUS machines

alc255-dell1
ALC255 fixups on Dell machines

alc255-dell2
ALC255 fixups on Dell machines, variant 2

alc293-dell1
ALC293 fixups on Dell machines

alc283-headset
Headset pin fixups on ALC283

aspire-v5
Acer Aspire V5 fixups

hp-gpio4
GPIO and Mic1 pin mute LED fixups for HP

hp-gpio-led
GPIO mute LEDs on HP

hp-gpio2-hotkey
GPIO mute LED with hot key handling on HP

hp-dock-pins
GPIO mute LEDs and dock support on HP

hp-dock-gpio-mic
GPIO, Mic mute LED and dock support on HP

hp-9480m
HP 9480m fixups

alc288-dell1
ALC288 fixups on Dell machines

alc288-dell-xps13
ALC288 fixups on Dell XPS13

dell-e7x
Dell E7x fixups

alc293-dell
ALC293 fixups on Dell machines

alc298-dell1
ALC298 fixups on Dell machines

alc298-dell-aio
    ALC298 fixups on Dell AIO machines

alc275-dell-xps
    ALC275 fixups on Dell XPS models

lenovo-spk-noise
    Workaround for speaker noise on Lenovo machines

lenovo-hotkey
    Hot-key support via Mic2 pin on Lenovo machines

dell-spk-noise
    Workaround for speaker noise on Dell machines

alc255-dell1
    ALC255 fixups on Dell machines

alc295-disable-dac3
    Disable DAC3 routing on ALC295

alc280-hp-headset
    HP Elitebook fixups

alc221-hp-mic
    Front mic pin fixup on HP machines

alc298-spk-volume
    Speaker pin routing workaround on ALC298

dell-inspiron-7559
    Dell Inspiron 7559 fixups

ativ-book
    Samsung Ativ book 8 fixups

alc221-hp-mic
    ALC221 headset fixups on HP machines

alc256-asus-mic
    ALC256 fixups on ASUS machines

alc256-asus-aio
    ALC256 fixups on ASUS AIO machines

alc233-eapd
    ALC233 fixups on ASUS machines

alc294-lenovo-mic
    ALC294 Mic pin fixup for Lenovo AIO machines

alc225-wyse
    Dell Wyse fixups

alc274-dell-aio
    ALC274 fixups on Dell AIO machines

alc255-dummy-lineout
    Dell Precision 3930 fixups

alc255-dell-headset
    Dell Precision 3630 fixups

alc295-hp-x360
    HP Spectre X360 fixups

alc-sense-combo
    Headset button support for Chrome platform

huawei-mbx-stereo
    Enable initialization verbs for Huawei MBX stereo speakers; might be risky, try this at your own risk

alc298-samsung-headphone
    Samsung laptops with ALC298

alc256-samsung-headphone
    Samsung laptops with ALC256

5.2.6 ALC66x/67x/892

aspire
    Subwoofer pin fixup for Aspire laptops

ideapad
    Subwoofer pin fixup for Ideapad laptops

mario
    Chromebook mario model fixup

hp-rp5800
    Headphone pin fixup for HP RP5800

asus-mode1
    ASUS

asus-mode2
    ASUS

asus-mode3
    ASUS

asus-mode4
    ASUS

asus-mode5
    ASUS

asus-mode6
    ASUS

asus-mode7
    ASUS

asus-mode8
    ASUS

zotac-z68
    Front HP fixup for Zotac Z68

inv-dmic
Inverted internal mic workaround

alc662-headset-multi
Dell headset jack, which can also be used as mic-in (ALC662)

dell-headset-multi
Headset jack, which can also be used as mic-in

alc662-headset
Headset mode support on ALC662

alc668-headset
Headset mode support on ALC668

bass16
Bass speaker fixup on pin 0x16

bass1a
Bass speaker fixup on pin 0x1a

automute
Auto-mute fixups for ALC668

dell-xps13
Dell XPS13 fixups

asus-nx50
ASUS Nx50 fixups

asus-nx51
ASUS Nx51 fixups

asus-g751
ASUS G751 fixups

alc891-headset
Headset mode support on ALC891

alc891-headset-multi
Dell headset jack, which can also be used as mic-in (ALC891)

acer-veriton
Acer Veriton speaker pin fixup

asrock-mobo
Fix invalid 0x15 / 0x16 pins

usi-headset
Headset support on USI machines

dual-codecs
Lenovo laptops with dual codecs

alc285-hp-amp-init
HP laptops which require speaker amplifier initialization (ALC285)

5.2.7 ALC680

N/A

5.2.8 ALC88x/898/1150/1220

abit-aw9d
Pin fixups for Abit AW9D-MAX

lenovo-y530
Pin fixups for Lenovo Y530

acer-aspire-7736
Fixup for Acer Aspire 7736

asus-w90v
Pin fixup for ASUS W90V

cd
Enable audio CD pin NID 0x1c

no-front-hp
Disable front HP pin NID 0x1b

vaio-tt
Pin fixup for VAIO TT

eee1601
COEF setups for ASUS Eee 1601

alc882-eapd
Change EAPD COEF mode on ALC882

alc883-eapd
Change EAPD COEF mode on ALC883

gpio1
Enable GPIO1

gpio2
Enable GPIO2

gpio3
Enable GPIO3

alc889-coef
Setup ALC889 COEF

asus-w2jc
Fixups for ASUS W2JC

acer-aspire-4930g
Acer Aspire 4930G/5930G/6530G/6930G/7730G

acer-aspire-8930g
Acer Aspire 8330G/6935G

acer-aspire
Acer Aspire others

macpro-gpio
 GPIO setup for Mac Pro

dac-route
 Workaround for DAC routing on Acer Aspire

mbp-vref
 Vref setup for Macbook Pro

imac91-vref
 Vref setup for iMac 9,1

mba11-vref
 Vref setup for MacBook Air 1,1

mba21-vref
 Vref setup for MacBook Air 2,1

mp11-vref
 Vref setup for Mac Pro 1,1

mp41-vref
 Vref setup for Mac Pro 4,1

inv-dmic
 Inverted internal mic workaround

no-primary-hp
 VAIO Z/VGC-LN51JGB workaround (for fixed speaker DAC)

asus-bass
 Bass speaker setup for ASUS ET2700

dual-codecs
 ALC1220 dual codecs for Gaming mobos

clevo-p950
 Fixups for Clevo P950

5.2.9 ALC861/660

N/A

5.2.10 ALC861VD/660VD

N/A

5.2.11 CMI9880

minimal
> 3-jack in back

min_fp
> 3-jack in back, 2-jack in front

full
> 6-jack in back, 2-jack in front

full_dig
> 6-jack in back, 2-jack in front, SPDIF I/O

allout
> 5-jack in back, 2-jack in front, SPDIF out

auto
> auto-config reading BIOS (default)

5.2.12 AD1882 / AD1882A

3stack
> 3-stack mode

3stack-automute
> 3-stack with automute front HP (default)

6stack
> 6-stack mode

5.2.13 AD1884A / AD1883 / AD1984A / AD1984B

desktop 3-stack desktop (default) laptop laptop with HP jack sensing mobile mobile devices with HP jack sensing thinkpad Lenovo Thinkpad X300 touchsmart HP Touchsmart

5.2.14 AD1884

N/A

5.2.15 AD1981

basic 3-jack (default) hp HP nx6320 thinkpad Lenovo Thinkpad T60/X60/Z60 toshiba Toshiba U205

5.2.16 AD1983

N/A

5.2.17 AD1984

basic default configuration thinkpad Lenovo Thinkpad T61/X61 dell_desktop Dell T3400

5.2.18 AD1986A

3stack
3-stack, shared surrounds

laptop
2-channel only (FSC V2060, Samsung M50)

laptop-imic
2-channel with built-in mic

eapd
Turn on EAPD constantly

5.2.19 AD1988/AD1988B/AD1989A/AD1989B

6stack
6-jack

6stack-dig
ditto with SPDIF

3stack
3-jack

3stack-dig
ditto with SPDIF

laptop
3-jack with hp-jack automute

laptop-dig
ditto with SPDIF

auto
auto-config reading BIOS (default)

5.2.20 Conexant 5045

cap-mix-amp
> Fix max input level on mixer widget

toshiba-p105
> Toshiba P105 quirk

hp-530
> HP 530 quirk

5.2.21 Conexant 5047

cap-mix-amp
> Fix max input level on mixer widget

5.2.22 Conexant 5051

lenovo-x200
> Lenovo X200 quirk

5.2.23 Conexant 5066

stereo-dmic
> Workaround for inverted stereo digital mic

gpio1
> Enable GPIO1 pin

headphone-mic-pin
> Enable headphone mic NID 0x18 without detection

tp410
> Thinkpad T400 & co quirks

thinkpad
> Thinkpad mute/mic LED quirk

lemote-a1004
> Lemote A1004 quirk

lemote-a1205
> Lemote A1205 quirk

olpc-xo
> OLPC XO quirk

mute-led-eapd
> Mute LED control via EAPD

hp-dock
> HP dock support

mute-led-gpio
> Mute LED control via GPIO

hp-mic-fix
> Fix for headset mic pin on HP boxes

5.2.24 STAC9200

ref
> Reference board

oqo
> OQO Model 2

dell-d21
> Dell (unknown)

dell-d22
> Dell (unknown)

dell-d23
> Dell (unknown)

dell-m21
> Dell Inspiron 630m, Dell Inspiron 640m

dell-m22
> Dell Latitude D620, Dell Latitude D820

dell-m23
> Dell XPS M1710, Dell Precision M90

dell-m24
> Dell Latitude 120L

dell-m25
> Dell Inspiron E1505n

dell-m26
> Dell Inspiron 1501

dell-m27
> Dell Inspiron E1705/9400

gateway-m4
> Gateway laptops with EAPD control

gateway-m4-2
> Gateway laptops with EAPD control

panasonic
> Panasonic CF-74

auto
> BIOS setup (default)

5.2.25 STAC9205/9254

ref
> Reference board

dell-m42
> Dell (unknown)

dell-m43
> Dell Precision

dell-m44
> Dell Inspiron

eapd
> Keep EAPD on (e.g. Gateway T1616)

auto
> BIOS setup (default)

5.2.26 STAC9220/9221

ref
> Reference board

3stack
> D945 3stack

5stack
> D945 5stack + SPDIF

intel-mac-v1
> Intel Mac Type 1

intel-mac-v2
> Intel Mac Type 2

intel-mac-v3
> Intel Mac Type 3

intel-mac-v4
> Intel Mac Type 4

intel-mac-v5
> Intel Mac Type 5

intel-mac-auto
> Intel Mac (detect type according to subsystem id)

macmini
> Intel Mac Mini (equivalent with type 3)

macbook
> Intel Mac Book (eq. type 5)

macbook-pro-v1
> Intel Mac Book Pro 1st generation (eq. type 3)

macbook-pro
    Intel Mac Book Pro 2nd generation (eq. type 3)

imac-intel
    Intel iMac (eq. type 2)

imac-intel-20
    Intel iMac (newer version) (eq. type 3)

ecs202
    ECS/PC chips

dell-d81
    Dell (unknown)

dell-d82
    Dell (unknown)

dell-m81
    Dell (unknown)

dell-m82
    Dell XPS M1210

auto
    BIOS setup (default)

5.2.27 STAC9202/9250/9251

ref
    Reference board, base config

m1
    Some Gateway MX series laptops (NX560XL)

m1-2
    Some Gateway MX series laptops (MX6453)

m2
    Some Gateway MX series laptops (M255)

m2-2
    Some Gateway MX series laptops

m3
    Some Gateway MX series laptops

m5
    Some Gateway MX series laptops (MP6954)

m6
    Some Gateway NX series laptops

auto
    BIOS setup (default)

5.2.28 STAC9227/9228/9229/927x

ref
Reference board

ref-no-jd
Reference board without HP/Mic jack detection

3stack
D965 3stack

5stack
D965 5stack + SPDIF

5stack-no-fp
D965 5stack without front panel

dell-3stack
Dell Dimension E520

dell-bios
Fixes with Dell BIOS setup

dell-bios-amic
Fixes with Dell BIOS setup including analog mic

volknob
Fixes with volume-knob widget 0x24

auto
BIOS setup (default)

5.2.29 STAC92HD71B*

ref
Reference board

dell-m4-1
Dell desktops

dell-m4-2
Dell desktops

dell-m4-3
Dell desktops

hp-m4
HP mini 1000

hp-dv5
HP dv series

hp-hdx
HP HDX series

hp-dv4-1222nr
HP dv4-1222nr (with LED support)

auto
    BIOS setup (default)

5.2.30 STAC92HD73*

ref
    Reference board

no-jd
    BIOS setup but without jack-detection

intel
    Intel D*45* mobos

dell-m6-amic
    Dell desktops/laptops with analog mics

dell-m6-dmic
    Dell desktops/laptops with digital mics

dell-m6
    Dell desktops/laptops with both type of mics

dell-eq
    Dell desktops/laptops

alienware
    Alienware M17x

asus-mobo
    Pin configs for ASUS mobo with 5.1/SPDIF out

auto
    BIOS setup (default)

5.2.31 STAC92HD83*

ref
    Reference board

mic-ref
    Reference board with power management for ports

dell-s14
    Dell laptop

dell-vostro-3500
    Dell Vostro 3500 laptop

hp-dv7-4000
    HP dv-7 4000

hp_cNB11_intquad
    HP CNB models with 4 speakers

hp-zephyr
    HP Zephyr

hp-led
> HP with broken BIOS for mute LED

hp-inv-led
> HP with broken BIOS for inverted mute LED

hp-mic-led
> HP with mic-mute LED

headset-jack
> Dell Latitude with a 4-pin headset jack

hp-envy-bass
> Pin fixup for HP Envy bass speaker (NID 0x0f)

hp-envy-ts-bass
> Pin fixup for HP Envy TS bass speaker (NID 0x10)

hp-bnb13-eq
> Hardware equalizer setup for HP laptops

hp-envy-ts-bass
> HP Envy TS bass support

auto
> BIOS setup (default)

5.2.32 STAC92HD95

hp-led
> LED support for HP laptops

hp-bass
> Bass HPF setup for HP Spectre 13

5.2.33 STAC9872

vaio
> VAIO laptop without SPDIF

auto
> BIOS setup (default)

5.2.34 Cirrus Logic CS4206/4207

mbp53
> MacBook Pro 5,3

mbp55
> MacBook Pro 5,5

imac27
> IMac 27 Inch

imac27_122
> iMac 12,2

apple
>   Generic Apple quirk

mbp101
>   MacBookPro 10,1

mbp81
>   MacBookPro 8,1

mba42
>   MacBookAir 4,2

auto
>   BIOS setup (default)

5.2.35 Cirrus Logic CS4208

mba6
>   MacBook Air 6,1 and 6,2

gpio0
>   Enable GPIO 0 amp

mbp11
>   MacBookPro 11,2

macmini
>   MacMini 7,1

auto
>   BIOS setup (default)

5.2.36 VIA VT17xx/VT18xx/VT20xx

auto
>   BIOS setup (default)

5.3 HD-Audio Codec-Specific Mixer Controls

This file explains the codec-specific mixer controls.

5.3.1 Realtek codecs

Channel Mode
>   This is an enum control to change the surround-channel setup, appears only when the surround channels are available. It gives the number of channels to be used, "2ch", "4ch", "6ch", and "8ch". According to the configuration, this also controls the jack-retasking of multi-I/O jacks.

Auto-Mute Mode
>   This is an enum control to change the auto-mute behavior of the headphone and line-out jacks. If built-in speakers and headphone and/or line-out jacks are available on a machine,

this controls appears. When there are only either headphones or line-out jacks, it gives "Disabled" and "Enabled" state. When enabled, the speaker is muted automatically when a jack is plugged.

When both headphone and line-out jacks are present, it gives "Disabled", "Speaker Only" and "Line-Out+Speaker". When speaker-only is chosen, plugging into a headphone or a line-out jack mutes the speakers, but not line-outs. When line-out+speaker is selected, plugging to a headphone jack mutes both speakers and line-outs.

5.3.2 IDT/Sigmatel codecs

Analog Loopback
> This control enables/disables the analog-loopback circuit. This appears only when "loopback" is set to true in a codec hint (see HD-Audio.txt). Note that on some codecs the analog-loopback and the normal PCM playback are exclusive, i.e. when this is on, you won't hear any PCM stream.

Swap Center/LFE
> Swaps the center and LFE channel order. Normally, the left corresponds to the center and the right to the LFE. When this is ON, the left to the LFE and the right to the center.

Headphone as Line Out
> When this control is ON, treat the headphone jacks as line-out jacks. That is, the headphone won't auto-mute the other line-outs, and no HP-amp is set to the pins.

Mic Jack Mode, Line Jack Mode, etc
> These enum controls the direction and the bias of the input jack pins. Depending on the jack type, it can set as "Mic In" and "Line In", for determining the input bias, or it can be set to "Line Out" when the pin is a multi-I/O jack for surround channels.

5.3.3 VIA codecs

Smart 5.1
> An enum control to re-task the multi-I/O jacks for surround outputs. When it's ON, the corresponding input jacks (usually a line-in and a mic-in) are switched as the surround and the CLFE output jacks.

Independent HP
> When this enum control is enabled, the headphone output is routed from an individual stream (the third PCM such as hw:0,2) instead of the primary stream. In the case the headphone DAC is shared with a side or a CLFE-channel DAC, the DAC is switched to the headphone automatically.

Loopback Mixing
> An enum control to determine whether the analog-loopback route is enabled or not. When it's enabled, the analog-loopback is mixed to the front-channel. Also, the same route is used for the headphone and speaker outputs. As a side-effect, when this mode is set, the individual volume controls will be no longer available for headphones and speakers because there is only one DAC connected to a mixer widget.

Dynamic Power-Control
> This control determines whether the dynamic power-control per jack detection is enabled

or not. When enabled, the widgets power state (D0/D3) are changed dynamically depending on the jack plugging state for saving power consumptions. However, if your system doesn't provide a proper jack-detection, this won't work; in such a case, turn this control OFF.

Jack Detect

This control is provided only for VT1708 codec which gives no proper unsolicited event per jack plug. When this is on, the driver polls the jack detection so that the headphone auto-mute can work, while turning this off would reduce the power consumption.

5.3.4 Conexant codecs

Auto-Mute Mode

See Realtek codecs.

5.3.5 Analog codecs

Channel Mode

This is an enum control to change the surround-channel setup, appears only when the surround channels are available. It gives the number of channels to be used, "2ch", "4ch" and "6ch". According to the configuration, this also controls the jack-retasking of multi-I/O jacks.

Independent HP

When this enum control is enabled, the headphone output is routed from an individual stream (the third PCM such as hw:0,2) instead of the primary stream.

5.4 HD-Audio DP-MST Support

To support DP MST audio, HD Audio hdmi codec driver introduces virtual pin and dynamic pcm assignment.

Virtual pin is an extension of per_pin. The most difference of DP MST from legacy is that DP MST introduces device entry. Each pin can contain several device entries. Each device entry behaves as a pin.

As each pin may contain several device entries and each codec may contain several pins, if we use one pcm per per_pin, there will be many PCMs. The new solution is to create a few PCMs and to dynamically bind pcm to per_pin. Driver uses spec->dyn_pcm_assign flag to indicate whether to use the new solution.

5.4.1 PCM

To be added

5.4.2 Pin Initialization

Each pin may have several device entries (virtual pins). On Intel platform, the device entries number is dynamically changed. If DP MST hub is connected, it is in DP MST mode, and the device entries number is 3. Otherwise, the device entries number is 1.

To simplify the implementation, all the device entries will be initialized when bootup no matter whether it is in DP MST mode or not.

5.4.3 Connection list

DP MST reuses connection list code. The code can be reused because device entries on the same pin have the same connection list.

This means DP MST gets the device entry connection list without the device entry setting.

5.4.4 Jack

Presume:

- MST must be dyn_pcm_assign, and it is acomp (for Intel scenario);
- NON-MST may or may not be dyn_pcm_assign, it can be acomp or !acomp;

So there are the following scenarios:

a. MST (&& dyn_pcm_assign && acomp)

b. NON-MST && dyn_pcm_assign && acomp

c. NON-MST && !dyn_pcm_assign && !acomp

Below discussion will ignore MST and NON-MST difference as it doesn't impact on jack handling too much.

Driver uses struct hdmi_pcm pcm[] array in hdmi_spec and snd_jack is a member of hdmi_pcm. Each pin has one struct hdmi_pcm * pcm pointer.

For !dyn_pcm_assign, per_pin->pcm will assigned to spec->pcm[n] statically.

For dyn_pcm_assign, per_pin->pcm will assigned to spec->pcm[n] when monitor is hotplugged.

Build Jack

- dyn_pcm_assign

  Will not use hda_jack but use snd_jack in spec->pcm_rec[pcm_idx].jack directly.

- !dyn_pcm_assign

  Use hda_jack and assign spec->pcm_rec[pcm_idx].jack = jack->jack statically.

Unsolicited Event Enabling

Enable unsolicited event if !acomp.

Monitor Hotplug Event Handling

- acomp

  pin_eld_notify() -> check_presence_and_report() -> hdmi_present_sense() -> sync_eld_via_acomp().

  Use directly *snd_jack_report()* on spec->pcm_rec[pcm_idx].jack for both dyn_pcm_assign and !dyn_pcm_assign

- !acomp

  hdmi_unsol_event() -> hdmi_intrinsic_event() -> check_presence_and_report() -> hdmi_present_sense() -> hdmi_prepsent_sense_via_verbs()

  Use directly *snd_jack_report()* on spec->pcm_rec[pcm_idx].jack for dyn_pcm_assign. Use hda_jack mechanism to handle jack events.

5.4.5 Others to be added later

5.5 Realtek PC Beep Hidden Register

This file documents the "PC Beep Hidden Register", which is present in certain Realtek HDA codecs and controls a muxer and pair of passthrough mixers that can route audio between pins but aren't themselves exposed as HDA widgets. As far as I can tell, these hidden routes are designed to allow flexible PC Beep output for codecs that don't have mixer widgets in their output paths. Why it's easier to hide a mixer behind an undocumented vendor register than to just expose it as a widget, I have no idea.

5.5.1 Register Description

The register is accessed via processing coefficient 0x36 on NID 20h. Bits not identified below have no discernible effect on my machine, a Dell XPS 13 9350:

```
MSB                         LSB
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| |h|S|L|        | B |R|        | Known bits
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|0|0|1|1|  0x7   |0|0x0|1|  0x7  | Reset value
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

1Ah input select (B): 2 bits
> When zero, expose the PC Beep line (from the internal beep generator, when enabled with the Set Beep Generation verb on NID 01h, or else from the external PCBEEP pin) on the 1Ah pin node. When nonzero, expose the headphone jack (or possibly Line In on some machines) input instead. If PC Beep is selected, the 1Ah boost control has no effect.

Amplify 1Ah loopback, left (L): 1 bit
> Amplify the left channel of 1Ah before mixing it into outputs as specified by h and S bits. Does not affect the level of 1Ah exposed to other widgets.

Amplify 1Ah loopback, right (R): 1 bit
> Amplify the right channel of 1Ah before mixing it into outputs as specified by h and S bits. Does not affect the level of 1Ah exposed to other widgets.

Loopback 1Ah to 21h [active low] (h): 1 bit
> When zero, mix 1Ah (possibly with amplification, depending on L and R bits) into 21h (headphone jack on my machine). Mixed signal respects the mute setting on 21h.
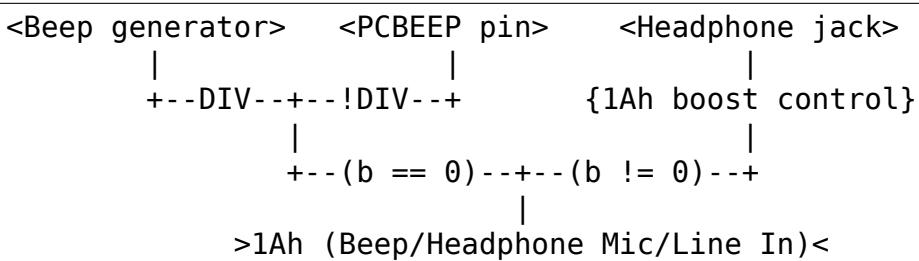
Loopback 1Ah to 14h (S): 1 bit
> When one, mix 1Ah (possibly with amplification, depending on L and R bits) into 14h (internal speaker on my machine). Mixed signal **ignores** the mute setting on 14h and is present whenever 14h is configured as an output.
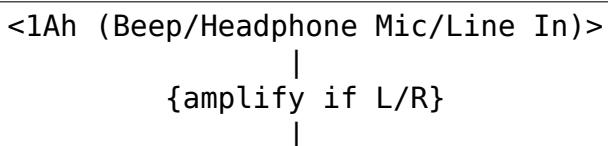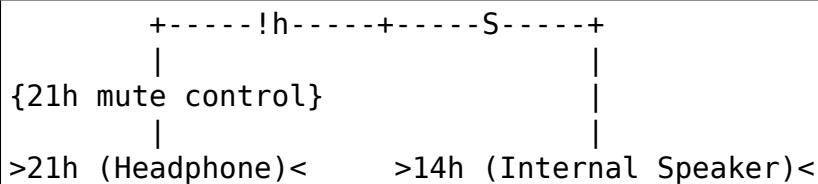
5.5.2 Path diagrams

1Ah input selection (DIV is the PC Beep divider set on NID 01h):

```
<Beep generator>    <PCBEEP pin>      <Headphone jack>
        |                |                   |
     +--DIV--+--!DIV--+       {1Ah boost control}
             |                       |
           +--(b == 0)--+--(b != 0)--+
                        |
          >1Ah (Beep/Headphone Mic/Line In)<
```

Loopback of 1Ah to 21h/14h:

```
    <1Ah (Beep/Headphone Mic/Line In)>
                   |
           {amplify if L/R}
                   |
```

```
        +-----!h-----+-----S-----+
        |                        |
{21h mute control}              |
        |                        |
>21h (Headphone)<      >14h (Internal Speaker)<
```

5.5.3 Background

All Realtek HDA codecs have a vendor-defined widget with node ID 20h which provides access to a bank of registers that control various codec functions. Registers are read and written via the standard HDA processing coefficient verbs (Set/Get Coefficient Index, Set/Get Processing Coefficient). The node is named "Realtek Vendor Registers" in public datasheets' verb listings and, apart from that, is entirely undocumented.

This particular register, exposed at coefficient 0x36 and named in commits from Realtek, is of note: unlike most registers, which seem to control detailed amplifier parameters not in scope of the HDA specification, it controls audio routing which could just as easily have been defined using standard HDA mixer and selector widgets.

Specifically, it selects between two sources for the input pin widget with Node ID (NID) 1Ah: the widget's signal can come either from an audio jack (on my laptop, a Dell XPS 13 9350, it's the headphone jack, but comments in Realtek commits indicate that it might be a Line In on some machines) or from the PC Beep line (which is itself multiplexed between the codec's internal beep generator and external PCBEEP pin, depending on if the beep generator is enabled via verbs on NID 01h). Additionally, it can mix (with optional amplification) that signal onto the 21h and/or 14h output pins.

The register's reset value is 0x3717, corresponding to PC Beep on 1Ah that is then amplified and mixed into both the headphones and the speakers. Not only does this violate the HDA specification, which says that "[a vendor defined beep input pin] connection may be maintained *only* while the Link reset (**RST#**) is asserted", it means that we cannot ignore the register if we care about the input that 1Ah would otherwise expose or if the PCBEEP trace is poorly shielded and picks up chassis noise (both of which are the case on my machine).

Unfortunately, there are lots of ways to get this register configuration wrong. Linux, it seems, has gone through most of them. For one, the register resets after S3 suspend: judging by existing code, this isn't the case for all vendor registers, and it's led to some fixes that improve behavior on cold boot but don't last after suspend. Other fixes have successfully switched the 1Ah input away from PC Beep but have failed to disable both loopback paths. On my machine, this means that the headphone input is amplified and looped back to the headphone output, which uses the exact same pins! As you might expect, this causes terrible headphone noise, the character of which is controlled by the 1Ah boost control. (If you've seen instructions online to fix XPS 13 headphone noise by changing "Headphone Mic Boost" in ALSA, now you know why.)

The information here has been obtained through black-box reverse engineering of the ALC256 codec's behavior and is not guaranteed to be correct. It likely also applies for the ALC255, ALC257, ALC235, and ALC236, since those codecs seem to be close relatives of the ALC256. (They all share one initialization function.) Additionally, other codecs like the ALC225 and ALC285 also have this register, judging by existing fixups in `patch_realtek.c`, but specific data (e.g. node IDs, bit positions, pin mappings) for those codecs may differ from what I've described here.

5.6 HDAudio multi-link extensions on Intel platforms

Copyright
© 2023 Intel Corporation

This file documents the 'multi-link structure' introduced in 2015 with the Skylake processor and recently extended in newer Intel platforms
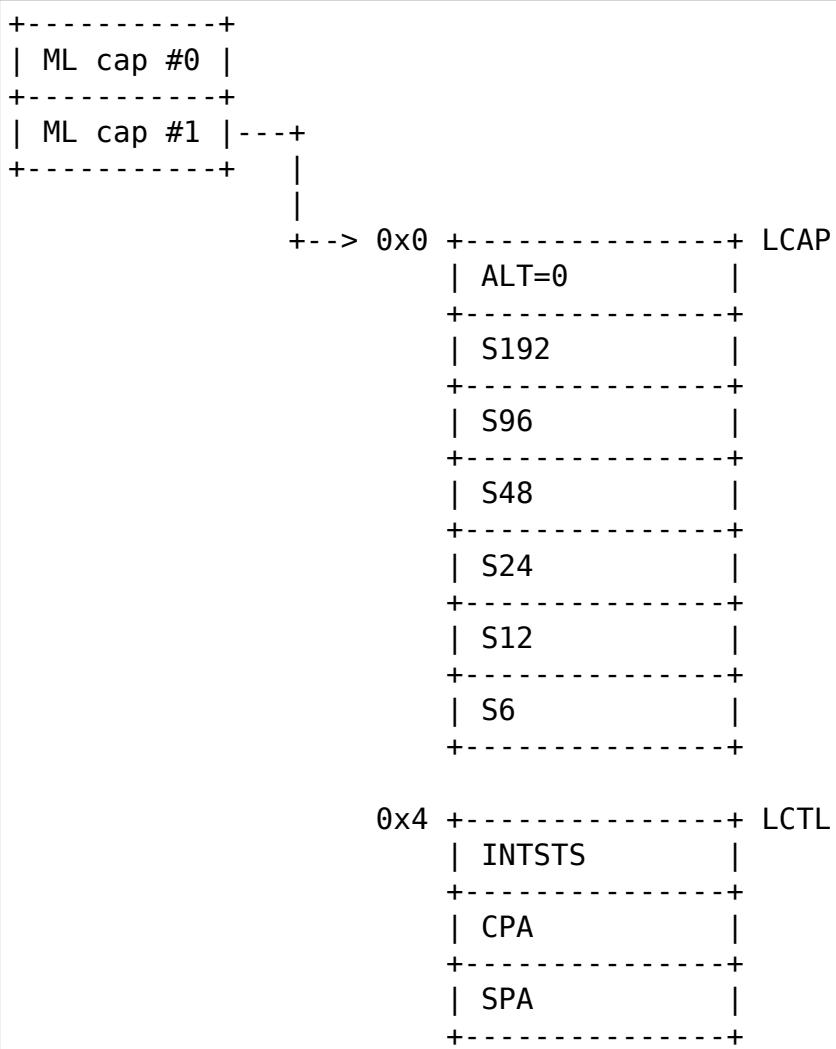
5.6.1 HDaudio existing link mapping (2015 addition in SkyLake)
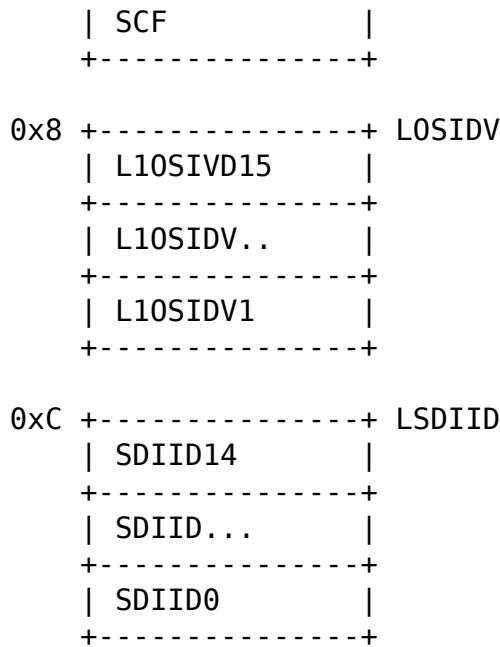
External HDAudio codecs are handled with link #0, while iDISP codec for HDMI/DisplayPort is handled with link #1.

The only change to the 2015 definitions is the declaration of the LCAP.ALT=0x0 - since the ALT bit was previously reserved, this is a backwards-compatible change.

LCTL.SPA and LCTL.CPA are automatically set when exiting reset. They are only used in existing drivers when the SCF value needs to be corrected.

Basic structure for HDaudio codecs

```
+-----------+
| ML cap #0 |
+-----------+
| ML cap #1 |---+
+-----------+   |
                |
                |
              +--> 0x0 +---------------+ LCAP
                       | ALT=0         |
                       +---------------+
                       | S192          |
                       +---------------+
                       | S96           |
                       +---------------+
                       | S48           |
                       +---------------+
                       | S24           |
                       +---------------+
                       | S12           |
                       +---------------+
                       | S6            |
                       +---------------+

                  0x4 +---------------+ LCTL
                       | INTSTS        |
                       +---------------+
                       | CPA           |
                       +---------------+
                       | SPA           |
                       +---------------+
```

```
                        | SCF           |
                        +--------------+

            0x8 +--------------+ LOSIDV
                | L10SIVD15    |
                +--------------+
                | L10SIDV..    |
                +--------------+
                | L10SIDV1     |
                +--------------+

            0xC +--------------+ LSDIID
                | SDIID14      |
                +--------------+
                | SDIID...     |
                +--------------+
                | SDIID0       |
                +--------------+
```

5.6.2 SoundWire HDaudio extended link mapping
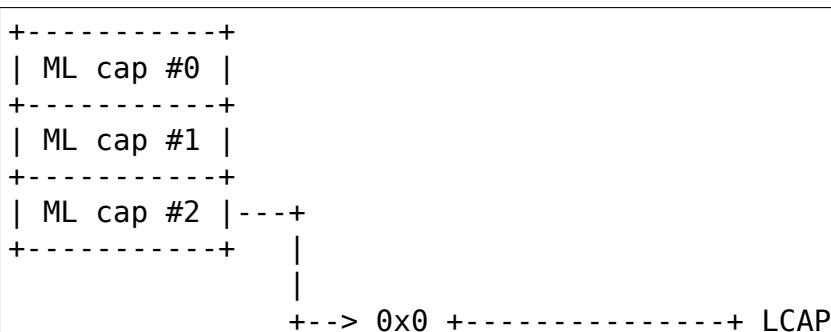
A SoundWire extended link is identified when LCAP.ALT=1 and LEPTR.ID=0.
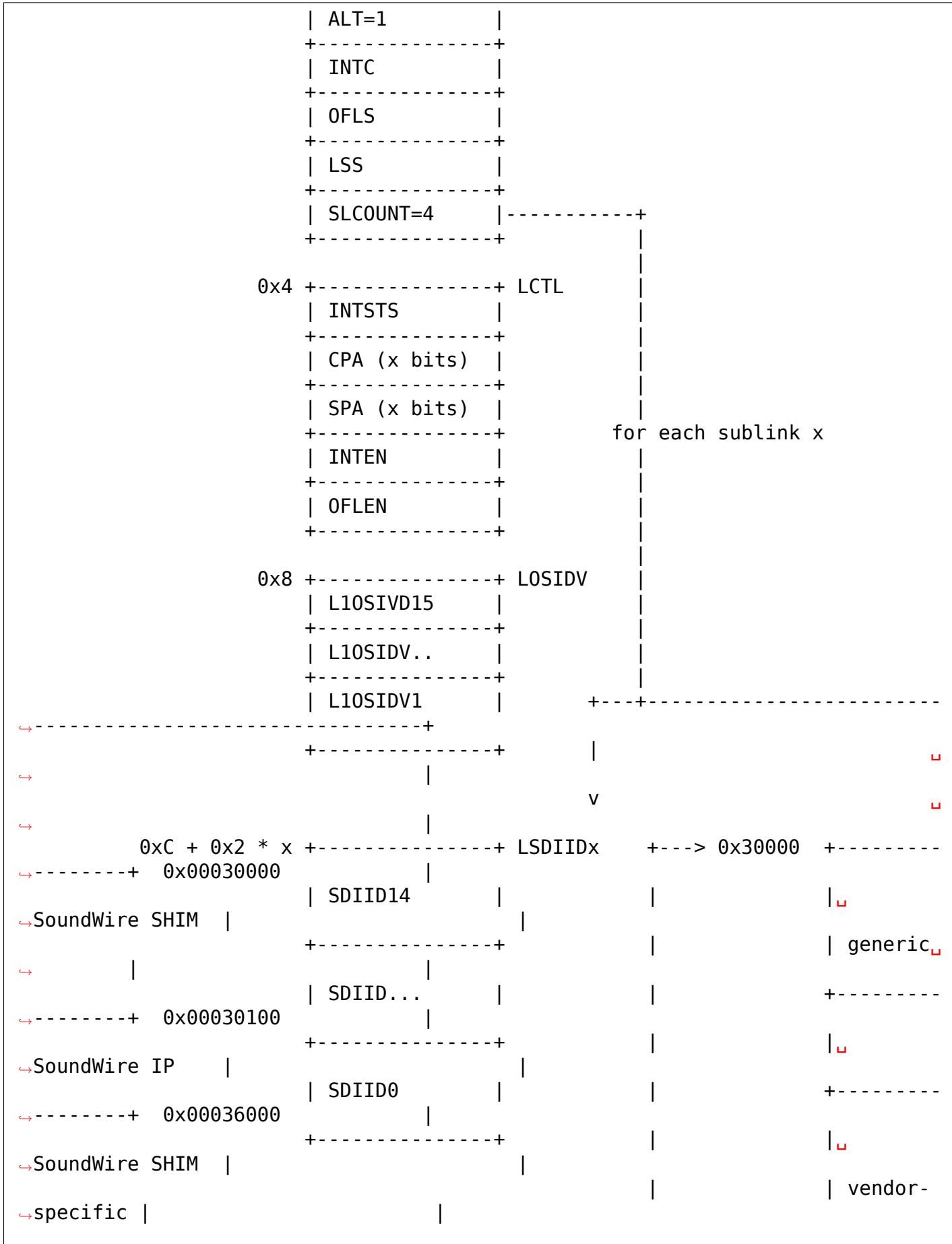
DMA control uses the existing LOSIDV register.
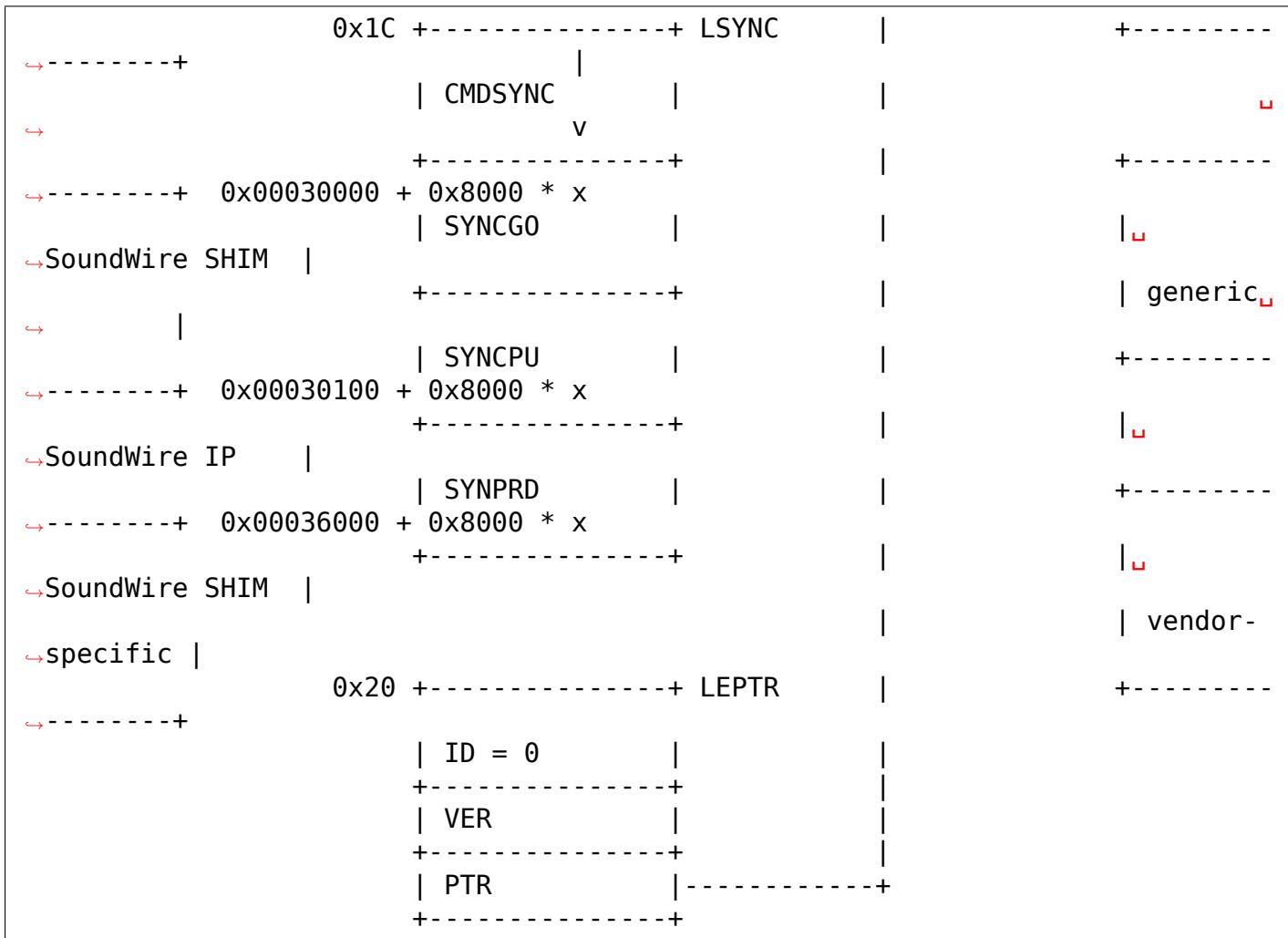
Changes include additional descriptions for enumeration that were not present in earlier generations.

- multi-link synchronization: capabilities in LCAP.LSS and control in LSYNC
- number of sublinks (manager IP) in LCAP.LSCOUNT
- power management moved from SHIM to LCTL.SPA bits
- hand-over to the DSP for access to multi-link registers, SHIM/IP with LCTL.OFLEN
- mapping of SoundWire codecs to SDI ID bits
- move of SHIM and Cadence registers to different offsets, with no change in functionality. The LEPTR.PTR value is an offset from the ML address, with a default value of 0x30000.

Extended structure for SoundWire (assuming 4 Manager IP)

```
+-----------+
| ML cap #0 |
+-----------+
| ML cap #1 |
+-----------+
| ML cap #2 |---+
+-----------+   |
                |
        +--> 0x0 +--------------+ LCAP
```

```
                              | ALT=1          |
                              +--------------+
                              | INTC          |
                              +--------------+
                              | OFLS          |
                              +--------------+
                              | LSS           |
                              +--------------+
                              | SLCOUNT=4     |-----------+
                              +--------------+           |
                                                         |
                   0x4 +--------------+ LCTL             |
                              | INTSTS        |          |
                              +--------------+           |
                              | CPA (x bits) |           |
                              +--------------+           |
                              | SPA (x bits) |           |
                              +--------------+    for each sublink x
                              | INTEN         |          |
                              +--------------+           |
                              | OFLEN         |          |
                              +--------------+           |
                                                         |
                   0x8 +--------------+ LOSIDV            |
                              | L1OSIVD15     |          |
                              +--------------+           |
                              | L1OSIDV..     |          |
                              +--------------+           |
                              | L1OSIDV1      |      +---+------------------------
→-----------------------------------+
                              +--------------+      |                            ␣
→                                     |
                                      |                v                         ␣
→                                     |
            0xC + 0x2 * x +--------------+ LSDIIDx    +---> 0x30000  +---------
→--------+  0x00030000
                              | SDIID14       |                    |␣
→SoundWire SHIM  |            |
                              +--------------+                    | generic␣
→           |                |
                              | SDIID...      |                    +---------
→--------+  0x00030100
                              +--------------+                    |␣
→SoundWire IP    |            |
                              | SDIID0        |                    +---------
→--------+  0x00036000
                              +--------------+                    |␣
→SoundWire SHIM  |            |
                                                                  | vendor-
→specific |                  |
```

```
                0x1C +---------------+ LSYNC      |          +---------
↪--------+                   |
                     | CMDSYNC       |           |                    ⌴
↪                            v
                     +---------------+           |          +---------
↪--------+  0x00030000 + 0x8000 * x
                     | SYNCGO        |           |          |⌴
↪SoundWire SHIM  |
                     +---------------+           |          | generic⌴
↪        |
                     | SYNCPU        |           |          +---------
↪--------+  0x00030100 + 0x8000 * x
                     +---------------+           |          |⌴
↪SoundWire IP    |
                     | SYNPRD        |           |          +---------
↪--------+  0x00036000 + 0x8000 * x
                     +---------------+           |          |⌴
↪SoundWire SHIM  |
                                                 |          | vendor-
↪specific |
                0x20 +---------------+ LEPTR      |          +---------
↪--------+
                     | ID = 0        |           |
                     +---------------+           |
                     | VER           |           |
                     +---------------+           |
                     | PTR           |-----------+
                     +---------------+
```

5.6.3 DMIC HDaudio extended link mapping
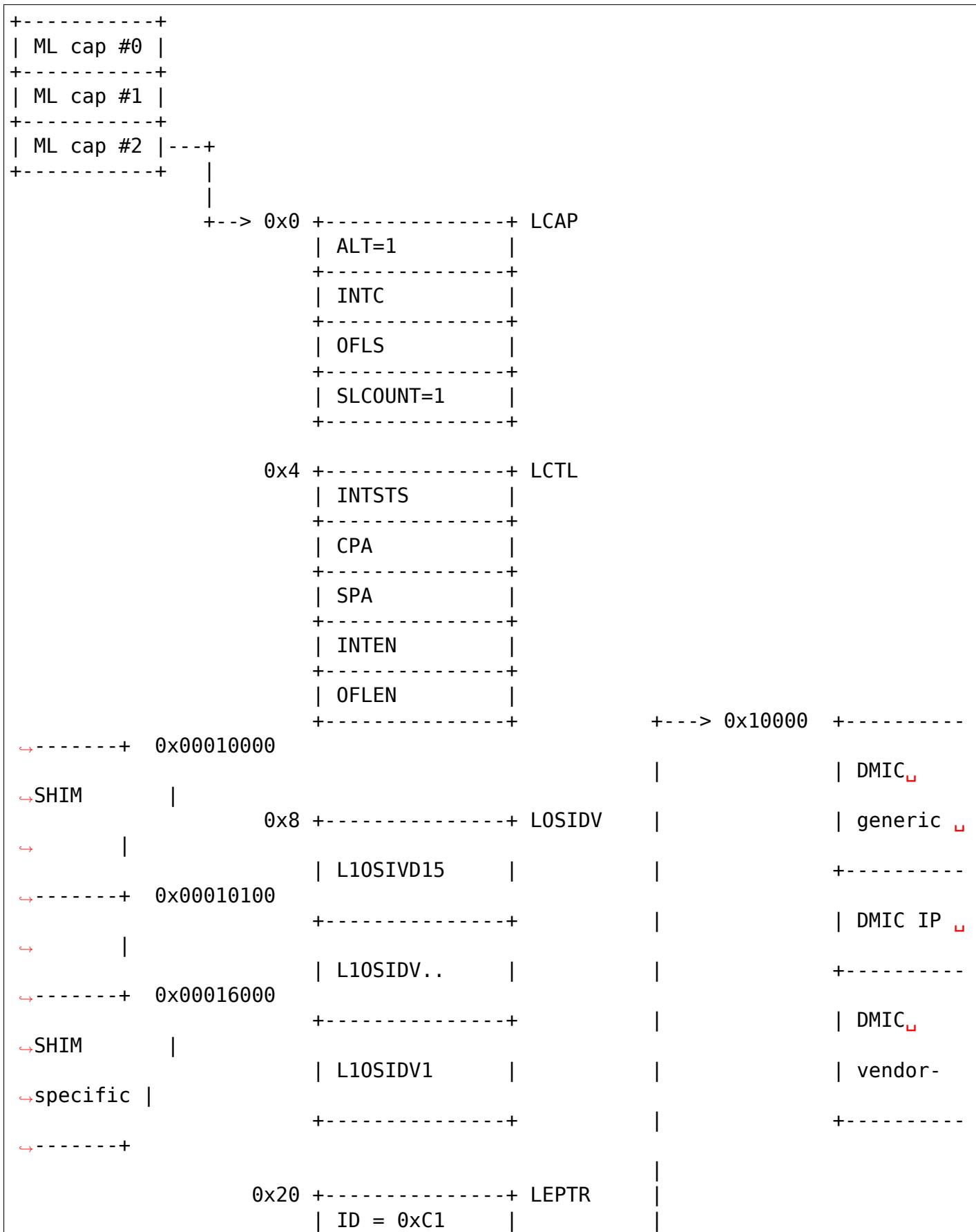
A DMIC extended link is identified when LCAP.ALT=1 and LEPTR.ID=0xC1 are set.
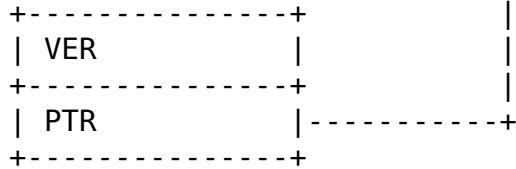
DMA control uses the existing LOSIDV register

Changes include additional descriptions for enumeration that were not present in earlier generations.

- multi-link synchronization: capabilities in LCAP.LSS and control in LSYNC

- power management with LCTL.SPA bits

- hand-over to the DSP for access to multi-link registers, SHIM/IP with LCTL.OFLEN

- move of DMIC registers to different offsets, with no change in functionality. The LEPTR.PTR value is an offset from the ML address, with a default value of 0x10000.

Extended structure for DMIC

```
+-----------+
| ML cap #0 |
+-----------+
| ML cap #1 |
+-----------+
| ML cap #2 |---+
+-----------+   |
                |
            +--> 0x0 +---------------+ LCAP
                    | ALT=1         |
                    +---------------+
                    | INTC          |
                    +---------------+
                    | OFLS          |
                    +---------------+
                    | SLCOUNT=1     |
                    +---------------+

            0x4 +---------------+ LCTL
                    | INTSTS        |
                    +---------------+
                    | CPA           |
                    +---------------+
                    | SPA           |
                    +---------------+
                    | INTEN         |
                    +---------------+
                    | OFLEN         |
                    +---------------+                    +---> 0x10000  +----------
↪-------+  0x00010000
                                                         |              | DMIC␣
↪SHIM       |
            0x8 +---------------+ LOSIDV       |              | generic ␣
↪       |
                    | L10SIVD15     |          |              +----------
↪-------+  0x00010100
                    +---------------+          |              | DMIC IP ␣
↪       |
                    | L10SIDV..     |          |              +----------
↪-------+  0x00016000
                    +---------------+          |              | DMIC␣
↪SHIM       |
                    | L10SIDV1      |          |              | vendor-
↪specific |
                    +---------------+          |              +----------
↪-------+
                                               |
            0x20 +---------------+ LEPTR        |
                    | ID = 0xC1     |          |
```

```
                    +---------------+            |
                    | VER           |            |
                    +---------------+            |
                    | PTR           |-----------+
                    +---------------+
```
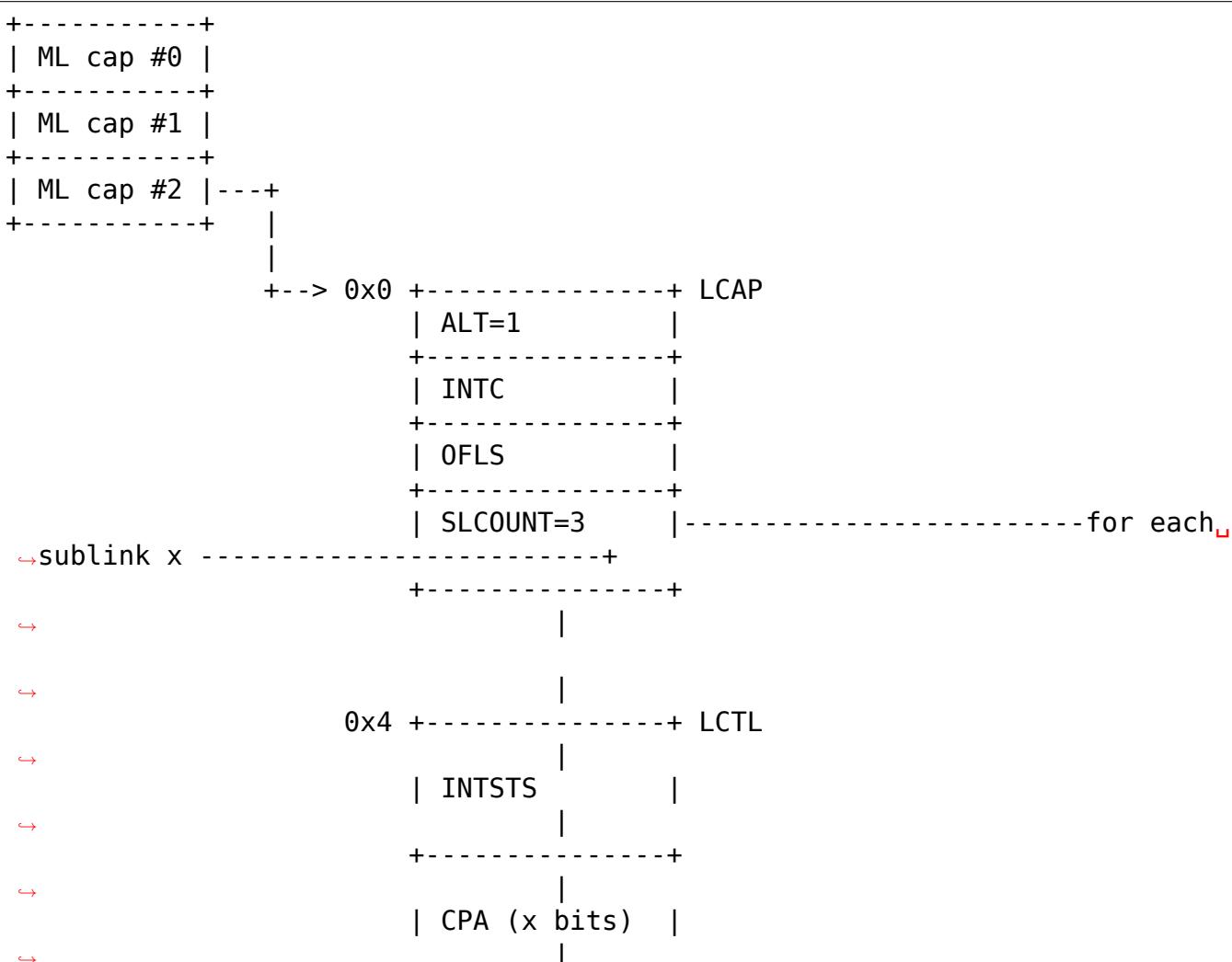
5.6.4 SSP HDaudio extended link mapping
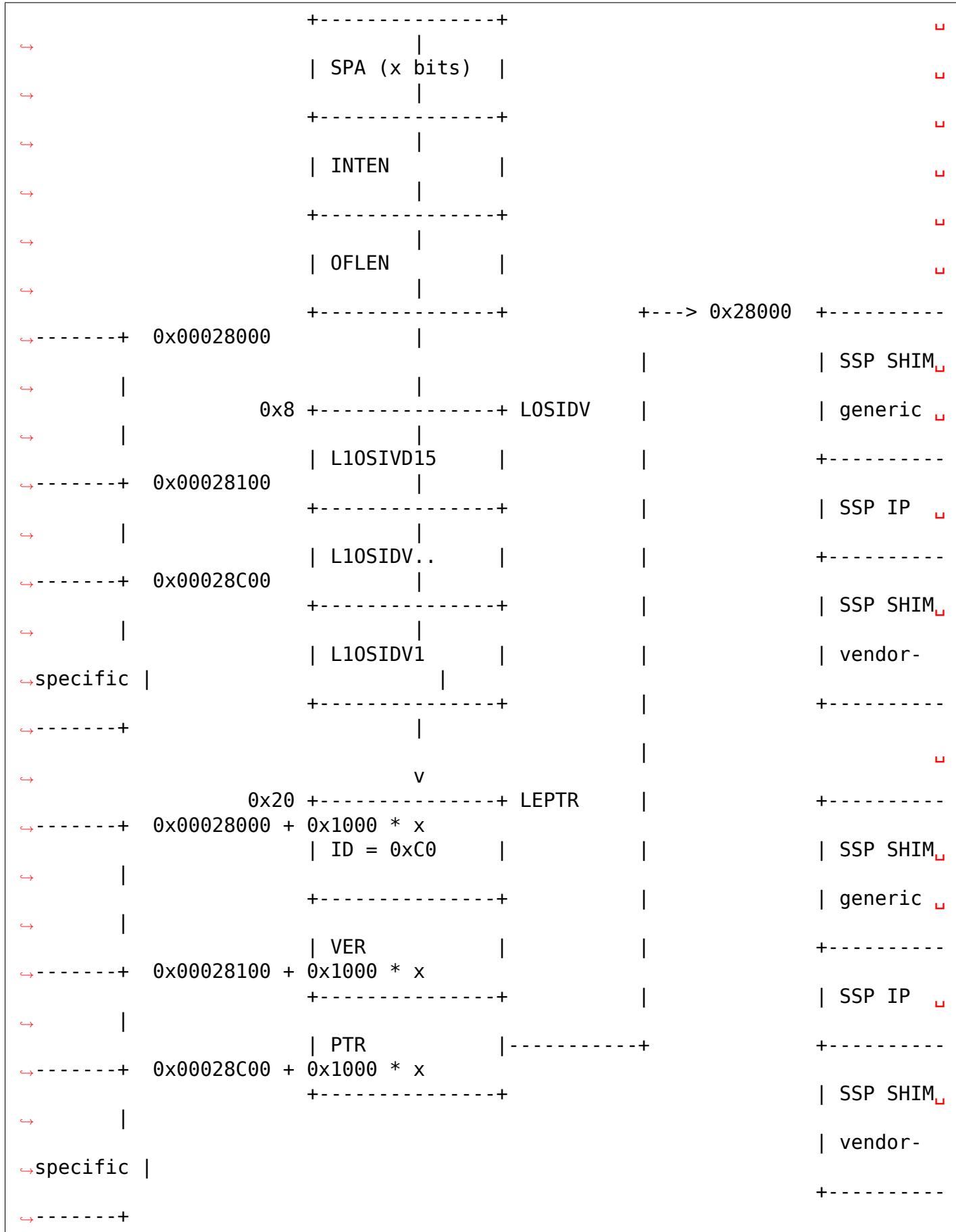
A DMIC extended link is identified when LCAP.ALT=1 and LEPTR.ID=0xC0 are set.

DMA control uses the existing LOSIDV register

Changes include additional descriptions for enumeration and control that were not present in earlier generations: - number of sublinks (SSP IP instances) in LCAP.LSCOUNT - power management moved from SHIM to LCTL.SPA bits - hand-over to the DSP for access to multi-link registers, SHIM/IP with LCTL.OFLEN - move of SHIM and SSP IP registers to different offsets, with no change in functionality. The LEPTR.PTR value is an offset from the ML address, with a default value of 0x28000.

Extended structure for SSP (assuming 3 instances of the IP)

```
+-----------+
| ML cap #0 |
+-----------+
| ML cap #1 |
+-----------+
| ML cap #2 |---+
+-----------+   |
                |
           +--> 0x0 +---------------+ LCAP
                    | ALT=1         |
                    +---------------+
                    | INTC          |
                    +---------------+
                    | OFLS          |
                    +---------------+
                    | SLCOUNT=3     |-----------------------for each␣
 ↪sublink x -----------------------+
                    +---------------+                                ␣
 ↪                          |                                       ␣
 ↪                          |                                       ␣
           0x4 +---------------+ LCTL                               ␣
 ↪                          |                                       ␣
                    | INTSTS        |                               ␣
 ↪                          |                                       ␣
                    +---------------+                               ␣
 ↪                          |                                       ␣
                    | CPA (x bits)  |                               ␣
 ↪                          |
```

```
                    +---------------+                              ␣
↪
                    | SPA (x bits)  |                              ␣
↪
                    +---------------+                              ␣
↪
                    | INTEN         |                              ␣
↪
                    +---------------+                              ␣
↪
                    | OFLEN         |                              ␣
↪
                    +---------------+              +---> 0x28000  +----------
↪-------+  0x00028000         |
↪         |                   |                    |             | SSP SHIM␣
↪         |               0x8 +---------------+ LOSIDV |          | generic ␣
↪         |                   |               |        |          +----------
↪-------+  0x00028100         | L1OSIVD15     |        |          | SSP IP  ␣
↪         |                   +---------------+        |          +----------
↪-------+  0x00028C00         | L1OSIDV..     |        |          | SSP SHIM␣
↪         |                   +---------------+        |          | vendor-
↪specific |                   | L1OSIDV1      |        |          +----------
↪-------+                     |               |        |
↪                             +---------------+        |          +----------
↪                                     |                |              ␣
↪                                     v                |          +----------
↪                        0x20 +---------------+ LEPTR  |
↪-------+  0x00028000 + 0x1000 * x
                             | ID = 0xC0     |        |          | SSP SHIM␣
↪         |                   +---------------+        |          | generic ␣
↪         |                   | VER           |        |          +----------
↪-------+  0x00028100 + 0x1000 * x
                             +---------------+        |          | SSP IP  ␣
↪         |                   | PTR           |-----------+      +----------
↪-------+  0x00028C00 + 0x1000 * x
                             +---------------+                   | SSP SHIM␣
↪         |                                                      | vendor-
↪specific |
↪                                                                +----------
↪-------+
```

CARD-SPECIFIC INFORMATION

6.1 Analog Joystick Support on ALSA Drivers

Oct. 14, 2003

Takashi Iwai <tiwai@suse.de>

6.1.1 General

First of all, you need to enable GAMEPORT support on Linux kernel for using a joystick with the ALSA driver. For the details of gameport support, refer to Documentation/input/joydev/joystick.rst.

The joystick support of ALSA drivers is different between ISA and PCI cards. In the case of ISA (PnP) cards, it's usually handled by the independent module (ns558). Meanwhile, the ALSA PCI drivers have the built-in gameport support. Hence, when the ALSA PCI driver is built in the kernel, CONFIG_GAMEPORT must be 'y', too. Otherwise, the gameport support on that card will be (silently) disabled.

Some adapter modules probe the physical connection of the device at the load time. It'd be safer to plug in the joystick device before loading the module.

6.1.2 PCI Cards

For PCI cards, the joystick is enabled when the appropriate module option is specified. Some drivers don't need options, and the joystick support is always enabled. In the former ALSA version, there was a dynamic control API for the joystick activation. It was changed, however, to the static module options because of the system stability and the resource management.

The following PCI drivers support the joystick natively.

Driver	Module Option	Available Values
als4000	joystick_port	0 = disable (default), 1 = auto-detect, manual: any address (e.g. 0x200)
au88x0	N/A	N/A
azf3328	joystick	0 = disable, 1 = enable, -1 = auto (default)
ens1370	joystick	0 = disable (default), 1 = enable
ens1371	joystick_port	0 = disable (default), 1 = auto-detect, manual: 0x200, 0x208, 0x210, 0x218
cmipci	joystick_port	0 = disable (default), 1 = auto-detect, manual: any address (e.g. 0x200)
cs4281	N/A	N/A
cs46xx	N/A	N/A
es1938	N/A	N/A
es1968	joystick	0 = disable (default), 1 = enable
sonicvibes	N/A	N/A
trident	N/A	N/A
via82xx[1]	joystick	0 = disable (default), 1 = enable
ymfpci	joystick_port	0 = disable (default), 1 = auto-detect, manual: 0x201, 0x202, 0x204, 0x205[2]

The following drivers don't support gameport natively, but there are additional modules. Load the corresponding module to add the gameport support.

Driver	Additional Module
emu10k1	emu10k1-gp
fm801	fm801-gp

Note: the "pcigame" and "cs461x" modules are for the OSS drivers only. These ALSA drivers (cs46xx, trident and au88x0) have the built-in gameport support.

As mentioned above, ALSA PCI drivers have the built-in gameport support, so you don't have to load ns558 module. Just load "joydev" and the appropriate adapter module (e.g. "analog").

---

[1] VIA686A/B only

[2] With YMF744/754 chips, the port address can be chosen arbitrarily

6.1.3 ISA Cards

ALSA ISA drivers don't have the built-in gameport support. Instead, you need to load "ns558" module in addition to "joydev" and the adapter module (e.g. "analog").

6.2 Brief Notes on C-Media 8338/8738/8768/8770 Driver

Takashi Iwai <tiwai@suse.de>

6.2.1 Front/Rear Multi-channel Playback

CM8x38 chip can use ADC as the second DAC so that two different stereo channels can be used for front/rear playbacks. Since there are two DACs, both streams are handled independently unlike the 4/6ch multi- channel playbacks in the section below.

As default, ALSA driver assigns the first PCM device (i.e. hw:0,0 for card#0) for front and 4/6ch playbacks, while the second PCM device (hw:0,1) is assigned to the second DAC for rear playback.

There are slight differences between the two DACs:

- The first DAC supports U8 and S16LE formats, while the second DAC supports only S16LE.

- The second DAC supports only two channel stereo.

Please note that the CM8x38 DAC doesn't support continuous playback rate but only fixed rates: 5512, 8000, 11025, 16000, 22050, 32000, 44100 and 48000 Hz.

The rear output can be heard only when "Four Channel Mode" switch is disabled. Otherwise no signal will be routed to the rear speakers. As default it's turned on.

> **Warning:** When "Four Channel Mode" switch is off, the output from rear speakers will be FULL VOLUME regardless of Master and PCM volumes[1]. This might damage your audio equipment. Please disconnect speakers before your turn off this switch.

If your card has an extra output jack for the rear output, the rear playback should be routed there as default. If not, there is a control switch in the driver "Line-In As Rear", which you can change via alsamixer or somewhat else. When this switch is on, line-in jack is used as rear output.

There are two more controls regarding to the rear output. The "Exchange DAC" switch is used to exchange front and rear playback routes, i.e. the 2nd DAC is output from front output.

---

[1] Well.. I once got the output with correct volume (i.e. same with the front one) and was so excited. It was even with "Four Channel" bit on and "double DAC" mode. Actually I could hear separate 4 channels from front and rear speakers! But.. after reboot, all was gone. It's a very pity that I didn't save the register dump at that time.. Maybe there is an unknown register to achieve this...

6.2.2 4/6 Multi-Channel Playback

The recent CM8738 chips support for the 4/6 multi-channel playback function. This is useful especially for AC3 decoding.

When the multi-channel is supported, the driver name has a suffix "-MC" such like "CMI8738-MC6". You can check this name from /proc/asound/cards.

When the 4/6-ch output is enabled, the second DAC accepts up to 6 (or 4) channels. While the dual DAC supports two different rates or formats, the 4/6-ch playback supports only the same condition for all channels. Since the multi-channel playback mode uses both DACs, you cannot operate with full-duplex.

The 4.0 and 5.1 modes are defined as the pcm "surround40" and "surround51" in alsa-lib. For example, you can play a WAV file with 6 channels like

```
% aplay -Dsurround51 sixchannels.wav
```

For programming the 4/6 channel playback, you need to specify the PCM channels as you like and set the format S16LE. For example, for playback with 4 channels,

```
snd_pcm_hw_params_set_access(pcm, hw, SND_PCM_ACCESS_RW_INTERLEAVED);
    // or mmap if you like
snd_pcm_hw_params_set_format(pcm, hw, SND_PCM_FORMAT_S16_LE);
snd_pcm_hw_params_set_channels(pcm, hw, 4);
```

and use the interleaved 4 channel data.

There are some control switches affecting to the speaker connections:

Line-In Mode
an enum control to change the behavior of line-in jack. Either "Line-In", "Rear Output" or "Bass Output" can be selected. The last item is available only with model 039 or newer. When "Rear Output" is chosen, the surround channels 3 and 4 are output to line-in jack.

Mic-In Mode
an enum control to change the behavior of mic-in jack. Either "Mic-In" or "Center/LFE Output" can be selected. When "Center/LFE Output" is chosen, the center and bass channels (channels 5 and 6) are output to mic-in jack.

6.2.3 Digital I/O

The CM8x38 provides the excellent SPDIF capability with very cheap price (yes, that's the reason I bought the card :)

The SPDIF playback and capture are done via the third PCM device (hw:0,2). Usually this is assigned to the PCM device "spdif". The available rates are 44100 and 48000 Hz. For playback with aplay, you can run like below:

```
% aplay -Dhw:0,2 foo.wav
```

or

```
% aplay -Dspdif foo.wav
```

24bit format is also supported experimentally.

The playback and capture over SPDIF use normal DAC and ADC, respectively, so you cannot playback both analog and digital streams simultaneously.

To enable SPDIF output, you need to turn on "IEC958 Output Switch" control via mixer or alsactl ("IEC958" is the official name of so-called S/PDIF). Then you'll see the red light on from the card so you know that's working obviously :) The SPDIF input is always enabled, so you can hear SPDIF input data from line-out with "IEC958 In Monitor" switch at any time (see below).

You can play via SPDIF even with the first device (hw:0,0), but SPDIF is enabled only when the proper format (S16LE), sample rate (441100 or 48000) and channels (2) are used. Otherwise it's turned off. (Also don't forget to turn on "IEC958 Output Switch", too.)

Additionally there are relevant control switches:

IEC958 Mix Analog
Mix analog PCM playback and FM-OPL/3 streams and output through SPDIF. This switch appears only on old chip models (CM8738 033 and 037).

Note: without this control you can output PCM to SPDIF. This is "mixing" of streams, so e.g. it's not for AC3 output (see the next section).

IEC958 In Select
Select SPDIF input, the internal CD-in (false) and the external input (true).

IEC958 Loop
SPDIF input data is loop back into SPDIF output (aka bypass)

IEC958 Copyright
Set the copyright bit.

IEC958 5V
Select 0.5V (coax) or 5V (optical) interface. On some cards this doesn't work and you need to change the configuration with hardware dip-switch.

IEC958 In Monitor
SPDIF input is routed to DAC.

IEC958 In Phase Inverse
Set SPDIF input format as inverse. [FIXME: this doesn't work on all chips..]

IEC958 In Valid
Set input validity flag detection.

Note: When "PCM Playback Switch" is on, you'll hear the digital output stream through analog line-out.

6.2.4 The AC3 (RAW DIGITAL) OUTPUT

The driver supports raw digital (typically AC3) i/o over SPDIF. This can be toggled via IEC958 playback control, but usually you need to access it via alsa-lib. See alsa-lib documents for more details.

On the raw digital mode, the "PCM Playback Switch" is automatically turned off so that non-audio data is heard from the analog line-out. Similarly the following switches are off: "IEC958 Mix Analog" and "IEC958 Loop". The switches are resumed after closing the SPDIF PCM device automatically to the previous state.

On the model 033, AC3 is implemented by the software conversion in the alsa-lib. If you need to bypass the software conversion of IEC958 subframes, pass the "soft_ac3=0" module option. This doesn't matter on the newer models.

6.2.5 ANALOG MIXER INTERFACE

The mixer interface on CM8x38 is similar to SB16. There are Master, PCM, Synth, CD, Line, Mic and PC Speaker playback volumes. Synth, CD, Line and Mic have playback and capture switches, too, as well as SB16.

In addition to the standard SB mixer, CM8x38 provides more functions. - PCM playback switch - PCM capture switch (to capture the data sent to DAC) - Mic Boost switch - Mic capture volume - Aux playback volume/switch and capture switch - 3D control switch

6.2.6 MIDI CONTROLLER

With CMI8338 chips, the MPU401-UART interface is disabled as default. You need to set the module option "mpu_port" to a valid I/O port address to enable MIDI support. Valid I/O ports are 0x300, 0x310, 0x320 and 0x330. Choose a value that doesn't conflict with other cards.

With CMI8738 and newer chips, the MIDI interface is enabled by default and the driver automatically chooses a port address.

There is *no* hardware wavetable function on this chip (except for OPL3 synth below). What's said as MIDI synth on Windows is a software synthesizer emulation. On Linux use TiMidity or other softsynth program for playing MIDI music.

6.2.7 FM OPL/3 Synth

The FM OPL/3 is also enabled as default only for the first card. Set "fm_port" module option for more cards.

The output quality of FM OPL/3 is, however, very weird. I don't know why..

CMI8768 and newer chips do not have the FM synth.

6.2.8 Joystick and Modem

The legacy joystick is supported. To enable the joystick support, pass joystick_port=1 module option. The value 1 means the auto-detection. If the auto-detection fails, try to pass the exact I/O address.

The modem is enabled dynamically via a card control switch "Modem".

6.2.9 Debugging Information

The registers are shown in /proc/asound/cardX/cmipci. If you have any problem (especially unexpected behavior of mixer), please attach the output of this proc file together with the bug report.

6.3 Sound Blaster Live mixer / default DSP code

The EMU10K1 chips have a DSP part which can be programmed to support various ways of sample processing, which is described here. (This article does not deal with the overall functionality of the EMU10K1 chips. See the manuals section for further details.)

The ALSA driver programs this portion of chip by default code (can be altered later) which offers the following functionality:

6.3.1 IEC958 (S/PDIF) raw PCM

This PCM device (it's the 3rd PCM device (index 2!) and first subdevice (index 0) for a given card) allows to forward 48kHz, stereo, 16-bit little endian streams without any modifications to the digital output (coaxial or optical). The universal interface allows the creation of up to 8 raw PCM devices operating at 48kHz, 16-bit little endian. It would be easy to add support for multichannel devices to the current code, but the conversion routines exist only for stereo (2-channel streams) at the time.

Look to tram_poke routines in lowlevel/emu10k1/emufx.c for more details.

6.3.2 Digital mixer controls

These controls are built using the DSP instructions. They offer extended functionality. Only the default built-in code in the ALSA driver is described here. Note that the controls work as attenuators: the maximum value is the neutral position leaving the signal unchanged. Note that if the same destination is mentioned in multiple controls, the signal is accumulated and can be clipped (set to maximal or minimal value without checking for overflow).

Explanation of used abbreviations:

DAC
digital to analog converter

ADC
analog to digital converter

I2S
one-way three wire serial bus for digital sound by Philips Semiconductors (this standard is used for connecting standalone D/A and A/D converters)

LFE
low frequency effects (used as subwoofer signal)

AC97
a chip containing an analog mixer, D/A and A/D converters

IEC958

   S/PDIF

FX-bus

   the EMU10K1 chip has an effect bus containing 16 accumulators. Each of the synthesizer voices can feed its output to these accumulators and the DSP microcontroller can operate with the resulting sum.

name='Wave Playback Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM samples. The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

name='Wave Surround Playback Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM samples. The result samples are forwarded to the rear I2S DACs. These DACs operates separately (they are not inside the AC97 codec).

name='Wave Center Playback Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM samples. The result is mixed to mono signal (single channel) and forwarded to the ??rear?? right DAC PCM slot of the AC97 codec.

name='Wave LFE Playback Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM. The result is mixed to mono signal (single channel) and forwarded to the ??rear?? left DAC PCM slot of the AC97 codec.

name='Wave Capture Volume',index=0, name='Wave Capture Switch',index=0

These controls are used to attenuate samples from left and right PCM FX-bus accumulator. ALSA uses accumulators 0 and 1 for left and right PCM. The result is forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

name='Synth Playback Volume',index=0

This control is used to attenuate samples from left and right MIDI FX-bus accumulators. ALSA uses accumulators 4 and 5 for left and right MIDI samples. The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

`name='Synth Capture Volume',index=0, name='Synth Capture Switch',index=0`

These controls are used to attenuate samples from left and right MIDI FX-bus accumulator. ALSA uses accumulators 4 and 5 for left and right MIDI samples. The result is forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

`name='Surround Playback Volume',index=0`

This control is used to attenuate samples from left and right rear PCM FX-bus accumulators. ALSA uses accumulators 2 and 3 for left and right rear PCM samples. The result samples are forwarded to the rear I2S DACs. These DACs operate separately (they are not inside the AC97 codec).

`name='Surround Capture Volume',index=0, name='Surround Capture Switch',index=0`

These controls are used to attenuate samples from left and right rear PCM FX-bus accumulators. ALSA uses accumulators 2 and 3 for left and right rear PCM samples. The result is forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

`name='Center Playback Volume',index=0`

This control is used to attenuate sample for center PCM FX-bus accumulator. ALSA uses accumulator 6 for center PCM sample. The result sample is forwarded to the ??rear?? right DAC PCM slot of the AC97 codec.

`name='LFE Playback Volume',index=0`

This control is used to attenuate sample for center PCM FX-bus accumulator. ALSA uses accumulator 6 for center PCM sample. The result sample is forwarded to the ??rear?? left DAC PCM slot of the AC97 codec.

`name='AC97 Playback Volume',index=0`

This control is used to attenuate samples from left and right front ADC PCM slots of the AC97 codec. The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

Note: This control should be zero for the standard operations, otherwise a digital loopback is activated.

---

name='AC97 Capture Volume',index=0

This control is used to attenuate samples from left and right front ADC PCM slots of the AC97 codec. The result is forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

Note: This control should be 100 (maximal value), otherwise no analog inputs of the AC97 codec can be captured (recorded).

name='IEC958 TTL Playback Volume',index=0

This control is used to attenuate samples from left and right IEC958 TTL digital inputs (usually used by a CDROM drive). The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

name='IEC958 TTL Capture Volume',index=0

This control is used to attenuate samples from left and right IEC958 TTL digital inputs (usually used by a CDROM drive). The result samples are forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

name='Zoom Video Playback Volume',index=0

This control is used to attenuate samples from left and right zoom video digital inputs (usually used by a CDROM drive). The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

name='Zoom Video Capture Volume',index=0

This control is used to attenuate samples from left and right zoom video digital inputs (usually used by a CDROM drive). The result samples are forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

name='IEC958 LiveDrive Playback Volume',index=0

This control is used to attenuate samples from left and right IEC958 optical digital input. The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

name='IEC958 LiveDrive Capture Volume',index=0

This control is used to attenuate samples from left and right IEC958 optical digital inputs. The result samples are forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

name='IEC958 Coaxial Playback Volume',index=0

This control is used to attenuate samples from left and right IEC958 coaxial digital inputs. The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

name='IEC958 Coaxial Capture Volume',index=0

This control is used to attenuate samples from left and right IEC958 coaxial digital inputs. The result samples are forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

name='Line LiveDrive Playback Volume',index=0,   name='Line LiveDrive Playback Volume',index=1

This control is used to attenuate samples from left and right I2S ADC inputs (on the LiveDrive). The result samples are forwarded to the front DAC PCM slots of the AC97 codec.

name='Line LiveDrive Capture Volume',index=1,   name='Line LiveDrive Capture Volume',index=1

This control is used to attenuate samples from left and right I2S ADC inputs (on the LiveDrive). The result samples are forwarded to the ADC capture FIFO (thus to the standard capture PCM device).

name='Tone Control - Switch',index=0

This control turns the tone control on or off. The samples for front, rear and center / LFE outputs are affected.

name='Tone Control - Bass',index=0

This control sets the bass intensity. There is no neutral value!! When the tone control code is activated, the samples are always modified. The closest value to pure signal is 20.

name='Tone Control - Treble',index=0

This control sets the treble intensity. There is no neutral value!! When the tone control code is activated, the samples are always modified. The closest value to pure signal is 20.

name='IEC958 Optical Raw Playback Switch',index=0

If this switch is on, then the samples for the IEC958 (S/PDIF) digital output are taken only from the raw FX8010 PCM, otherwise standard front PCM samples are taken.

name='Headphone Playback Volume',index=1

This control attenuates the samples for the headphone output.

name='Headphone Center Playback Switch',index=1

If this switch is on, then the sample for the center PCM is put to the left headphone output (useful for SB Live cards without separate center/LFE output).

name='Headphone LFE Playback Switch',index=1

If this switch is on, then the sample for the center PCM is put to the right headphone output (useful for SB Live cards without separate center/LFE output).

6.3.3 PCM stream related controls

name='EMU10K1 PCM Volume',index 0-31

Channel volume attenuation in range 0-0x1fffd. The middle value (no attenuation) is default. The channel mapping for three values is as follows:

- 0 - mono, default 0xffff (no attenuation)
- 1 - left, default 0xffff (no attenuation)
- 2 - right, default 0xffff (no attenuation)

name='EMU10K1 PCM Send Routing',index 0-31

This control specifies the destination - FX-bus accumulators. There are twelve values with this mapping:

- 0 - mono, A destination (FX-bus 0-15), default 0
- 1 - mono, B destination (FX-bus 0-15), default 1
- 2 - mono, C destination (FX-bus 0-15), default 2
- 3 - mono, D destination (FX-bus 0-15), default 3
- 4 - left, A destination (FX-bus 0-15), default 0

- 5 - left, B destination (FX-bus 0-15), default 1
- 6 - left, C destination (FX-bus 0-15), default 2
- 7 - left, D destination (FX-bus 0-15), default 3
- 8 - right, A destination (FX-bus 0-15), default 0
- 9 - right, B destination (FX-bus 0-15), default 1
- 10 - right, C destination (FX-bus 0-15), default 2
- 11 - right, D destination (FX-bus 0-15), default 3

Don't forget that it's illegal to assign a channel to the same FX-bus accumulator more than once (it means 0=0 && 1=0 is an invalid combination).

name='EMU10K1 PCM Send Volume',index 0-31

It specifies the attenuation (amount) for given destination in range 0-255. The channel mapping is following:

- 0 - mono, A destination attn, default 255 (no attenuation)
- 1 - mono, B destination attn, default 255 (no attenuation)
- 2 - mono, C destination attn, default 0 (mute)
- 3 - mono, D destination attn, default 0 (mute)
- 4 - left, A destination attn, default 255 (no attenuation)
- 5 - left, B destination attn, default 0 (mute)
- 6 - left, C destination attn, default 0 (mute)
- 7 - left, D destination attn, default 0 (mute)
- 8 - right, A destination attn, default 0 (mute)
- 9 - right, B destination attn, default 255 (no attenuation)
- 10 - right, C destination attn, default 0 (mute)
- 11 - right, D destination attn, default 0 (mute)

6.3.4 MANUALS/PATENTS

ftp://opensource.creative.com/pub/doc

Note that the site is defunct, but the documents are available from various other locations.

LM4545.pdf
   AC97 Codec

m2049.pdf
   The EMU10K1 Digital Audio Processor

hog63.ps
   FX8010 - A DSP Chip Architecture for Audio Effects

WIPO Patents

WO 9901813 (A1)
> Audio Effects Processor with multiple asynchronous streams (Jan. 14, 1999)

WO 9901814 (A1)
> Processor with Instruction Set for Audio Effects (Jan. 14, 1999)

WO 9901953 (A1)
> Audio Effects Processor having Decoupled Instruction Execution and Audio Data Sequencing (Jan. 14, 1999)

US Patents (https://www.uspto.gov/)

US 5925841
> Digital Sampling Instrument employing cache memory (Jul. 20, 1999)

US 5928342
> Audio Effects Processor integrated on a single chip with a multiport memory onto which multiple asynchronous digital sound samples can be concurrently loaded (Jul. 27, 1999)

US 5930158
> Processor with Instruction Set for Audio Effects (Jul. 27, 1999)

US 6032235
> Memory initialization circuit (Tram) (Feb. 29, 2000)

US 6138207
> Interpolation looping of audio samples in cache connected to system bus with prioritization and modification of bus transfers in accordance with loop ends and minimum block sizes (Oct. 24, 2000)

US 6151670
> Method for conserving memory storage using a pool of short term memory registers (Nov. 21, 2000)

US 6195715
> Interrupt control for multiple programs communicating with a common interrupt by associating programs to GP registers, defining interrupt register, polling GP registers, and invoking callback routine associated with defined interrupt register (Feb. 27, 2001)

6.4 Sound Blaster Audigy mixer / default DSP code

This is based on sb-live-mixer.rst.

The EMU10K2 chips have a DSP part which can be programmed to support various ways of sample processing, which is described here. (This article does not deal with the overall functionality of the EMU10K2 chips. See the manuals section for further details.)

The ALSA driver programs this portion of chip by default code (can be altered later) which offers the following functionality:

6.4.1 Digital mixer controls

These controls are built using the DSP instructions. They offer extended functionality. Only the default built-in code in the ALSA driver is described here. Note that the controls work as attenuators: the maximum value is the neutral position leaving the signal unchanged. Note that if the same destination is mentioned in multiple controls, the signal is accumulated and can be clipped (set to maximal or minimal value without checking for overflow).

Explanation of used abbreviations:

DAC
    digital to analog converter

ADC
    analog to digital converter

I2S
    one-way three wire serial bus for digital sound by Philips Semiconductors (this standard is used for connecting standalone D/A and A/D converters)

LFE
    low frequency effects (used as subwoofer signal)

AC97
    a chip containing an analog mixer, D/A and A/D converters

IEC958
    S/PDIF

FX-bus
    the EMU10K2 chip has an effect bus containing 64 accumulators. Each of the synthesizer voices can feed its output to these accumulators and the DSP microcontroller can operate with the resulting sum.

name='PCM Front Playback Volume',index=0

This control is used to attenuate samples from left and right front PCM FX-bus accumulators. ALSA uses accumulators 8 and 9 for left and right front PCM samples for 5.1 playback. The result samples are forwarded to the front speakers.

name='PCM Surround Playback Volume',index=0

This control is used to attenuate samples from left and right surround PCM FX-bus accumulators. ALSA uses accumulators 2 and 3 for left and right surround PCM samples for 5.1 playback. The result samples are forwarded to the surround (rear) speakers.

name='PCM Side Playback Volume',index=0

This control is used to attenuate samples from left and right side PCM FX-bus accumulators. ALSA uses accumulators 14 and 15 for left and right side PCM samples for 7.1 playback. The result samples are forwarded to the side speakers.

name='PCM Center Playback Volume',index=0

This control is used to attenuate samples from center PCM FX-bus accumulator. ALSA uses accumulator 6 for center PCM samples for 5.1 playback. The result samples are forwarded to the center speaker.

name='PCM LFE Playback Volume',index=0

This control is used to attenuate sample for LFE PCM FX-bus accumulator. ALSA uses accumulator 7 for LFE PCM samples for 5.1 playback. The result samples are forwarded to the subwoofer.

name='PCM Playback Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM samples for stereo playback. The result samples are forwarded to the front speakers.

name='PCM Capture Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM samples for stereo playback. The result is forwarded to the standard capture PCM device.

name='Music Playback Volume',index=0

This control is used to attenuate samples from left and right MIDI FX-bus accumulators. ALSA uses accumulators 4 and 5 for left and right MIDI samples. The result samples are forwarded to the virtual stereo mixer.

name='Music Capture Volume',index=0

These controls are used to attenuate samples from left and right MIDI FX-bus accumulator. ALSA uses accumulators 4 and 5 for left and right MIDI samples. The result is forwarded to the standard capture PCM device.

name='Mic Playback Volume',index=0

This control is used to attenuate samples from left and right Mic input of the AC97 codec. The result samples are forwarded to the virtual stereo mixer.

name='Mic Capture Volume',index=0

This control is used to attenuate samples from left and right Mic input of the AC97 codec. The result is forwarded to the standard capture PCM device.

The original samples are also forwarded to the Mic capture PCM device (device 1; 16bit/8KHz mono) without volume control.

name='Audigy CD Playback Volume',index=0

This control is used to attenuate samples from left and right IEC958 TTL digital inputs (usually used by a CDROM drive). The result samples are forwarded to the virtual stereo mixer.

name='Audigy CD Capture Volume',index=0

This control is used to attenuate samples from left and right IEC958 TTL digital inputs (usually used by a CDROM drive). The result is forwarded to the standard capture PCM device.

name='IEC958 Optical Playback Volume',index=0

This control is used to attenuate samples from left and right IEC958 optical digital input. The result samples are forwarded to the virtual stereo mixer.

name='IEC958 Optical Capture Volume',index=0

This control is used to attenuate samples from left and right IEC958 optical digital inputs. The result is forwarded to the standard capture PCM device.

name='Line2 Playback Volume',index=0

This control is used to attenuate samples from left and right I2S ADC inputs (on the Audigy-Drive). The result samples are forwarded to the virtual stereo mixer.

name='Line2 Capture Volume',index=1

This control is used to attenuate samples from left and right I2S ADC inputs (on the Audigy-Drive). The result is forwarded to the standard capture PCM device.

name='Analog Mix Playback Volume',index=0

This control is used to attenuate samples from left and right I2S ADC inputs from Philips ADC. The result samples are forwarded to the virtual stereo mixer. This contains mix from analog sources like CD, Line In, Aux, ...

name='Analog Mix Capture Volume',index=1

This control is used to attenuate samples from left and right I2S ADC inputs Philips ADC. The result is forwarded to the standard capture PCM device.

name='Aux2 Playback Volume',index=0

This control is used to attenuate samples from left and right I2S ADC inputs (on the Audigy-Drive). The result samples are forwarded to the virtual stereo mixer.

name='Aux2 Capture Volume',index=1

This control is used to attenuate samples from left and right I2S ADC inputs (on the Audigy-Drive). The result is forwarded to the standard capture PCM device.

name='Front Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the front speakers.

name='Surround Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the surround (rear) speakers.

name='Side Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the side speakers.

name='Center Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the center speaker.

name='LFE Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the subwoofer.

name='Tone Control - Switch',index=0

This control turns the tone control on or off. The samples forwarded to the speaker outputs are affected.

name='Tone Control - Bass',index=0

This control sets the bass intensity. There is no neutral value!! When the tone control code is activated, the samples are always modified. The closest value to pure signal is 20.

name='Tone Control - Treble',index=0

This control sets the treble intensity. There is no neutral value!! When the tone control code is activated, the samples are always modified. The closest value to pure signal is 20.

name='Master Playback Volume',index=0

This control is used to attenuate samples forwarded to the speaker outputs.

name='IEC958 Optical Raw Playback Switch',index=0

If this switch is on, then the samples for the IEC958 (S/PDIF) digital output are taken only from the raw iec958 ALSA PCM device (which uses accumulators 20 and 21 for left and right PCM by default).

6.4.2 PCM stream related controls

name='EMU10K1 PCM Volume',index 0-31

Channel volume attenuation in range 0-0x1fffd. The middle value (no attenuation) is default. The channel mapping for three values is as follows:

- 0 - mono, default 0xffff (no attenuation)
- 1 - left, default 0xffff (no attenuation)
- 2 - right, default 0xffff (no attenuation)

name='EMU10K1 PCM Send Routing',index 0-31

This control specifies the destination - FX-bus accumulators. There are 24 values in this mapping:

- 0 - mono, A destination (FX-bus 0-63), default 0
- 1 - mono, B destination (FX-bus 0-63), default 1
- 2 - mono, C destination (FX-bus 0-63), default 2
- 3 - mono, D destination (FX-bus 0-63), default 3
- 4 - mono, E destination (FX-bus 0-63), default 4
- 5 - mono, F destination (FX-bus 0-63), default 5
- 6 - mono, G destination (FX-bus 0-63), default 6
- 7 - mono, H destination (FX-bus 0-63), default 7
- 8 - left, A destination (FX-bus 0-63), default 0
- 9 - left, B destination (FX-bus 0-63), default 1
- 10 - left, C destination (FX-bus 0-63), default 2
- 11 - left, D destination (FX-bus 0-63), default 3
- 12 - left, E destination (FX-bus 0-63), default 4
- 13 - left, F destination (FX-bus 0-63), default 5
- 14 - left, G destination (FX-bus 0-63), default 6
- 15 - left, H destination (FX-bus 0-63), default 7
- 16 - right, A destination (FX-bus 0-63), default 0
- 17 - right, B destination (FX-bus 0-63), default 1
- 18 - right, C destination (FX-bus 0-63), default 2
- 19 - right, D destination (FX-bus 0-63), default 3
- 20 - right, E destination (FX-bus 0-63), default 4
- 21 - right, F destination (FX-bus 0-63), default 5
- 22 - right, G destination (FX-bus 0-63), default 6
- 23 - right, H destination (FX-bus 0-63), default 7

Don't forget that it's illegal to assign a channel to the same FX-bus accumulator more than once (it means 0=0 && 1=0 is an invalid combination).

name='EMU10K1 PCM Send Volume',index 0-31

It specifies the attenuation (amount) for given destination in range 0-255. The channel mapping is following:

- 0 - mono, A destination attn, default 255 (no attenuation)
- 1 - mono, B destination attn, default 255 (no attenuation)
- 2 - mono, C destination attn, default 0 (mute)
- 3 - mono, D destination attn, default 0 (mute)
- 4 - mono, E destination attn, default 0 (mute)
- 5 - mono, F destination attn, default 0 (mute)
- 6 - mono, G destination attn, default 0 (mute)
- 7 - mono, H destination attn, default 0 (mute)
- 8 - left, A destination attn, default 255 (no attenuation)
- 9 - left, B destination attn, default 0 (mute)
- 10 - left, C destination attn, default 0 (mute)
- 11 - left, D destination attn, default 0 (mute)
- 12 - left, E destination attn, default 0 (mute)
- 13 - left, F destination attn, default 0 (mute)
- 14 - left, G destination attn, default 0 (mute)
- 15 - left, H destination attn, default 0 (mute)
- 16 - right, A destination attn, default 0 (mute)
- 17 - right, B destination attn, default 255 (no attenuation)
- 18 - right, C destination attn, default 0 (mute)
- 19 - right, D destination attn, default 0 (mute)
- 20 - right, E destination attn, default 0 (mute)
- 21 - right, F destination attn, default 0 (mute)
- 22 - right, G destination attn, default 0 (mute)
- 23 - right, H destination attn, default 0 (mute)

6.4.3 MANUALS/PATENTS

See sb-live-mixer.rst.

6.5 E-MU Digital Audio System mixer / default DSP code

This document covers the E-MU 0404/1010/1212/1616/1820 PCI/PCI-e/CardBus cards.

These cards use regular EMU10K2 (SoundBlaster Audigy) chips, but with an alternative frontend geared towards semi-professional studio recording.

This document is based on audigy-mixer.rst.

6.5.1 Hardware compatibility

The EMU10K2 chips have a very short capture FIFO, which makes recording unreliable if the card's PCI bus requests are not handled with the appropriate priority. This is the case on more modern motherboards, where the PCI bus is only a secondary peripheral, rather than the actual arbiter of device access. In particular, I got recording glitches during simultaneous playback on an Intel DP55 board (memory controller in the CPU), but had success with an Intel DP45 board (memory controller in the north bridge).

The PCI Express variants of these cards (which have a PCI bridge on board, but are otherwise identical) may be less problematic.

6.5.2 Driver capabilities

This driver supports only 16-bit 44.1/48 kHz operation. The multi-channel device (see emu10k1-jack.rst) additionally supports 24-bit capture.

A patchset to enhance the driver is available from a GitHub repository. Its multi-channel device supports 24-bit for both playback and capture, and also supports full 88.2/96/176.4/192 kHz operation. It is not going to be upstreamed due to a fundamental disagreement about what constitutes a good user experience.

6.5.3 Digital mixer controls

Note that the controls work as attenuators: the maximum value is the neutral position leaving the signal unchanged. Note that if the same destination is mentioned in multiple controls, the signal is accumulated and can be clipped (set to maximal or minimal value without checking for overflow).

Explanation of used abbreviations:

DAC
digital to analog converter

ADC
analog to digital converter

LFE
low frequency effects (used as subwoofer signal)

IEC958
S/PDIF

FX-bus

> the EMU10K2 chip has an effect bus containing 64 accumulators. Each of the synthesizer voices can feed its output to these accumulators and the DSP microcontroller can operate with the resulting sum.

name='Clock Source',index=0

This control allows switching the word clock between interally generated 44.1 or 48 kHz, or a number of external sources.

Note: the sources for the 1616 CardBus card are unclear. Please report your findings.

name='Clock Fallback',index=0

This control determines the internal clock which the card switches to when the selected external clock source is/becomes invalid.

name='DAC1 0202 14dB PAD',index=0, etc.

Output attenuation controls. Not available on 0404 cards.

name='ADC1 14dB PAD 0202',index=0, etc.

Input attenuation controls. Not available on 0404 cards.

name='Optical Output Mode',index=0

Switches the TOSLINK output port between S/PDIF and ADAT. Not available on 0404 cards (fixed to S/PDIF).

name='Optical Input Mode',index=0

Switches the TOSLINK input port between S/PDIF and ADAT. Not available on 0404 cards (fixed to S/PDIF).

name='PCM Front Playback Volume',index=0

This control is used to attenuate samples from left and right front PCM FX-bus accumulators. ALSA uses accumulators 8 and 9 for left and right front PCM samples for 5.1 playback. The result samples are forwarded to the DSP 0 & 1 playback channels.

name='PCM Surround Playback Volume',index=0

This control is used to attenuate samples from left and right surround PCM FX-bus accumulators. ALSA uses accumulators 2 and 3 for left and right surround PCM samples for 5.1 playback. The result samples are forwarded to the DSP 2 & 3 playback channels.

name='PCM Side Playback Volume',index=0

This control is used to attenuate samples from left and right side PCM FX-bus accumulators. ALSA uses accumulators 14 and 15 for left and right side PCM samples for 7.1 playback. The result samples are forwarded to the DSP 6 & 7 playback channels.

name='PCM Center Playback Volume',index=0

This control is used to attenuate samples from the center PCM FX-bus accumulator. ALSA uses accumulator 6 for center PCM samples for 5.1 playback. The result samples are forwarded to the DSP 4 playback channel.

name='PCM LFE Playback Volume',index=0

This control is used to attenuate samples from the LFE PCM FX-bus accumulator. ALSA uses accumulator 7 for LFE PCM samples for 5.1 playback. The result samples are forwarded to the DSP 5 playback channel.

name='PCM Playback Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM samples for stereo playback. The result samples are forwarded to the virtual stereo mixer.

name='PCM Capture Volume',index=0

This control is used to attenuate samples from left and right PCM FX-bus accumulators. ALSA uses accumulators 0 and 1 for left and right PCM. The result is forwarded to the standard capture PCM device.

name='Music Playback Volume',index=0

This control is used to attenuate samples from left and right MIDI FX-bus accumulators. ALSA uses accumulators 4 and 5 for left and right MIDI samples. The result samples are forwarded to the virtual stereo mixer.

name='Music Capture Volume',index=0

These controls are used to attenuate samples from left and right MIDI FX-bus accumulator. ALSA uses accumulators 4 and 5 for left and right MIDI samples. The result is forwarded to the standard capture PCM device.

name='Front Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the DSP 0 & 1 playback channels.

name='Surround Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the DSP 2 & 3 playback channels.

name='Side Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the DSP 6 & 7 playback channels.

name='Center Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the DSP 4 playback channel.

name='LFE Playback Volume',index=0

This control is used to attenuate samples from the virtual stereo mixer. The result samples are forwarded to the DSP 5 playback channel.

name='Tone Control - Switch',index=0

This control turns the tone control on or off. The samples forwarded to the DSP playback channels are affected.

name='Tone Control - Bass',index=0

This control sets the bass intensity. There is no neutral value!! When the tone control code is activated, the samples are always modified. The closest value to pure signal is 20.

name='Tone Control - Treble',index=0

This control sets the treble intensity. There is no neutral value!! When the tone control code is activated, the samples are always modified. The closest value to pure signal is 20.

name='Master Playback Volume',index=0

This control is used to attenuate samples for all DSP playback channels.

name='EMU Capture Volume',index=0

This control is used to attenuate samples from the DSP 0 & 1 capture channels. The result is forwarded to the standard capture PCM device.

name='DAC Left',index=0, etc.

Select the source for the given physical audio output. These may be physical inputs, playback channels (DSP xx, specified as a decimal number), or silence.

name='DSP x',index=0

Select the source for the given capture channel (specified as a hexadecimal digit). Same options as for the physical audio outputs.

6.5.4 PCM stream related controls

These controls are described in audigy-mixer.rst.

6.5.5 MANUALS/PATENTS

See sb-live-mixer.rst.

6.6 Low latency, multichannel audio with JACK and the emu10k1/emu10k2

This document is a guide to using the emu10k1 based devices with JACK for low latency, multichannel recording functionality. All of my recent work to allow Linux users to use the full capabilities of their hardware has been inspired by the kX Project. Without their work I never would have discovered the true power of this hardware.

> **http://www.kxproject.com**
>> • Lee Revell, 2005.03.30

Until recently, emu10k1 users on Linux did not have access to the same low latency, multichannel features offered by the "kX ASIO" feature of their Windows driver. As of ALSA 1.0.9 this is no more!

For those unfamiliar with kX ASIO, this consists of 16 capture and 16 playback channels. With a post 2.6.9 Linux kernel, latencies down to 64 (1.33 ms) or even 32 (0.66ms) frames should work well.

The configuration is slightly more involved than on Windows, as you have to select the correct device for JACK to use. Actually, for qjackctl users it's fairly self explanatory - select Duplex, then for capture and playback select the multichannel devices, set the in and out channels to 16, and the sample rate to 48000Hz. The command line looks like this:

```
/usr/local/bin/jackd -R -dalsa -r48000 -p64 -n2 -D -Chw:0,2 -Phw:0,3 -S
```

This will give you 16 input ports and 16 output ports.

The 16 output ports map onto the 16 FX buses (or the first 16 of 64, for the Audigy). The mapping from FX bus to physical output is described in sb-live-mixer.rst (or audigy-mixer.rst).

The 16 input ports are connected to the 16 physical inputs. Contrary to popular belief, all emu10k1 cards are multichannel cards. Which of these input channels have physical inputs connected to them depends on the card model. Trial and error is highly recommended; the pinout diagrams for the card have been reverse engineered by some enterprising kX users and are available on the internet. Meterbridge is helpful here, and the kX forums are packed with useful information.

Each input port will either correspond to a digital (SPDIF) input, an analog input, or nothing. The one exception is the SBLive! 5.1. On these devices, the second and third input ports are wired to the center/LFE output. You will still see 16 capture channels, but only 14 are available for recording inputs.

This chart, borrowed from kxfxlib/da_asio51.cpp, describes the mapping of JACK ports to FXBUS2 (multitrack recording input) and EXTOUT (physical output) channels.

JACK (& ASIO) mappings on 10k1 5.1 SBLive cards:

JACK	Epilog	FXBUS2(nr)
capture_1	asio14	FXBUS2(0xe)
capture_2	asio15	FXBUS2(0xf)
capture_3	asio0	FXBUS2(0x0)
~capture_4	Center	EXTOUT(0x11) // mapped to by Center
~capture_5	LFE	EXTOUT(0x12) // mapped to by LFE
capture_6	asio3	FXBUS2(0x3)
capture_7	asio4	FXBUS2(0x4)
capture_8	asio5	FXBUS2(0x5)
capture_9	asio6	FXBUS2(0x6)
capture_10	asio7	FXBUS2(0x7)
capture_11	asio8	FXBUS2(0x8)
capture_12	asio9	FXBUS2(0x9)
capture_13	asio10	FXBUS2(0xa)
capture_14	asio11	FXBUS2(0xb)
capture_15	asio12	FXBUS2(0xc)
capture_16	asio13	FXBUS2(0xd)

TODO: describe use of ld10k1/qlo10k1 in conjunction with JACK

6.7 VIA82xx mixer

On many VIA82xx boards, the `Input Source Select` mixer control does not work. Setting it to `Input2` on such boards will cause recording to hang, or fail with EIO (input/output error) via OSS emulation. This control should be left at `Input1` for such cards.

6.8 Guide to using M-Audio Audiophile USB with ALSA and Jack

v1.5

Thibault Le Meur <Thibault.LeMeur@supelec.fr>

This document is a guide to using the M-Audio Audiophile USB (tm) device with ALSA and JACK.

6.8.1 History

- v1.4 - Thibault Le Meur (2007-07-11)
    - Added Low Endianness nature of 16bits-modes found by Hakan Lennestal <Hakan.Lennestal@brfsodrahamn.se>
    - Modifying document structure
- v1.5 - Thibault Le Meur (2007-07-12) - Added AC3/DTS passthru info

6.8.2 Audiophile USB Specs and correct usage

This part is a reminder of important facts about the functions and limitations of the device.

The device has 4 audio interfaces, and 2 MIDI ports:

- Analog Stereo Input (Ai)
    - This port supports 2 pairs of line-level audio inputs (1/4" TS and RCA)
    - When the 1/4" TS (jack) connectors are connected, the RCA connectors are disabled
- Analog Stereo Output (Ao)
- Digital Stereo Input (Di)
- Digital Stereo Output (Do)
- Midi In (Mi)
- Midi Out (Mo)

The internal DAC/ADC has the following characteristics:

- sample depth of 16 or 24 bits
- sample rate from 8kHz to 96kHz
- Two interfaces can't use different sample depths at the same time.

Moreover, the Audiophile USB documentation gives the following Warning:

Please exit any audio application running before switching between bit depths

Due to the USB 1.1 bandwidth limitation, a limited number of interfaces can be activated at the same time depending on the audio mode selected:

- 16-bit/48kHz ==> 4 channels in + 4 channels out

  - Ai+Ao+Di+Do

- 24-bit/48kHz ==> 4 channels in + 2 channels out, or 2 channels in + 4 channels out

  - Ai+Ao+Do or Ai+Di+Ao or Ai+Di+Do or Di+Ao+Do

- 24-bit/96kHz ==> 2 channels in _or_ 2 channels out (half duplex only)

  - Ai or Ao or Di or Do

Important facts about the Digital interface:

- The Do port additionally supports surround-encoded AC-3 and DTS passthrough, though I haven't tested it under Linux

  - Note that in this setup only the Do interface can be enabled

- Apart from recording an audio digital stream, enabling the Di port is a way to synchronize the device to an external sample clock

  - As a consequence, the Di port must be enable only if an active Digital source is connected

  - Enabling Di when no digital source is connected can result in a synchronization error (for instance sound played at an odd sample rate)

6.8.3 Audiophile USB MIDI support in ALSA

The Audiophile USB MIDI ports will be automatically supported once the following modules have been loaded:

- snd-usb-audio
- snd-seq-midi

No additional setting is required.

6.8.4 Audiophile USB Audio support in ALSA

Audio functions of the Audiophile USB device are handled by the snd-usb-audio module. This module can work in a default mode (without any device-specific parameter), or in an "advanced" mode with the device-specific parameter called `device_setup`.

Default Alsa driver mode

The default behavior of the snd-usb-audio driver is to list the device capabilities at startup and activate the required mode when required by the applications: for instance if the user is recording in a 24bit-depth-mode and immediately after wants to switch to a 16bit-depth mode, the snd-usb-audio module will reconfigure the device on the fly.

This approach has the advantage to let the driver automatically switch from sample rates/depths automatically according to the user's needs. However, those who are using the device under windows know that this is not how the device is meant to work: under windows applications must be closed before using the m-audio control panel to switch the device working mode. Thus as we'll see in next section, this Default Alsa driver mode can lead to device misconfigurations.

Let's get back to the Default Alsa driver mode for now. In this case the Audiophile interfaces are mapped to alsa pcm devices in the following way (I suppose the device's index is 1):

- hw:1,0 is Ao in playback and Di in capture
- hw:1,1 is Do in playback and Ai in capture
- hw:1,2 is Do in AC3/DTS passthrough mode

In this mode, the device uses Big Endian byte-encoding so that supported audio format are S16_BE for 16-bit depth modes and S24_3BE for 24-bits depth mode.

One exception is the hw:1,2 port which was reported to be Little Endian compliant (supposedly supporting S16_LE) but processes in fact only S16_BE streams. This has been fixed in kernel 2.6.23 and above and now the hw:1,2 interface is reported to be big endian in this default driver mode.

Examples:

- playing a S24_3BE encoded raw file to the Ao port:

```
% aplay -D hw:1,0 -c2 -t raw -r48000 -fS24_3BE test.raw
```

- recording a S24_3BE encoded raw file from the Ai port:

```
% arecord -D hw:1,1 -c2  -t raw -r48000 -fS24_3BE test.raw
```

- playing a S16_BE encoded raw file to the Do port:

```
% aplay -D hw:1,1 -c2 -t raw -r48000 -fS16_BE test.raw
```

- playing an ac3 sample file to the Do port:

```
% aplay -D hw:1,2 --channels=6 ac3_S16_BE_encoded_file.raw
```

If you're happy with the default Alsa driver mode and don't experience any issue with this mode, then you can skip the following chapter.

Advanced module setup

Due to the hardware constraints described above, the device initialization made by the Alsa driver in default mode may result in a corrupted state of the device. For instance, a particularly annoying issue is that the sound captured from the Ai interface sounds distorted (as if boosted with an excessive high volume gain).

For people having this problem, the snd-usb-audio module has a new module parameter called `device_setup` (this parameter was introduced in kernel release 2.6.17)

Initializing the working mode of the Audiophile USB

As far as the Audiophile USB device is concerned, this value let the user specify:

- the sample depth
- the sample rate
- whether the Di port is used or not

When initialized with `device_setup=0x00`, the snd-usb-audio module has the same behaviour as when the parameter is omitted (see paragraph "Default Alsa driver mode" above)

Others modes are described in the following subsections.

16-bit modes

The two supported modes are:

- `device_setup=0x01`
    - 16bits 48kHz mode with Di disabled
    - Ai,Ao,Do can be used at the same time
    - hw:1,0 is not available in capture mode
    - hw:1,2 is not available
- `device_setup=0x11`
    - 16bits 48kHz mode with Di enabled
    - Ai,Ao,Di,Do can be used at the same time
    - hw:1,0 is available in capture mode
    - hw:1,2 is not available

In this modes the device operates only at 16bits-modes. Before kernel 2.6.23, the devices where reported to be Big-Endian when in fact they were Little-Endian so that playing a file was a matter of using:

```
% aplay -D hw:1,1 -c2 -t raw -r48000 -fS16_BE test_S16_LE.raw
```

where "test_S16_LE.raw" was in fact a little-endian sample file.

Thanks to Hakan Lennestal (who discovered the Little-Endiannes of the device in these modes) a fix has been committed (expected in kernel 2.6.23) and Alsa now reports Little-Endian interfaces. Thus playing a file now is as simple as using:

```
% aplay -D hw:1,1 -c2 -t raw -r48000 -fS16_LE test_S16_LE.raw
```

24-bit modes

The three supported modes are:

- device_setup=0x09
    - 24bits 48kHz mode with Di disabled
    - Ai,Ao,Do can be used at the same time
    - hw:1,0 is not available in capture mode
    - hw:1,2 is not available
- device_setup=0x19
    - 24bits 48kHz mode with Di enabled
    - 3 ports from {Ai,Ao,Di,Do} can be used at the same time
    - hw:1,0 is available in capture mode and an active digital source must be connected to Di
    - hw:1,2 is not available
- device_setup=0x0D or 0x10
    - 24bits 96kHz mode
    - Di is enabled by default for this mode but does not need to be connected to an active source
    - Only 1 port from {Ai,Ao,Di,Do} can be used at the same time
    - hw:1,0 is available in captured mode
    - hw:1,2 is not available

In these modes the device is only Big-Endian compliant (see "Default Alsa driver mode" above for an aplay command example)

AC3 w/ DTS passthru mode

Thanks to Hakan Lennestal, I now have a report saying that this mode works.

- device_setup=0x03
    - 16bits 48kHz mode with only the Do port enabled
    - AC3 with DTS passthru
    - Caution with this setup the Do port is mapped to the pcm device hw:1,0

The command line used to playback the AC3/DTS encoded .wav-files in this mode:

```
% aplay -D hw:1,0 --channels=6 ac3_S16_LE_encoded_file.raw
```

How to use the `device_setup` parameter

The parameter can be given:

- By manually probing the device (as root)::

```
# modprobe -r snd-usb-audio
# modprobe snd-usb-audio index=1 device_setup=0x09
```

- Or while configuring the modules options in your modules configuration file (typically a .conf file in /etc/modprobe.d/ directory::

```
alias snd-card-1 snd-usb-audio
options snd-usb-audio index=1 device_setup=0x09
```

CAUTION when initializing the device

- Correct initialization on the device requires that device_setup is given to the module BE-FORE the device is turned on. So, if you use the "manual probing" method described above, take care to power-on the device AFTER this initialization.

- Failing to respect this will lead to a misconfiguration of the device. In this case turn off the device, unprobe the snd-usb-audio module, then probe it again with correct device_setup parameter and then (and only then) turn on the device again.

- If you've correctly initialized the device in a valid mode and then want to switch to another mode (possibly with another sample-depth), please use also the following procedure:

    - first turn off the device

    - de-register the snd-usb-audio module (modprobe -r)

    - change the device_setup parameter by changing the device_setup option in `/etc/modprobe.d/*.conf`

    - turn on the device

- A workaround for this last issue has been applied to kernel 2.6.23, but it may not be enough to ensure the 'stability' of the device initialization.

Technical details for hackers

This section is for hackers, wanting to understand details about the device internals and how Alsa supports it.

Audiophile USB's `device_setup` structure

If you want to understand the device_setup magic numbers for the Audiophile USB, you need some very basic understanding of binary computation. However, this is not required to use the parameter and you may skip this section.

The device_setup is one byte long and its structure is the following:

```
+---+---+---+---+---+---+---+---+
| b7| b6| b5| b4| b3| b2| b1| b0|
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | Di|24B|96K|DTS|SET|
+---+---+---+---+---+---+---+---+
```

Where:

- b0 is the SET bit

    - it MUST be set if device_setup is initialized

- b1 is the DTS bit

    - it is set only for Digital output with DTS/AC3

    - this setup is not tested

- b2 is the Rate selection flag

    - When set to 1 the rate range is 48.1-96kHz

    - Otherwise the sample rate range is 8-48kHz

- b3 is the bit depth selection flag

    - When set to 1 samples are 24bits long

    - Otherwise they are 16bits long

    - Note that b2 implies b3 as the 96kHz mode is only supported for 24 bits samples

- b4 is the Digital input flag

    - When set to 1 the device assumes that an active digital source is connected

    - You shouldn't enable Di if no source is seen on the port (this leads to synchronization issues)

    - b4 is implied by b2 (since only one port is enabled at a time no synch error can occur)

- b5 to b7 are reserved for future uses, and must be set to 0

    - might become Ao, Do, Ai, for b7, b6, b4 respectively

Caution:

- there is no check on the value you will give to device_setup

    - for instance choosing 0x05 (16bits 96kHz) will fail back to 0x09 since b2 implies b3. But _there_will_be_no_warning_ in /var/log/messages

- Hardware constraints due to the USB bus limitation aren't checked

---

> – choosing b2 will prepare all interfaces for 24bits/96kHz but you'll only be able to use one at the same time

USB implementation details for this device

You may safely skip this section if you're not interested in driver hacking.

This section describes some internal aspects of the device and summarizes the data I got by usb-snooping the windows and Linux drivers.

The M-Audio Audiophile USB has 7 USB Interfaces: a "USB interface":

- USB Interface nb.0
- USB Interface nb.1
  - Audio Control function
- USB Interface nb.2
  - Analog Output
- USB Interface nb.3
  - Digital Output
- USB Interface nb.4
  - Analog Input
- USB Interface nb.5
  - Digital Input
- USB Interface nb.6
  - MIDI interface compliant with the MIDIMAN quirk

Each interface has 5 altsettings (AltSet 1,2,3,4,5) except:

- Interface 3 (Digital Out) has an extra Alset nb.6
- Interface 5 (Digital In) does not have Alset nb.3 and 5

Here is a short description of the AltSettings capabilities:

- AltSettings 1 corresponds to
  - 24-bit depth, 48.1-96kHz sample mode
  - Adaptive playback (Ao and Do), Synch capture (Ai), or Asynch capture (Di)
- AltSettings 2 corresponds to
  - 24-bit depth, 8-48kHz sample mode
  - Asynch capture and playback (Ao,Ai,Do,Di)
- AltSettings 3 corresponds to
  - 24-bit depth, 8-48kHz sample mode
  - Synch capture (Ai) and Adaptive playback (Ao,Do)
- AltSettings 4 corresponds to

- **–** 16-bit depth, 8-48kHz sample mode

- **–** Asynch capture and playback (Ao,Ai,Do,Di)

- AltSettings 5 corresponds to

  - **–** 16-bit depth, 8-48kHz sample mode

  - **–** Synch capture (Ai) and Adaptive playback (Ao,Do)

- AltSettings 6 corresponds to

  - **–** 16-bit depth, 8-48kHz sample mode

  - **–** Synch playback (Do), audio format type III IEC1937_AC-3

In order to ensure a correct initialization of the device, the driver *must know* how the device will be used:

- if DTS is chosen, only Interface 2 with AltSet nb.6 must be registered

- if 96KHz only AltSets nb.1 of each interface must be selected

- if samples are using 24bits/48KHz then AltSet 2 must me used if Digital input is connected, and only AltSet nb.3 if Digital input is not connected

- if samples are using 16bits/48KHz then AltSet 4 must me used if Digital input is connected, and only AltSet nb.5 if Digital input is not connected

When device_setup is given as a parameter to the snd-usb-audio module, the parse_audio_endpoints function uses a quirk called `audiophile_skip_setting_quirk` in order to prevent AltSettings not corresponding to device_setup from being registered in the driver.

6.8.5 Audiophile USB and Jack support

This section deals with support of the Audiophile USB device in Jack.

There are 2 main potential issues when using Jackd with the device:

- support for Big-Endian devices in 24-bit modes

- support for 4-in / 4-out channels

Direct support in Jackd

Jack supports big endian devices only in recent versions (thanks to Andreas Steinmetz for his first big-endian patch). I can't remember exactly when this support was released into jackd, let's just say that with jackd version 0.103.0 it's almost ok (just a small bug is affecting 16bits Big-Endian devices, but since you've read carefully the above paragraphs, you're now using kernel >= 2.6.23 and your 16bits devices are now Little Endians ;-)).

You can run jackd with the following command for playback with Ao and record with Ai:

```
% jackd -R -dalsa -Phw:1,0 -r48000 -p128 -n2 -D -Chw:1,1
```

Using Alsa plughw

If you don't have a recent Jackd installed, you can downgrade to using the Alsa `plug` converter.

For instance here is one way to run Jack with 2 playback channels on Ao and 2 capture channels from Ai:

```
% jackd -R -dalsa -dplughw:1 -r48000 -p256 -n2 -D -Cplughw:1,1
```

However you may see the following warning message:
> You appear to be using the ALSA software "plug" layer, probably a result of using the "default" ALSA device. This is less efficient than it could be. Consider using a hardware device instead rather than using the plug layer.

Getting 2 input and/or output interfaces in Jack

As you can see, starting the Jack server this way will only enable 1 stereo input (Di or Ai) and 1 stereo output (Ao or Do).

This is due to the following restrictions:

- Jack can only open one capture device and one playback device at a time
- The Audiophile USB is seen as 2 (or three) Alsa devices: hw:1,0, hw:1,1 (and optionally hw:1,2)

If you want to get Ai+Di and/or Ao+Do support with Jack, you would need to combine the Alsa devices into one logical "complex" device.

If you want to give it a try, I recommend reading the information from this page: http://www.sound-man.co.uk/linuxaudio/ice1712multi.html It is related to another device (ice1712) but can be adapted to suit the Audiophile USB.

Enabling multiple Audiophile USB interfaces for Jackd will certainly require:

- Making sure your Jackd version has the MMAP_COMPLEX patch (see the ice1712 page)
- (maybe) patching the alsa-lib/src/pcm/pcm_multi.c file (see the ice1712 page)
- define a multi device (combination of hw:1,0 and hw:1,1) in your .asoundrc file
- start jackd with this device

I had no success in testing this for now, if you have any success with this kind of setup, please drop me an email.

6.9 Alsa driver for Digigram miXart8 and miXart8AES/EBU soundcards

Digigram <alsa@digigram.com>

6.9.1 GENERAL

The miXart8 is a multichannel audio processing and mixing soundcard that has 4 stereo audio inputs and 4 stereo audio outputs. The miXart8AES/EBU is the same with a add-on card that offers further 4 digital stereo audio inputs and outputs. Furthermore the add-on card offers external clock synchronisation (AES/EBU, Word Clock, Time Code and Video Synchro)

The mainboard has a PowerPC that offers onboard mpeg encoding and decoding, samplerate conversions and various effects.

The driver don't work properly at all until the certain firmwares are loaded, i.e. no PCM nor mixer devices will appear. Use the mixartloader that can be found in the alsa-tools package.

6.9.2 VERSION 0.1.0

One miXart8 board will be represented as 4 alsa cards, each with 1 stereo analog capture 'pcm0c' and 1 stereo analog playback 'pcm0p' device. With a miXart8AES/EBU there is in addition 1 stereo digital input 'pcm1c' and 1 stereo digital output 'pcm1p' per card.

Formats

U8, S16_LE, S16_BE, S24_3LE, S24_3BE, FLOAT_LE, FLOAT_BE Sample rates : 8000 - 48000 Hz continuously

Playback

For instance the playback devices are configured to have max. 4 substreams performing hardware mixing. This could be changed to a maximum of 24 substreams if wished. Mono files will be played on the left and right channel. Each channel can be muted for each stream to use 8 analog/digital outputs separately.

Capture

There is one substream per capture device. For instance only stereo formats are supported.

Mixer

<Master> and <Master Capture>
    analog volume control of playback and capture PCM.

<PCM 0-3> and <PCM Capture>
    digital volume control of each analog substream.

<AES 0-3> and <AES Capture>
    digital volume control of each AES/EBU substream.

<Monitoring>
    Loopback from 'pcm0c' to 'pcm0p' with digital volume and mute control.

Rem : for best audio quality try to keep a 0 attenuation on the PCM and AES volume controls which is set by 219 in the range from 0 to 255 (about 86% with alsamixer)

6.9.3 NOT YET IMPLEMENTED

- external clock support (AES/EBU, Word Clock, Time Code, Video Sync)
- MPEG audio formats
- mono record
- on-board effects and samplerate conversions
- linked streams

6.9.4 FIRMWARE

[As of 2.6.11, the firmware can be loaded automatically with hotplug
when CONFIG_FW_LOADER is set. The mixartloader is necessary only for older versions or when you build the driver into kernel.]

For loading the firmware automatically after the module is loaded, use a install command. For example, add the following entry to /etc/modprobe.d/mixart.conf for miXart driver:

```
install snd-mixart /sbin/modprobe --first-time -i snd-mixart && \
                   /usr/bin/mixartloader
```

(for 2.2/2.4 kernels, add "post-install snd-mixart /usr/bin/vxloader" to /etc/modules.conf, instead.)

The firmware binaries are installed on /usr/share/alsa/firmware (or /usr/local/share/alsa/firmware, depending to the prefix option of configure). There will be a miXart.conf file, which define the dsp image files.

The firmware files are copyright by Digigram SA

6.9.5 COPYRIGHT

Copyright (c) 2003 Digigram SA <alsa@digigram.com> Distributable under GPL.

6.10 ALSA BT87x Driver

6.10.1 Intro

You might have noticed that the bt878 grabber cards have actually *two* PCI functions:

```
$ lspci
[ ... ]
00:0a.0 Multimedia video controller: Brooktree Corporation Bt878 (rev 02)
00:0a.1 Multimedia controller: Brooktree Corporation Bt878 (rev 02)
[ ... ]
```

The first does video, it is backward compatible to the bt848. The second does audio. snd-bt87x is a driver for the second function. It's a sound driver which can be used for recording sound (and *only* recording, no playback). As most TV cards come with a short cable which can be

plugged into your sound card's line-in you probably don't need this driver if all you want to do is just watching TV...

Some cards do not bother to connect anything to the audio input pins of the chip, and some other cards use the audio function to transport MPEG video data, so it's quite possible that audio recording may not work with your card.

6.10.2 Driver Status

The driver is now stable. However, it doesn't know about many TV cards, and it refuses to load for cards it doesn't know.

If the driver complains (”Unknown TV card found, the audio driver will not load”), you can specify the `load_all=1` option to force the driver to try to use the audio capture function of your card. If the frequency of recorded data is not right, try to specify the `digital_rate` option with other values than the default 32000 (often it's 44100 or 64000).

If you have an unknown card, please mail the ID and board name to <alsa-devel@alsa-project.org>, regardless of whether audio capture works or not, so that future versions of this driver know about your card.

6.10.3 Audio modes

The chip knows two different modes (digital/analog). snd-bt87x registers two PCM devices, one for each mode. They cannot be used at the same time.

6.10.4 Digital audio mode

The first device (hw:X,0) gives you 16 bit stereo sound. The sample rate depends on the external source which feeds the Bt87x with digital sound via I2S interface.

6.10.5 Analog audio mode (A/D)

The second device (hw:X,1) gives you 8 or 16 bit mono sound. Supported sample rates are between 119466 and 448000 Hz (yes, these numbers are that high). If you've set the CONFIG_SND_BT87X_OVERCLOCK option, the maximum sample rate is 1792000 Hz, but audio data becomes unusable beyond 896000 Hz on my card.

The chip has three analog inputs. Consequently you'll get a mixer device to control these.

Have fun,

>   Clemens

Written by Clemens Ladisch <clemens@ladisch.de> big parts copied from btaudio.txt by Gerd Knorr <kraxel@bytesex.org>

6.11 Notes on Maya44 USB Audio Support

Note: The following is the original document of Rainer's patch that the current maya44 code based on. Some contents might be obsoleted, but I keep here as reference -- tiwai

Feb 14, 2008

Rainer Zimmermann <mail@lightshed.de>

6.11.1 STATE OF DEVELOPMENT

This driver is being developed on the initiative of Piotr Makowski (oponek@gmail.com) and financed by Lars Bergmann. Development is carried out by Rainer Zimmermann (mail@lightshed.de).

ESI provided a sample Maya44 card for the development work.

However, unfortunately it has turned out difficult to get detailed programming information, so I (Rainer Zimmermann) had to find out some card-specific information by experiment and conjecture. Some information (in particular, several GPIO bits) is still missing.

This is the first testing version of the Maya44 driver released to the alsa-devel mailing list (Feb 5, 2008).

The following functions work, as tested by Rainer Zimmermann and Piotr Makowski:

- playback and capture at all sampling rates
- input/output level
- crossmixing
- line/mic switch
- phantom power switch
- analogue monitor a.k.a bypass

The following functions *should* work, but are not fully tested:

- Channel 3+4 analogue - S/PDIF input switching
- S/PDIF output
- all inputs/outputs on the M/IO/DIO extension card
- internal/external clock selection

In particular, we would appreciate testing of these functions by anyone who has access to an M/IO/DIO extension card.

Things that do not seem to work:

- The level meters ("multi track") in 'alsamixer' do not seem to react to signals in (if this is a bug, it would probably be in the existing ICE1724 code).
- Ardour 2.1 seems to work only via JACK, not using ALSA directly or via OSS. This still needs to be tracked down.

6.11.2 DRIVER DETAILS

the following files were added:

- pci/ice1724/maya44.c - Maya44 specific code
- pci/ice1724/maya44.h
- pci/ice1724/ice1724.patch
- pci/ice1724/ice1724.h.patch - PROPOSED patch to ice1724.h (see SAMPLING RATES)
- i2c/other/wm8776.c - low-level access routines for Wolfson WM8776 codecs
- include/wm8776.h

Note that the wm8776.c code is meant to be card-independent and does not actually register the codec with the ALSA infrastructure. This is done in maya44.c, mainly because some of the WM8776 controls are used in Maya44-specific ways, and should be named appropriately.

the following files were created in pci/ice1724, simply #including the corresponding file from the alsa-kernel tree:

- wtm.h
- vt1720_mobo.h
- revo.h
- prodigy192.h
- pontis.h
- phase.h
- maya44.h
- juli.h
- aureon.h
- amp.h
- envy24ht.h
- se.h
- prodigy_hifi.h

I hope this is the correct way to do things.

6.11.3 SAMPLING RATES

The Maya44 card (or more exactly, the Wolfson WM8776 codecs) allow a maximum sampling rate of 192 kHz for playback and 92 kHz for capture.

As the ICE1724 chip only allows one global sampling rate, this is handled as follows:

- setting the sampling rate on any open PCM device on the maya44 card will always set the *global* sampling rate for all playback and capture channels.
- In the current state of the driver, setting rates of up to 192 kHz is permitted even for capture devices.

AVOID CAPTURING AT RATES ABOVE 96kHz, even though it may appear to work. The codec cannot actually capture at such rates, meaning poor quality.

I propose some additional code for limiting the sampling rate when setting on a capture pcm device. However because of the global sampling rate, this logic would be somewhat problematic.

The proposed code (currently deactivated) is in ice1712.h.patch, ice1724.c and maya44.c (in pci/ice1712).

6.11.4 SOUND DEVICES

PCM devices correspond to inputs/outputs as follows (assuming Maya44 is card #0):

- hw:0,0 input - stereo, analog input 1+2
- hw:0,0 output - stereo, analog output 1+2
- hw:0,1 input - stereo, analog input 3+4 OR S/PDIF input
- hw:0,1 output - stereo, analog output 3+4 (and SPDIF out)

6.11.5 NAMING OF MIXER CONTROLS

(for more information about the signal flow, please refer to the block diagram on p.24 of the ESI Maya44 manual, or in the ESI windows software).

PCM
　　(digital) output level for channel 1+2

PCM 1
　　same for channel 3+4

Mic Phantom+48V
　　switch for +48V phantom power for electrostatic microphones on input 1/2.

　　Make sure this is not turned on while any other source is connected to input 1/2. It might damage the source and/or the maya44 card.

Mic/Line input
　　if switch is on, input jack 1/2 is microphone input (mono), otherwise line input (stereo).

Bypass
　　analogue bypass from ADC input to output for channel 1+2. Same as ”Monitor” in the windows driver.

Bypass 1
　　same for channel 3+4.

Crossmix
　　cross-mixer from channels 1+2 to channels 3+4

Crossmix 1
　　cross-mixer from channels 3+4 to channels 1+2

IEC958 Output
　　switch for S/PDIF output.

This is not supported by the ESI windows driver. S/PDIF should output the same signal as channel 3+4. [untested!]

Digital output selectors
These switches allow a direct digital routing from the ADCs to the DACs. Each switch determines where the digital input data to one of the DACs comes from. They are not supported by the ESI windows driver. For normal operation, they should all be set to "PCM out".

H/W
Output source channel 1

H/W 1
Output source channel 2

H/W 2
Output source channel 3

H/W 3
Output source channel 4

H/W 4 ... H/W 9
unknown function, left in to enable testing.

Possibly some of these control S/PDIF output(s). If these turn out to be unused, they will go away in later driver versions.

Selectable values for each of the digital output selectors are:

PCM out
DAC output of the corresponding channel (default setting)

Input 1 ... Input 4
direct routing from ADC output of the selected input channel

6.12 Software Interface ALSA-DSP MADI Driver

(translated from German, so no good English ;-),

2004 - winfried ritsch

Full functionality has been added to the driver. Since some of the Controls and startup-options are ALSA-Standard and only the special Controls are described and discussed below.

6.12.1 Hardware functionality

Audio transmission

- number of channels -- depends on transmission mode

    The number of channels chosen is from 1..Nmax. The reason to use for a lower number of channels is only resource allocation, since unused DMA channels are disabled and less memory is allocated. So also the throughput of the PCI system can be scaled. (Only important for low performance boards).

- Single Speed -- 1..64 channels

Note: (Note: Choosing the 56channel mode for transmission or as receiver, only 56 are transmitted/received over the MADI, but all 64 channels are available for the mixer, so channel count for the driver)

- Double Speed -- 1..32 channels

Note: Note: Choosing the 56-channel mode for transmission/receive-mode , only 28 are transmitted/received over the MADI, but all 32 channels are available for the mixer, so channel count for the driver

- Quad Speed -- 1..16 channels

Note: Choosing the 56-channel mode for transmission/receive-mode , only 14 are transmitted/received over the MADI, but all 16 channels are available for the mixer, so channel count for the driver

- Format -- signed 32 Bit Little Endian (SNDRV_PCM_FMTBIT_S32_LE)
- Sample Rates --

    Single Speed -- 32000, 44100, 48000

    Double Speed -- 64000, 88200, 96000 (untested)

    Quad Speed -- 128000, 176400, 192000 (untested)
- access-mode -- MMAP (memory mapped), Not interleaved (PCM_NON-INTERLEAVED)
- buffer-sizes -- 64,128,256,512,1024,2048,8192 Samples
- fragments -- 2
- Hardware-pointer -- 2 Modi

    The Card supports the readout of the actual Buffer-pointer, where DMA reads/writes. Since of the bulk mode of PCI it is only 64 Byte accurate. SO it is not really usable for the ALSA-mid-level functions (here the buffer-ID gives a better result), but if MMAP is used by the application. Therefore it can be configured at load-time with the parameter precise-pointer.

Hint:

(Hint: Experimenting I found that the pointer is maximum 64 to large never to small. So if you subtract 64 you always have a safe pointer for writing, which is used on this mode inside ALSA. In theory now you can get now a latency as low as 16 Samples, which is a quarter of the interrupt possibilities.)

- **Precise Pointer -- off**
    interrupt used for pointer-calculation

- **Precise Pointer -- on**
    hardware pointer used.

Controller

Since DSP-MADI-Mixer has 8152 Fader, it does not make sense to use the standard mixer-controls, since this would break most of (especially graphic) ALSA-Mixer GUIs. So Mixer control has be provided by a 2-dimensional controller using the hwdep-interface.

Also all 128+256 Peak and RMS-Meter can be accessed via the hwdep-interface. Since it could be a performance problem always copying and converting Peak and RMS-Levels even if you just need one, I decided to export the hardware structure, so that of needed some driver-guru can implement a memory-mapping of mixer or peak-meters over ioctl, or also to do only copying and no conversion. A test-application shows the usage of the controller.

- Latency Controls --- not implemented !!!

Note: Note: Within the windows-driver the latency is accessible of a control-panel, but buffer-sizes are controlled with ALSA from hwparams-calls and should not be changed in run-state, I did not implement it here.

- System Clock -- suspended !!!!
    - Name -- "System Clock Mode"
    - Access -- Read Write
    - Values -- "Master" "Slave"

Note: !!!! This is a hardware-function but is in conflict with the Clock-source controller, which is a kind of ALSA-standard. I makes sense to set the card to a special mode (master at some frequency or slave), since even not using an Audio-application a studio should have working synchronisations setup. So use Clock-source-controller instead !!!!

- Clock Source
    - Name -- "Sample Clock Source"
    - Access -- Read Write
    - Values -- "AutoSync", "Internal 32.0 kHz", "Internal 44.1 kHz", "Internal 48.0 kHz", "Internal 64.0 kHz", "Internal 88.2 kHz", "Internal 96.0 kHz"

        Choose between Master at a specific Frequency and so also the Speed-mode or Slave (Autosync). Also see "Preferred Sync Ref"

Warning: !!!! This is no pure hardware function but was implemented by ALSA by some ALSA-drivers before, so I use it also. !!!

- Preferred Sync Ref
    - Name -- "Preferred Sync Reference"
    - Access -- Read Write
    - Values -- "Word" "MADI"

Within the Auto-sync-Mode the preferred Sync Source can be chosen. If it is not available another is used if possible.

Note: Note: Since MADI has a much higher bit-rate than word-clock, the card should synchronise better in MADI Mode. But since the RME-PLL is very good, there are almost no problems with word-clock too. I never found a difference.

- TX 64 channel

    - Name -- "TX 64 channels mode"

    - Access -- Read Write

    - Values -- 0 1

        Using 64-channel-modus (1) or 56-channel-modus for MADI-transmission (0).

Note: Note: This control is for output only. Input-mode is detected automatically from hardware sending MADI.

- Clear TMS

    - Name -- "Clear Track Marker"

    - Access -- Read Write

    - Values -- 0 1

        Don't use to lower 5 Audio-bits on AES as additional Bits.

- Safe Mode oder Auto Input

    - Name -- "Safe Mode"

    - Access -- Read Write

    - Values -- 0 1 (default on)

        If on (1), then if either the optical or coaxial connection has a failure, there is a takeover to the working one, with no sample failure. Its only useful if you use the second as a backup connection.

- Input

    - Name -- "Input Select"

    - Access -- Read Write

    - Values -- optical coaxial

        Choosing the Input, optical or coaxial. If Safe-mode is active, this is the preferred Input.

Mixer

- Mixer

    - Name -- "Mixer"

    - Access -- Read Write

    - Values - <channel-number 0-127> <Value 0-65535>

        Here as a first value the channel-index is taken to get/set the corresponding mixer channel, where 0-63 are the input to output fader and 64-127 the playback to outputs fader. Value 0 is channel muted 0 and 32768 an amplification of 1.

- Chn 1-64

    fast mixer for the ALSA-mixer utils. The diagonal of the mixer-matrix is implemented from playback to output.

- Line Out

    - Name -- "Line Out"

    - Access -- Read Write

    - Values -- 0 1

        Switching on and off the analog out, which has nothing to do with mixing or routing. the analog outs reflects channel 63,64.

Information (only read access)

- Sample Rate

    - Name -- "System Sample Rate"

    - Access -- Read-only

        getting the sample rate.

- External Rate measured

    - Name -- "External Rate"

    - Access -- Read only

        Should be "Autosync Rate", but Name used is ALSA-Scheme. External Sample frequency liked used on Autosync is reported.

- MADI Sync Status

    - Name -- "MADI Sync Lock Status"

    - Access -- Read

    - Values -- 0,1,2

        MADI-Input is 0=Unlocked, 1=Locked, or 2=Synced.

- Word Clock Sync Status

    - Name -- "Word Clock Lock Status"

- **–** Access -- Read
- **–** Values -- 0,1,2

    Word Clock Input is 0=Unlocked, 1=Locked, or 2=Synced.

- AutoSync
    - **–** Name -- "AutoSync Reference"
    - **–** Access -- Read
    - **–** Values -- "WordClock", "MADI", "None"

        Sync-Reference is either "WordClock", "MADI" or none.

- RX 64ch --- noch nicht implementiert

    MADI-Receiver is in 64 channel mode oder 56 channel mode.

- AB_inp --- not tested

    Used input for Auto-Input.

- actual Buffer Position --- not implemented

    !!! this is a ALSA internal function, so no control is used !!!

6.12.2 Calling Parameter

- index int array (min = 1, max = 8)

    Index value for RME HDSPM interface. card-index within ALSA

    note: ALSA-standard

- id string array (min = 1, max = 8)

    ID string for RME HDSPM interface.

    note: ALSA-standard

- enable int array (min = 1, max = 8)

    Enable/disable specific HDSPM sound-cards.

    note: ALSA-standard

- precise_ptr int array (min = 1, max = 8)

    Enable precise pointer, or disable.

Note: note: Use only when the application supports this (which is a special case).

- line_outs_monitor int array (min = 1, max = 8)

    Send playback streams to analog outs by default.

> **Note:** note: each playback channel is mixed to the same numbered output channel (routed). This is against the ALSA-convention, where all channels have to be muted on after loading the driver, but was used before on other cards, so i historically use it again)

- enable_monitor int array (min = 1, max = 8)

    Enable Analog Out on Channel 63/64 by default.

> **Note:** note: here the analog output is enabled (but not routed).

6.13 Serial UART 16450/16550 MIDI driver

The adaptor module parameter allows you to select either:

- 0 - Roland Soundcanvas support (default)
- 1 - Midiator MS-124T support (1)
- 2 - Midiator MS-124W S/A mode (2)
- 3 - MS-124W M/B mode support (3)
- 4 - Generic device with multiple input support (4)

For the Midiator MS-124W, you must set the physical M-S and A-B switches on the Midiator to match the driver mode you select.

In Roland Soundcanvas mode, multiple ALSA raw MIDI substreams are supported (midiCnD0-midiCnD15). Whenever you write to a different substream, the driver sends the nonstandard MIDI command sequence F5 NN, where NN is the substream number plus 1. Roland modules use this command to switch between different "parts", so this feature lets you treat each part as a distinct raw MIDI substream. The driver provides no way to send F5 00 (no selection) or to not send the F5 NN command sequence at all; perhaps it ought to.

Usage example for simple serial converter:

```
/sbin/setserial /dev/ttyS0 uart none
/sbin/modprobe snd-serial-u16550 port=0x3f8 irq=4 speed=115200
```

Usage example for Roland SoundCanvas with 4 MIDI ports:

```
/sbin/setserial /dev/ttyS0 uart none
/sbin/modprobe snd-serial-u16550 port=0x3f8 irq=4 outs=4
```

In MS-124T mode, one raw MIDI substream is supported (midiCnD0); the outs module parameter is automatically set to 1. The driver sends the same data to all four MIDI Out connectors. Set the A-B switch and the speed module parameter to match (A=19200, B=9600).

Usage example for MS-124T, with A-B switch in A position:

```
/sbin/setserial /dev/ttyS0 uart none
/sbin/modprobe snd-serial-u16550 port=0x3f8 irq=4 adaptor=1 \
                speed=19200
```

In MS-124W S/A mode, one raw MIDI substream is supported (midiCnD0); the outs module parameter is automatically set to 1. The driver sends the same data to all four MIDI Out connectors at full MIDI speed.

Usage example for S/A mode:

```
/sbin/setserial /dev/ttyS0 uart none
/sbin/modprobe snd-serial-u16550 port=0x3f8 irq=4 adaptor=2
```

In MS-124W M/B mode, the driver supports 16 ALSA raw MIDI substreams; the outs module parameter is automatically set to 16. The substream number gives a bitmask of which MIDI Out connectors the data should be sent to, with midiCnD1 sending to Out 1, midiCnD2 to Out 2, midiCnD4 to Out 3, and midiCnD8 to Out 4. Thus midiCnD15 sends the data to all 4 ports. As a special case, midiCnD0 also sends to all ports, since it is not useful to send the data to no ports. M/B mode has extra overhead to select the MIDI Out for each byte, so the aggregate data rate across all four MIDI Outs is at most one byte every 520 us, as compared with the full MIDI data rate of one byte every 320 us per port.

Usage example for M/B mode:

```
/sbin/setserial /dev/ttyS0 uart none
/sbin/modprobe snd-serial-u16550 port=0x3f8 irq=4 adaptor=3
```

The MS-124W hardware's M/A mode is currently not supported. This mode allows the MIDI Outs to act independently at double the aggregate throughput of M/B, but does not allow sending the same byte simultaneously to multiple MIDI Outs. The M/A protocol requires the driver to twiddle the modem control lines under timing constraints, so it would be a bit more complicated to implement than the other modes.

Midiator models other than MS-124W and MS-124T are currently not supported. Note that the suffix letter is significant; the MS-124 and MS-124B are not compatible, nor are the other known models MS-101, MS-101B, MS-103, and MS-114. I do have documentation (tim.mann@compaq.com) that partially covers these models, but no units to experiment with. The MS-124W support is tested with a real unit. The MS-124T support is untested, but should work.

The Generic driver supports multiple input and output substreams over a single serial port. Similar to Roland Soundcanvas mode, F5 NN is used to select the appropriate input or output stream (depending on the data direction). Additionally, the CTS signal is used to regulate the data flow. The number of inputs is specified by the ins parameter.

6.14 Imagination Technologies SPDIF Input Controllers

The Imagination Technologies SPDIF Input controller contains the following controls:

- name='IEC958 Capture Mask',index=0

This control returns a mask that shows which of the IEC958 status bits can be read using the 'IEC958 Capture Default' control.

- name='IEC958 Capture Default',index=0

This control returns the status bits contained within the SPDIF stream that is being received. The 'IEC958 Capture Mask' shows which bits can be read from this control.

- name='SPDIF In Multi Frequency Acquire',index=0
- name='SPDIF In Multi Frequency Acquire',index=1
- name='SPDIF In Multi Frequency Acquire',index=2
- name='SPDIF In Multi Frequency Acquire',index=3

This control is used to attempt acquisition of up to four different sample rates. The active rate can be obtained by reading the 'SPDIF In Lock Frequency' control.

When the value of this control is set to {0,0,0,0}, the rate given to hw_params will determine the single rate the block will capture. Else, the rate given to hw_params will be ignored, and the block will attempt capture for each of the four sample rates set here.

If less than four rates are required, the same rate can be specified more than once

- name='SPDIF In Lock Frequency',index=0

This control returns the active capture rate, or 0 if a lock has not been acquired

- name='SPDIF In Lock TRK',index=0

This control is used to modify the locking/jitter rejection characteristics of the block. Larger values increase the locking range, but reduce jitter rejection.

- name='SPDIF In Lock Acquire Threshold',index=0

This control is used to change the threshold at which a lock is acquired.

- name='SPDIF In Lock Release Threshold',index=0

This control is used to change the threshold at which a lock is released.

6.15 The Virtual PCM Test Driver

The Virtual PCM Test Driver emulates a generic PCM device, and can be used for testing/fuzzing of the userspace ALSA applications, as well as for testing/fuzzing of the PCM middle layer. Additionally, it can be used for simulating hard to reproduce problems with PCM devices.

6.15.1 What can this driver do?

At this moment the driver can do the following things:

- Simulate both capture and playback processes
- Generate random or pattern-based capturing data
- Inject delays into the playback and capturing processes
- Inject errors during the PCM callbacks

It supports up to 8 substreams and 4 channels. Also it supports both interleaved and non-interleaved access modes.

Also, this driver can check the playback stream for containing the predefined pattern, which is used in the corresponding selftest (alsa/pcmtest-test.sh) to check the PCM middle layer data transferring functionality. Additionally, this driver redefines the default RESET ioctl, and the selftest covers this PCM API functionality as well.

Configuration

The driver has several parameters besides the common ALSA module parameters:

- fill_mode (bool) - Buffer fill mode (see below)
- inject_delay (int)
- inject_hwpars_err (bool)
- inject_prepare_err (bool)
- inject_trigger_err (bool)

Capture Data Generation

The driver has two modes of data generation: the first (0 in the fill_mode parameter) means random data generation, the second (1 in the fill_mode) - pattern-based data generation. Let's look at the second mode.

First of all, you may want to specify the pattern for data generation. You can do it by writing the pattern to the debugfs file. There are pattern buffer debugfs entries for each channel, as well as entries which contain the pattern buffer length.

- /sys/kernel/debug/pcmtest/fill_pattern[0-3]
- /sys/kernel/debug/pcmtest/fill_pattern[0-3]_len

To set the pattern for the channel 0 you can execute the following command:

```
echo -n mycoolpattern > /sys/kernel/debug/pcmtest/fill_pattern0
```

Then, after every capture action performed on the 'pcmtest' device the buffer for the channel 0 will contain 'mycoolpatternmycoolpatternmycoolpatternmy...'.

The pattern itself can be up to 4096 bytes long.

Delay injection

The driver has 'inject_delay' parameter, which has very self-descriptive name and can be used for time delay/speedup simulations. The parameter has integer type, and it means the delay added between module's internal timer ticks.

If the 'inject_delay' value is positive, the buffer will be filled slower, if it is negative - faster. You can try it yourself by starting a recording in any audiorecording application (like Audacity) and selecting the 'pcmtest' device as a source.

This parameter can be also used for generating a huge amount of sound data in a very short period of time (with the negative 'inject_delay' value).

Errors injection

This module can be used for injecting errors into the PCM communication process. This action can help you to figure out how the userspace ALSA program behaves under unusual circumstances.

For example, you can make all 'hw_params' PCM callback calls return EBUSY error by writing '1' to the 'inject_hwpars_err' module parameter:

```
echo 1 > /sys/module/snd_pcmtest/parameters/inject_hwpars_err
```

Errors can be injected into the following PCM callbacks:

- hw_params (EBUSY)
- prepare (EINVAL)
- trigger (EINVAL)

Playback test

This driver can be also used for the playback functionality testing - every time you write the playback data to the 'pcmtest' PCM device and close it, the driver checks the buffer for containing the looped pattern (which is specified in the fill_pattern debugfs file for each channel). If the playback buffer content represents the looped pattern, 'pc_test' debugfs entry is set into '1'. Otherwise, the driver sets it to '0'.

ioctl redefinition test

The driver redefines the 'reset' ioctl, which is default for all PCM devices. To test this functionality, we can trigger the reset ioctl and check the 'ioctl_test' debugfs entry:

```
cat /sys/kernel/debug/pcmtest/ioctl_test
```

If the ioctl is triggered successfully, this file will contain '1', and '0' otherwise.

U