[Projet logiciel en C] API sujet Vorbis

1.0

Généré par Doxygen 1.8.6

Lundi 2 Juin 2014 08 :25 :21

Table des matières

1 Projet logiciel en C : sujet décodeur Vorbis

Date

Printemps 2012

Cette documentation (API : Application Programming Interface) fournit :
- le format d'utilisation du décodeur Vorbis à produire (module Main)
- les spécifications des différents modules à implanter : structures de données et fonctions.

2 Index des modules

2.1 Modules

Liste de tous les modules :

3 Index des structures de données

3.1 Structures de données

Liste des structures de données avec une brève description :

4 Documentation des modules

4.1 Codebook

Structures de données

- struct codebook
- struct codebook_setup

Définitions de type

- typedef struct codebook codebook_t
- typedef struct codebook_setup codebook_setup_t

Fonctions

- status_t codebook_setup_init (vorbis_stream_t *stream, codebook_setup_t **pset)
- void codebooks_free (codebook_setup_t *cb_desc)
- uint32_t codebook_translate_scalar (vorbis_stream_t *stream, codebook_t *book, uint32_t *scalar)
- uint32_t codebook_translate_vq (vorbis_stream_t *stream, codebook_t *book, int *sz, sample_t **vector)
- uint32_t codebook_get_dimension (codebook_t *book)

4.1.1 Description détaillée

En-tête commune des modules de gestion des codebooks : `codebook` et `codebook_read`. Ces deux modules vous faciliteront le déverminage. Il est encouragé de fusionner ces deux modules dans votre version finale.

4.1.2 Documentation des structures de données

4.1.2.1 struct codebook

Structure de description d'un codebook.

Champs de donnée

uint32_t	index	Rang

4.1.2.2 struct codebook_setup

La structure globale du module Codebook qui comporte l'ensemble des codebooks du flux.

Graphe de collaboration de codebook_setup :

Champs de donnée

codebook_t **	codebooks	tableau de pointeurs des codebooks
uint8_t	nb_cb	Nombre de codebooks

4.1.3 Documentation des définitions de type

4.1.3.1 typedef struct codebook_setup codebook_setup_t

4.1.3.2 typedef struct codebook codebook_t

4.1.4 Documentation des fonctions

4.1.4.1 uint32_t codebook_get_dimension (codebook_t * *book*)

Fonction du module `codebook` qui permet de récuperer la dimension des vecteurs de quantification d'un codebook *book*

Paramètres

in	*book*	est le pointeur sur le codebook à utiliser

Renvoie

la dimension des vecteurs de quantification du codebook.

Références codebook_int : :dimensions.

4.1.4.2 status_t codebook_setup_init (vorbis_stream_t * *stream*, codebook_setup_t ** *pset*)

Fonction du module `codebook` qui récupère les dictionnaires stockés dans le flux Vorbis *stream* (cf module `vorbis_io`), pour les stocker dans la structure des dictionnaires *pset*. La fonction stocke chacun des dictionnaires ainsi construit dans le tableau contenu dans la structure (en utilise les algorithmes présentés dans la section Headers3 Codebook de la documentation). Les arbres de Huffman qui représentent les dictionnaires sont stockés en utilisant la structure d'arbre `codebook_tree_t`

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en cours de décodage
out	*pset*	est un pointeur qui renvoie l'adresse de la structure nouvellement allouée et contenant la configuration lue

Renvoie

VBS_SUCCESS en cas de succes, VBS_BADSTREAM en cas d'erreur de lecture dans le flux ou VBS_OUTOFMEMORY en cas d'erreur d'allocation mémoire

Références codebook_int : :base, codebook_int : :cb_root, codebook_setup : :codebooks, cb_param : :codeword_lengths, codebook_int : :dimensions, cb_param : :dimensions, codebook_int : :entries, cb_param : :entries, index, cb_param : :index, vorbis_stream : :io_desc, cb_param : :multiplicands, codebook_setup : :nb_cb, et vorbis_io : :readbits.

Voici le graphe des appelants de cette fonction :

4.1.4.3 uint32_t codebook_translate_scalar (vorbis_stream_t * *stream*, codebook_t * *book*, uint32_t * *scalar*)

Fonction du module `codebook` qui permet d'utiliser les dictionnaires sur le flux Vorbis. *Elle* travaille avec le dictionnaire *book*, et lit le flux `stream` jusqu'à trouver un mot de code valide pour ce dictionnaire. Une fois trouvé, elle renseigne *scalar* avec l'entrée dans l'arbre de huffman correspondante au mot de code.

Paramètres

in	*stream*	est le flux en cours de décodage
in	*book*	est le pointeur sur le codebook à utiliser
out	*scalar*	est un pointeur permettant de renvoyer le pointeur sur l'entrée résultante

Renvoie

le nombre de bits lus dans le flux Vorbis en cas de succès, VBS_BADSTREAM en cas d'erreur de lecture dans le flux.

Références codebook_entry : :entry.

Voici le graphe des appelants de cette fonction :

4.1.4.4 uint32_t codebook_translate_vq (vorbis_stream_t * *stream*, codebook_t * *book*, int * *sz*, sample_t ** *vector*)

Fonction du module `codebook` qui permet d'utiliser les dictionnaires sur le flux Vorbis. *Elle* travaille avec le dictionnaire *book*, et lit le flux `stream` jusqu'à trouver un mot de code valide pour ce dictionnaire. Une fois trouvé, elle renvoie dans *vector* le vecteur des données quantifiées (VQ) correspondant au mot de code. Attention l'allocation et la libération de ce vecteur sont à la charge du module codebook.

Paramètres

in	*stream*	est le flux en cours de décodage
in	*book*	est le pointeur sur le codebook à utiliser
out	*sz*	est un pointeur permettant de renvoyer la taille du vecteur
out	*vector*	est un pointeur sur permettant de renvoyer un pointeur sur le vecteur de donnée

Renvoie

le nombre de bits lus dans le flux Vorbis en cas de succès, VBS_BADSTREAM en cas d'erreur de lecture dans le flux.

Références vq_vect : :dim, vq_vect : :vector, et codebook_entry : :vq.

4.1.4.5 void codebooks_free (codebook_setup_t * *cb_desc*)

Fonction du module `codebook` qui permet de libérer la structure pointée par *cb_desc*. En sortie de fonction, le descripteur des dictionnaires doit être complètement libéré, c'est à dire que le descripteur est libéré ainsi que l'ensemble des structures allouées dans le descripteur.

Paramètres

cb_desc	est le pointeur sur la structure à libérer

Références codebook_setup : :codebooks, et codebook_setup : :nb_cb.

Voici le graphe des appelants de cette fonction :

4.2 Dot_product

Fonctions

— status_t dot_product (sample_t ∗∗fsamp, sample_t ∗∗residues, uint32_t nb_chan, uint32_t n2)

4.2.1 Description détaillée

Module de gestion des dot_product.

4.2.2 Documentation des fonctions

4.2.2.1 status_t dot_product (sample_t ∗∗ *fsamp,* sample_t ∗∗ *residues,* uint32_t *nb_chan,* uint32_t *n2*)

Cette fonction effectue l'opération de produit case à case des échantillons en fréquence produits grâce au floor par les échantillons obtenus grâce aux residues. Les deux tableaux (*fsamp* et *residues*) sont de tailles *nb_chan* lignes par *n2* colonnes. Plus précisemment, chaque "tableau" est un pointeur vers un tableau de *nb_chan* pointeurs vers des tableaux de *n2* echantillons (de type sample_t). Attention, cette opération doit être faite en place, le tableau *fsamp* sert à la fois d'entrée et de sortie.

Paramètres

in,out	*fsamp*	est le tableau des échantillons fréquentiels ; en entrée il contient les floors et en sortie le résultat (produit floors par residues)
in	*residues*	est le tableau des échantillons fréquentiels des residues
in	*nb_chan*	est le nombre de canaux
in	*n2*	est le nombre d'échantillons par canal

Renvoie

VBS_SUCCESS en cas de succes ou VBS_FATAL en cas d'erreur

Voici le graphe des appelants de cette fonction :

4.3 Envelope

Structures de données

— struct envelope

Définitions de type

— typedef struct envelope envelope_t

Fonctions

— envelope_t ∗ envelope_init (uint16_t ∗blocksize)
— void envelope_free (envelope_t ∗env)
— status_t envelope_prepare (envelope_t ∗env, sample_t ∗filter)
— uint16_t envelope_overlap_add (envelope_t ∗env, sample_t ∗in, sample_t ∗cache, sample_t ∗out)

4.3.1 Description détaillée

Module de gestion des enveloppes. Une enveloppe correspond à la fonction de fenêtrage *w* d'un paquet Vorbis, soit le filtre appliqué au résultat de l'iMDCT. Sa forme dépend de la taille de la fenêtre courante (petite ou grande) et des tailles des fenêtres précédente et suivante. Le calcul de cette fonction, en fait les pentes à gauche et à droite, est détaillé dans la section 3.4 du sujet.

4.3.2 Documentation des structures de données

4.3.2.1 struct envelope

Structure de travail pour la gestion des enveloppes.

Elle contient en particulier trois tailles de fenêtre : celle de la fenêtre courante dans le flux, mais aussi celle de la fenêtre précédente et celle de la suivante. Ces tailles sont codées comme une valeur entière valant 0 si une fenêtre est petite ou 1 si elle est grande. Il s'agit en fait d'un indice dans le tableau *blocksize* des tailles de fenêtre.

Le champ *initialized* est relatif au démarrage du flux. En effet, la première moitié de la toute première fenêtre du flux doit être jetée. *initialized* doit donc être égal à 0 (faux) au début du décodage, puis valoir 1 (vrai) à partir de la fenêtre suivante.

Cette structure pourra être étendue si vous souhaitez stocker d'autres informations.

Champs de donnée

uint16_t ∗	blocksize	Tableau de 2 élements contenant les tailles de fenêtre du flux courant.
uint8_t	curr_window	Taille de la fenêtre courante (0 ou 1)
int	initialized	Première moitié de fenêtre initialisée ?
uint8_t	next_window	Taille de la fenêtre suivante (0 ou 1)
uint8_t	prev_window	Taille de la fenêtre précédente (0 ou 1)

4.3.3 Documentation des définitions de type

4.3.3.1 typedef struct envelope envelope_t

4.3.4 Documentation des fonctions

4.3.4.1 void envelope_free (envelope_t ∗ *env*)

Fonction qui permet de libérer la mémoire de la structure d'enveloppe.

Paramètres

in	*env*	est un pointeur sur la zone mémoire à libérer.

Références envelope_int : :slopes.

Voici le graphe des appelants de cette fonction :

4.3.4.2 envelope_t* envelope_init (uint16_t * *blocksize*)

Fonction permettant d'allouer et d'initialiser une enveloppe.

Paramètres

in	*blocksize*	est un tableau contenant les tailles des deux types de fenêtres.

Renvoie

un pointeur sur la structure nouvellement allouée et initialisée ou `NULL` en cas de problème d'allocation.

Références envelope_int : :base, blocksize, curr_window, initialized, next_window, prev_window, et envelope_int- : :slopes.

Voici le graphe des appelants de cette fonction :

4.3.4.3 uint16_t envelope_overlap_add (envelope_t * *env,* sample_t * *in,* sample_t * *cache,* sample_t * *out*)

Cette fonction réalise l'opération de recouvrement entre la fenêtre précédente et la fenêtre courante. Il s'agit simplement d'ajouter les signaux temporels se recouvrant, pour générer un ensemble d'échantillons temporels complètement reconstruits (voir section 3.4). Le recouvrement s'étend de la moitié de la fenêtre précédente à la moitié de la fenêtre courante. Les données non utilisées d'une fenêtre (sa seconde moitié) devront être conservées dans un cache, pour le recouvrement avec la fenêtre suivante.

En entrée de cette fonction :
- *in* contient les échantillons temporels de la fenêtre courante, décodés à l'aide de l'iMDCT et de l'enveloppe courante ;
- le vecteur *cache* contient la seconde moitié de la fenêtre précédente.

En sortie :
- *out* contient les échantillons complètement produits.
- *cache* contient la seconde moitié de la fenêtre *in*, non utilisée car elle n'était pas dans le recouvrement.

Le champ *initialized* de la structure `envelope_t` est utilisé par cette fonction pour savoir si on est en régime constant ou en initialisation. Lors du traitement de la première fenêtre, le cache ne contient aucune valeur et aucune donnée ne peut être produite. Par contre, le cache devra être rempli pour le prochain appel.

Paramètres

in	*env*	est un pointeur sur la structure d'enveloppe courante
in	*in*	est le vecteur de données d'entrée
in,out	*cache*	est un vecteur de données cachées
out	*out*	est le vecteur de données de sortie

Renvoie

le nombre d'échantillons produits

Références envelope_int : :base, blocksize, curr_window, initialized, envelope_int : :left_size, envelope_int : :left_- window_end, envelope_int : :left_window_start, prev_window, et envelope_int : :window_center.

Voici le graphe des appelants de cette fonction :

4.3.4.4 status_t envelope_prepare (envelope_t * *env,* sample_t * *filter*)

Cette fonction permet de générer le filtre à appliquer au résultat de l'iMDCT. Elle travaille à partir d'une configuration *env* préalablement remplie (les indices des tailles devront être mises à jour pour chaque paquet).

Cette fonction ne lit aucune donnée dans le flux Vorbis. La lecture de toutes les informations nécessaires devra donc être faite avant.

Paramètres

in	*env*	est un pointeur sur la structure d'enveloppe courante
out	*filter*	est un vecteur qui reçoit les coefficient du filtre

Renvoie

VBS_SUCCESS en cas de succes ou VBS_FATAL en cas d'erreur.

Références envelope_int : :base, blocksize, curr_window, envelope_int : :left_size, envelope_int : :left_window_-end, envelope_int : :left_window_start, next_window, prev_window, envelope_int : :right_size, envelope_int : :right-_window_end, envelope_int : :right_window_start, envelope_int : :slopes, et envelope_int : :window_center.

4.4 Floor

Structures de données

- struct floors_setup
- struct floor
- struct floor_data

Définitions de type

- typedef struct floors_setup floors_setup_t
- typedef struct floor_data floor_data_t
- typedef struct floor floor_t

Fonctions

- status_t floors_setup_init (vorbis_stream_t *stream, floors_setup_t **pset)
- void floors_free (floors_setup_t *set)
- status_t floor_decode (vorbis_stream_t *stream, floor_t *floor, sample_t *v, uint16_t v_size)

4.4.1 Description détaillée

Module de gestion des floors.

4.4.2 Documentation des structures de données

4.4.2.1 struct floors_setup

Cette structure rassemble l'ensemble des informations concernant la configuration des différents floors du flux vorbis lu.

Le champ *floor_count* donne donc le nombre de configurations de floor présentes dans le tableau *floors*. Ce tableau est donc constitué de *floor_count* pointeurs sur des structures de type `floor_t`.

Les éléments *data0* et *data1* sont des pointeurs sur les zones de travail partagées de type `floor_data_t`, utilisables par les floors de type respectifs 0 et 1, pour les opérations de décodage des packets audio et la génération de la courbe gros grain. Ce type pourra être étendu (voir ci-dessous la définition de `floor_data`), on utilise donc des pointeurs vers des floor_data_t. Il faut noter qu'un tel usage n'est pas obligatoire, vous pourrez opter pour l'allocation en pile (c'est à dire sans utiliser de pré-allocation), mais ceux qui optent pour l'utilisation de zones de travail partagées (plus compliquée, mais plus propre et plus efficace) ont la possibilité de le faire.

Graphe de collaboration de floors_setup :

Champs de donnée

floor_data_t *	data0	Données de travail pour les floor de type 0
floor_data_t *	data1	Données de travail pour les floor de type 1
uint8_t	floor_count	Nombre de floor
floor_t **	floors	Tableau des descriptions de floor

4.4.2.2 struct floor

Cette structure contient les informations basiques génériques (indépendantes du type de floor).

On y retrouve un type (*type*) et un identifiant (*id*) correspondant à l'indice où ce floor est référencé dans la configuration floors_setup.

Les deux derniers champs sont des pointeurs de fonctions :

- *decode* effectue le décodage d'un paquet audio et la synthèse de la courbe à gros grain. Cette fonction utilise un paramètre *stream* pour réaliser les opérations de lecture dans le paquet, un paramètre *floor_cfg*

correspondant à la configuration du floor courant et enfin un vecteur de sortie *v* et le nombre de valeurs à décoder *v_size* (correspondant à $\frac{N}{2}$).
— l'opération *free* réalise elle la libération de la mémoire de la structure de floor pointée par l'argument *floor*, une fois l'ensemble du décodage du flux terminé.

Ces fonctions *decode* et *free* seront en fait des références aux fonctions adéquates spécifiques à chaque type de floor. Selon *type*, *decode* sera ainsi l'adresse de la fonction *floor_type0_decode* ou celle de *floor_type1_decode*. Ces fonctions devront être définies de manière static dans `floor0.c` et `floor1.c`. Vous pouvez étudier le fichier `floor1_example.c` qui est fourni pour vous guider.

Champs de données

— int type
— uint8_t id
— status_t(∗ decode)(vorbis_stream_t ∗stream, floor_t ∗floor_cfg, sample_t ∗v, uint16_t v_size)
— void(∗ free)(floor_t ∗floor)

Documentation des champs

4.4.2.2.1 status_t(∗ floor : :decode)(vorbis_stream_t ∗stream, floor_t ∗floor_cfg, sample_t ∗v, uint16_t v_size)

Operation de décodage

4.4.2.2.2 void(∗ floor : :free)(floor_t ∗floor)

Opération de libération

4.4.2.2.3 uint8_t floor : :id

Identifiant de ce floor

4.4.2.2.4 int floor : :type

Type de floor

4.4.2.3 struct floor_data

Cette structure sert de base pour les structures de travail des différents types de floor.

En effet, chaque type de floor requiert des éléments différents lors de sa manipulation, et l'allocation de ces éléments peut vite se révéler compliquée. Cette structure permet d'allouer les éléments une bonne fois pour toutes pour chaque type, indépendamment du nombre de configurations de floor. On y retrouve juste un identifiant de type (*type*) et un nombre d'occurrences (*occ*). Ce dernier champ permet de recenser le nombre de floor du type correspondant (i.e. le nombre de configurations qui utilisent ce type de floor). On imaginera aisément que dans le cas où cette valeur est nulle, aucune allocation de zone de travail supplémentaire n'est nécessaire. Ce type pourra être étendu de manière spécifique pour chaque type de floor (voir floor1.h et floor1_example.c).

Champs de donnée

int	occ	Nombre d'occurences de ce type
int	type	Type de floor

4.4.3 Documentation des définitions de type

4.4.3.1 typedef struct floor_data floor_data_t

4.4.3.2 typedef struct floor floor_t

4.4.3.3 typedef struct floors_setup floors_setup_t

4.4.4 Documentation des fonctions

4.4.4.1 status_t floor_decode (vorbis_stream_t * *stream,* floor_t * *floor,* sample_t * *v,* uint16_t *v_size*)

Cette fonction est responsable du décodage de la partie floor des paquets audio Vorbis. Elle fera appel à la fonction spécifique au type de floor courant grâce au pointeur de fonction *decode* de la structure `floor_t`.

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en cours de décodage.
in	*floor*	est le floor à utiliser pour le paquet courant
out	*v*	est le vecteur des données produites
in	*v_size*	est la taille du vecteur de données

Renvoie

VBS_SUCCESS en cas de succes, VBS_UNUSED dans le cas où le floor n'est pas utilisé, VBS_BADSTREAM en cas d'erreur de lecture dans le flux ou d'erreur avec `read_nbits`.

Références decode.

4.4.4.2 void floors_free (**floors_setup_t** ∗ *set*)

Cette fonction permet de libérer l'ensemble des structures allouées à l'initialisation du *set*. Chaque floor_t sera libéré grâce à son pointeur de fonction (qui est spécifique de son type).

Paramètres

set	est un pointeur sur la structure à libérer.

Références floors_setup : :data0, floors_setup : :data1, floors_setup : :floor_count, floor_type1_data_free(), floors_setup : :floors, et free.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.4.4.3 status_t floors_setup_init (**vorbis_stream_t** ∗ *stream,* **floors_setup_t** ∗∗ *pset*)

Cette fonction permet d'initialiser un `floors_setup_t` par la lecture de la section correspondante dans l'en-tête 3 du flux Vorbis (voir le paragraphe 5.2.3.c du sujet). La structure `floors_setup_t` est allouée et initialisée par cette fonction, et un pointeur vers cette structure sera renvoyé dans ∗*pset*. Le premier argument *stream* permet de lire le flux (via stream->io_desc) et donne accès au vecteur des blocksize utilisé. La configuration des différents floor ainsi que la creation et l'allocation des structures `floor_data_t` seront réalisés par les fonctions appropriées spécifiques de chaque type de floor (voir floor0.h et floor1.h).

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en cours de lecture.
out	*pset*	est un pointeur pour le retour de l'adresse de la structure allouée et initialisée dans cette fonction.

Renvoie

VBS_SUCCESS en cas de succes, VBS_OUTOFMEMORY en cas d'erreur d'allocation ou VBS_BADSTREAM en cas d'erreur de lecture dans le flux

Références vorbis_codec : :blocksize, vorbis_stream : :codec, floors_setup : :data0, floors_setup : :data1, floors_setup : :floor_count, FLOOR_TYPE1, floor_type1_data_allocate(), floor_type1_data_new(), floor_type1_hdr_decode(), et floors_setup : :floors.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.5 Floor1

Macros

— #define FLOOR_TYPE1 1

Fonctions

— status_t floor_type1_hdr_decode (vorbis_stream_t ∗stream, uint8_t id, floor_t ∗∗floor, floor_data_t ∗data)
— status_t floor_type1_decode (vorbis_stream_t ∗stream, floor_t ∗floor_cfg, sample_t ∗v, uint16_t v_size)
— status_t floor_type1_data_new (floor_data_t ∗∗pfl_data, uint16_t ∗blocksize)
— status_t floor_type1_data_allocate (floor_data_t ∗fl_data)
— void floor_type1_data_free (floor_data_t ∗fl_data)

4.5.1 Description détaillée

Module de gestion des floor de type 1. Comme illustré dans le fichier floor1_example.c , nous vous conseillons d'étendre les types génériques floor_t et floor_data_t pour contenir en plus respectivement l'ensemble des informations de configurations spécifiques au floor de type 1, et l'ensemble des zones de travail utiles aux floors de type 1 pour décoder un paquet audio et faire la synthèse de la courbe gros grain.

4.5.2 Documentation des macros

4.5.2.1 #define FLOOR_TYPE1 1

4.5.3 Documentation des fonctions

4.5.3.1 status_t floor_type1_data_allocate (floor_data_t ∗ *fl_data*)

Fonction d'allocation des tampons internes de la zone de travail A utiliser évidemment après avoir décodé l'ensemble des configurations de floor présentes dans l'entête 3 du flux.

Paramètres

in	*fl_data*	est le pointeur sur la zone de travail

Renvoie

VBS_SUCCESS en cas de succès ou VBS_OUTOFMEMORY en cas d'erreur d'allocation

Références floor_type1_data : :blocksize, floor_type1_data : :dummy, floor_type1_data : :floor_vec, floor_type1_data : :max_points, floor_type1_data : :step2_flag, floor_type1_data : :Y_final, et floor_type1_data : :Y_list.

Voici le graphe des appelants de cette fonction :

4.5.3.2 void floor_type1_data_free (floor_data_t ∗ *fl_data*)

Fonction de libération de la zone de travail. Cette fonction désalloue la zone de travail et ses tampons interne. Elle doit être appelée après la lecture de l'ensemble des description de floor type 1.

Paramètres

in	*fl_data*	est le pointeur sur la zone de travail

Références floor_type1_data : :dummy, floor_type1_data : :floor_vec, floor : :free, floor_type1_data : :step2_flag, floor_type1_data : :Y_final, et floor_type1_data : :Y_list.

Voici le graphe des appelants de cette fonction :

4.5.3.3 status_t floor_type1_data_new (floor_data_t ∗∗ *pfl_data,* uint16_t ∗ *blocksize*)

Fonction d'allocation d'une zone de travail spécifique à ce type de floor.

Paramètres

out	*pfl_data*	est un pointeur sur la zone nouvellement allouée
in	*blocksize*	est le vecteur des tailles de bloc

Renvoie

VBS_SUCCESS en cas de succès ou VBS_OUTOFMEMORY en cas d'erreur d'allocation

Références floor_type1_data : :base, floor_type1_data : :blocksize, FLOOR_TYPE1, et floor_data : :type.

Voici le graphe des appelants de cette fonction :

4.5.3.4 status_t floor_type1_decode (vorbis_stream_t * *stream,* floor_t * *floor_cfg,* sample_t * *v,* uint16_t *v_size*)

Fonction qui permet le décodage d'un floor dans le flux Vorbis. (c.f. section Floor/décodage du paquet et suivantes)

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en coues de décodage
in	*floor_cfg*	est un pointeur sur le flux à utiliser pour la lecture
out	*v*	est le vecteur des données de sortie
in	*v_size*	est la taille du vecteur de sortie

Renvoie

VBS_SUCCESS en cas de succès, VBS_UNUSED si le floor n'est pas utilisé pour ce paquet, VBS_BADSTREAM en cas d'erreur de lecture dans le flux
— erreurs issues de vorbis_read_nbits

Références floor_type1 : :class_dimension, floor_type1 : :class_masterbooks, floor_type1 : :class_subclasses, codebook_translate_scalar(), floor_type1_data : :crt_values, floor_type1 : :data, ilog(), floor_type1 : :partitions, floor_type1 : :range, floor_type1 : :subclass_books, floor_type1 : :values, et floor_type1_data : :Y_list.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.5.3.5 status_t floor_type1_hdr_decode (vorbis_stream_t * *stream,* uint8_t *id,* floor_t ** *floor,* floor_data_t * *data*)

Fonction qui lit les informations de configuration d'un floor dans l'en-tête 3 (c.f. section Floor/décodage d'entête)

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en cours de décodage
in	*id*	est l'identifiant de la configuration de floor qui va être lue
out	*floor*	est un pointeur sur la structure allouée et initialisée
in	*data*	est la zone de travail du type de floor correspondant

Renvoie

VBS_SUCCESS en cas de succès, VBS_BADSTREAM en cas d'erreur de lecture dans le flux, VBS_OUTOFMEMORY en cas d'erreur d'allocation,
— erreurs issues de vorbis_read_nbits

Références floor_type1 : :base, floor_type1_data : :base, floor_type1 : :class_dimension, floor_type1 : :class_masterbooks, floor_type1 : :class_subclasses, floor_type1 : :data, floor : :decode, FLOOR_TYPE1, floor_type1_decode(), floor : :free, floor : :id, floor_type1_data : :max_points, floor_type1 : :maximum_class, floor_type1 : :multiplier, floor_data : :occ, floor_type1 : :partition_class_list, floor_type1 : :partitions, floor_type1 : :range, floor_type1 : :rangebits, floor_type1 : :sorted_idx, floor_type1 : :subclass_books, floor : :type, floor_type1 : :values, et floor_type1 : :X_list.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.6 Helpers

Fonctions

— uint32_t ilog (int32_t val)
— uint32_t lookup1_values (uint32_t a, uint32_t b)
— sample_t float32_unpack (uint32_t packed)

4.6.1 Description détaillée

Boîte à outils de notre décodeur. Elle contient des fonctions utilisées transversalement par plusieurs modules.

4.6.2 Documentation des fonctions

4.6.2.1 sample_t float32_unpack (uint32_t *packed*)

Fonction effectuant la conversion d'un nombre réel, depuis sa représentation entière dans le flux Vorbis vers un format flottant compris par la machine. L'algorithme est donné dans le sujet, dans la partie décrivant les "codebook" de la section "Header 3" (section 5.2.3.a).

Paramètres

in	*packed*	est le réel à convertir

Renvoie

le réel au format de la machine

4.6.2.2 uint32_t ilog (int32_t *val*)

Fonction retournant la position (commençant à 1) du bit de poids fort de l'entier passé en paramètre. Ceci correspond aussi au nombre de bits nécessaires au codage binaire de la valeur val.

Paramètres

in	*val*	est un entier quelconque

Renvoie

la position du bit de poids fort de l'entier passé en paramètre, 0 si val est négatif ou nul.

Voici le graphe des appelants de cette fonction :

4.6.2.3 uint32_t lookup1_values (uint32_t *a*, uint32_t *b*)

Fonction renvoyant le plus grand entier r tel que r à la puissance b est inférieur ou égal à a. Cette fonction est mentionnée dans la section 5.2.3.a du sujet.

Paramètres

in	*a*	est l'entier a
in	*b*	est l'entier b

Renvoie

la valeur de r

4.7 Main

Fonctions

— int main (int argc, char **argv)

4.7.1 Description détaillée

Le module *main* ne contient que la fonction principale de lancement du décodeur Vorbis.

4.7.2 Documentation des fonctions

4.7.2.1 int main (int *argc,* char ** *argv*)

Point d'entrée du programme principal.

Le programme prend un argument principal sur la ligne de commande qui est le nom du fichier ogg/vorbis d'entrée à décoder. Le programme essaye de décoder le premier flux logique du premier flux physique du fichier (appelé ici *chemin/nom*) et stocke le résultat dans un fichier au format WAV nommé *nom.wav* dans le répertoire courant.

Le programme accepte des options modidiant son comportement :
- `-h` : affiche l'aide
- `-l` : ne décode pas de flux logique, liste à la place les flux logiques du premier flux physique et affiche pour chacun son *codec* et son *id*.
- `-s <id>` : spécifie l' *id* du flux logique à décoder. Celui-ci doit être parmi ceux listés par l'option `-l`.
- `-f <outformat>` : spécifie le format de sortie, parmi les gestionnaires d'échantillons disponibles. Au minimum, votre programme doit supporter le format *"wav"*.
- `-o <outfile>` : spécifie le nom du fichier de sortie.

Pour implanter le traitement des options de la ligne de commande, l'utilisation de la librairie *getopt* est conseillée. Une documention et des exemples sont dans la page de man (`man 3 getopt` pour obtenir la page de manuel de la librairie c).

Attention

Il est très important que votre programme s'utilise exactement suivant ces spécifications. Ceci permettra notamment l'automatisation de tests de votre décodeur.

Paramètres

`in`	*argc*	(*argument count*) indique le nombre d'arguments entrés dans la ligne de commande du lancement du programme.
`in`	*argv*	tableau des *argc* arguments, sous forme de chaînes de caractères. Le premier argument est le nom du programme (`vorbis_decoder` ici).

Renvoie

0 en cas de terminaison normale de l'exécution.

Références _stream_t : :bos, _stream_t : :buffer, ogg_logical_stream : :codec, ogg_page_hdr : :crc, ogg_physical_-stream : :first, ogg_page_hdr : :gran_pos, _stream_t : :header, _stream_t : :id, _stream_t : :ifile, _stream_t : :iname, _comment_t : :length, comment : :length, ogg_page_hdr : :magic, ogg_page_hdr : :nb_segs, ogg_physical_stream-: :nb_streams, _stream_t : :next, ogg_logical_stream : :next, _stream_t : :offset, _stream_t : :ofile, ogg_decode(), ogg_init(), ogg_term(), OGG_VORBIS, _stream_t : :packet_id, _stream_t : :packet_size, _stream_t : :page_id, ogg-_page_hdr : :page_id, pcm_handler_create(), pcm_handler_delete(), _stream_t : :prev, _stream_t : :stream_id, ogg-_page_hdr : :stream_id, ogg_logical_stream : :stream_id, _stream_t : :table, ogg_page_hdr : :type, et ogg_page_-hdr : :version.

Voici le graphe d'appel pour cette fonction :

4.8 Mapping

Structures de données

— struct mappings_setup
— struct mapping

Macros

— #define MAPPING_TYPE0 0

Définitions de type

— typedef struct mappings_setup mappings_setup_t
— typedef struct mapping mapping_t

Fonctions

— status_t mappings_setup_init (vorbis_stream_t ∗stream, mappings_setup_t ∗∗pmap)
— status_t mapping_decode (vorbis_stream_t ∗stream, mapping_t ∗map, vorbis_packet_t ∗data)
— void mappings_free (mappings_setup_t ∗map)

4.8.1 Description détaillée

Module de gestion des Mappings.

4.8.2 Documentation des structures de données

4.8.2.1 struct mappings_setup

Structure regroupant l'ensemble des descripteurs de mapping utilisés dans le flux Vorbis en cours de décodage.

maps est un pointeur vers un tableau de mapping_count pointeurs vers les mappings. Ceci permet l'extension du type mapping_t (en mapping_type0_t actuellement, mais on pourrait faire une extension différente si d'autres types de mapping sont autorisés).

Graphe de collaboration de mappings_setup :

Champs de donnée

uint8_t	mapping_count	Nombre de mappings
mapping_t ∗∗	maps	Tableau des pointeurs vers les mapping

4.8.2.2 struct mapping

Structure générique décrivant le mapping, indépendemment de son type.

Même si la norme Vorbis ne compte qu'un type de mapping à l'heure actuelle (le type 0), la norme reste extensible de ce point de vue, et la représentation doit donc respecter cette possibilité. On y retrouve donc un type (*type*), qui doit actuellement valoir MAPPING_TYPE0, et un identifiant (*id*) correspondant à l'indice où ce mapping est référencé dans la configuration mappings_setup.

Les deux autres champs sont des pointeurs sur les fonctions spécifiques du type du mapping :
— *decode* pour décoder le paquet courant (au regard des informations du mapping courant). Pour plus de détails, voir les commentaires de *mapping_type0_decode* (qui sera la fonction pointée par *decode* puisque les mapping sont forcément de type 0).
— *free* libère la mémoire de la structure de mapping pointée par son argument *map*.

Champs de données

- int type
- uint8_t id
- status_t(∗ decode)(vorbis_stream_t ∗stream, mapping_t ∗map, vorbis_packet_t ∗data)
- void(∗ free)(mapping_t ∗map)

Documentation des champs

4.8.2.2.1 status_t(∗ mapping : :decode)(vorbis_stream_t ∗stream, mapping_t ∗map, vorbis_packet_t ∗data)

4.8.2.2.2 void(∗ mapping : :free)(mapping_t ∗map)

4.8.2.2.3 uint8_t mapping : :id

Identifiant de ce mapping

4.8.2.2.4 int mapping : :type

Type de mapping

4.8.3 Documentation des macros

4.8.3.1 #define MAPPING_TYPE0 0

4.8.4 Documentation des définitions de type

4.8.4.1 typedef struct mapping mapping_t

4.8.4.2 typedef struct mappings_setup mappings_setup_t

4.8.5 Documentation des fonctions

4.8.5.1 status_t mapping_decode (vorbis_stream_t ∗ *stream,* mapping_t ∗ *map,* vorbis_packet_t ∗ *data*)

Cette fonction permet, à partir d'un mapping donné par l'argument *map*, de décoder le flux Vorbis d'un paquet.

Cette fonction est une fonction générique, indépendante du type de mapping. Elle ne fait qu'appeler la bonne fonction de décodage, en fonction du type de *map*. Pour les utilisateurs du module `decode`, elle est une alternative à la fonction pointée par le champ map->decode.

Paramètres

`in`	*stream*	pointeur sur le flux Vorbis en cours de lecture.
`in`	*map*	pointeur vers le mapping courant.
`in,out`	*data*	pointeur vers une structure qui contient les tampons de décodage.

Renvoie

VBS_SUCCESS

Références decode, et id.

4.8.5.2 void mappings_free (mappings_setup_t ∗ *map*)

Cette fonction libère la mémoire allouée pour le stockage des descripteurs de mappings.

Paramètres

in	*map*	est le pointeur sur la structure à libérer.

Références free, mappings_setup : :mapping_count, et mappings_setup : :maps.

Voici le graphe des appelants de cette fonction :

4.8.5.3 status_t mappings_setup_init (**vorbis_stream_t** ∗ *stream,* **mappings_setup_t** ∗∗ *pmap*)

Cette fonction effectue l'allocation de la structure de type mappings_setup_t et l'initialise en lisant les informations de configuration de l'entête 3 (voir la section 5.2.3.e du sujet).

Cette fonction prend en argument un stream qui permettra les opérations de lecture du paquet d'entête, et d'avoir accès aux descripteurs de floor et de residue (via stream->codec->floors_desc et stream->codec->residues_desc, qui doivent donc avoir été préalablement initialisés).

Paramètres

in	*stream*	pointeur sur le flux Vorbis en cours de lecture.
out	*pmap*	adresse du pointeur sur la structure allouée et initialisée.

Renvoie

VBS_SUCCESS en cas de success, VBS_OUTOFMEMORY en cas d'erreur d'allocation ou VBS_BADSTREAM en cas d'erreur de lecture dans le flux

Références mappings_setup : :mapping_count, MAPPING_TYPE0, et mappings_setup : :maps.

Voici le graphe des appelants de cette fonction :

4.9 Mode

Structures de données

— struct window_modes_setup
— struct window_mode

Définitions de type

— typedef struct window_modes_setup window_modes_setup_t
— typedef struct window_mode window_mode_t

Fonctions

— status_t window_modes_setup_init (vorbis_stream_t ∗stream, window_modes_setup_t ∗∗pset)
— void window_modes_free (window_modes_setup_t ∗set)

4.9.1 Description détaillée

Module de gestion des Modes.

4.9.2 Documentation des structures de données

4.9.2.1 struct window_modes_setup

Structure regroupant l'ensemble des descripteurs de mode utilisés dans le flux Vorbis en cours de décodage.

Cette structure contient un tableau de modes *modes* de *mode_count* cases. Cette taille n'étant connue qu'à la lecture du flux, le tableau est alloué dynamiquement.

Le champ *mode_code_nbbits* représente le nombre de bits nécessaires pour coder (dans les paquets audio) l'identifiant du mode utilisé (cf. *ilog*). On parle bien ici de l'indice maximal du mode dans le tableau *modes*, il ne s'agit donc pas de *mode_count*, mais de *mode_count* - 1.

Graphe de collaboration de window_modes_setup :

Champs de donnée

uint8_t	mode_code_nbbits	Nombre de bits nécessaire pour coder l'indice d'un mode
uint8_t	mode_count	Nombre de modes
window_mode_t ∗	modes	Tableau des modes

4.9.2.2 struct window_mode

Structure décrivant le mode de la fenêtre courante.

Le champ *blockflag* définit la taille de la fenêtre : 0 pour une petite fenêtre, 1 pour une grande. Le champ *mapping* donne le mapping du mode courant.

Les deux champs *window_type* et *transform_type* sont réservés pour des extensions futures de Vorbis. Ils doivent donc toujours valoir zéro, sinon le descripteur de mode n'est pas valide.

Graphe de collaboration de window_mode :

Champs de donnée

uint8_t	blockflag	Taille de la fenêtre courante, 0 ou 1
mapping_t *	mapping	Mapping associé à ce mode
uint16_t	transform_type	Extension, doit valoir 0
uint16_t	window_type	Extension, doit valoir 0

4.9.3 Documentation des définitions de type

4.9.3.1 typedef struct window_mode window_mode_t

4.9.3.2 typedef struct window_modes_setup window_modes_setup_t

4.9.4 Documentation des fonctions

4.9.4.1 void window_modes_free (window_modes_setup_t * *set*)

Cette fonction libère la mémoire allouée pour le stockage des descripteurs de modes.

Paramètres

in	*set*	est le pointeur sur la structure à libérer.

Références window_modes_setup : :modes.

Voici le graphe des appelants de cette fonction :

4.9.4.2 status_t window_modes_setup_init (vorbis_stream_t * *stream,* window_modes_setup_t ** *pset*)

Fonction de lecture des modes dans l'entête 3.

Cette fonction est en charge de l'allocation et de l'initialisation de la structure window_modes_setup_t regroupant les différents modes (voir la section 5.2.3.f du sujet).

Remarque : *stream->codec->modes_desc* ne contient pas d'information à l'appel de cette fonction. La structure **pset* sera donc initialisée en lisant dans *stream->io_desc*. Par contre *stream->codec->mappings_desc* doit avoir été préalablement initialisée : cette fonction ne construit pas les mapping associés aux modes, elle se contente de pointer vers les mapping appropriés pré-existants dans *stream->codec->mappings_desc*.

Paramètres

in	*stream*	pointeur sur le flux Vorbis en cours de lecture.
out	*pset*	adresse du pointeur sur la structure allouée et initialisée.

Renvoie

VBS_SUCCESS en cas de success, VBS_OUTOFMEMORY en cas d'erreur d'allocation ou VBS_BADSTREAM en cas d'erreur de lecture dans le flux

Références window_mode : :blockflag, ilog(), window_mode : :mapping, window_modes_setup : :mode_code_nbbits, window_modes_setup : :mode_count, window_modes_setup : :modes, window_mode : :transform_type, et window_mode : :window_type.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.10 Ogg_core

Fichiers

- fichier ogg.h
- fichier ogg_core.h
- fichier ogg_internal.h

Structures de données

- struct ogg_physical_stream
- struct ogg_logical_stream
- struct ogg_page_hdr
- struct internal_ogg_logical_stream

Définitions de type

- typedef enum ogg_codec ogg_codec_t
- typedef struct ogg_physical_stream ogg_physical_stream_t
- typedef struct ogg_logical_stream ogg_logical_stream_t

Énumérations

- enum ogg_codec { OGG_UNKNOWN, OGG_VORBIS }

Fonctions

- ogg_status_t ogg_init (FILE *file, ogg_physical_stream_t **ppstream)
- ogg_status_t ogg_decode (ogg_logical_stream_t *lstream, pcm_handler_t *pcm_hdler)
- ogg_status_t ogg_term (ogg_physical_stream_t *pstream)
- struct ogg_page_hdr __attribute__ ((__packed__))
- ogg_status_t ogg_get_next_page (internal_ogg_logical_stream_t *lstream)

4.10.1 Description détaillée

Module de gestion du container Ogg. Ce module gère essentiellement le désentrelacement des pages des différents flux logiques.

Il regroupe des structures et fonctions :
- publiques, dans `ogg.h`, utilisables dans le projet
- et privées, dans `ogg_internal.h`, destinées à l'interface entre les deux modules ogg_core et ogg_packet.

4.10.2 Documentation des structures de données

4.10.2.1 struct ogg_physical_stream

Structure représentant un flux physique Ogg.

Cette structure contient une liste chaînée des flux logiques présents dans un flux physique. Comme expliqué au chapitre 4 du sujet, les premières pages des flux logiques (1 seule page par flux, celle qui a le flag BOS) sont les unes à la suite des autres au début du fichier Ogg. Il faut donc toutes les lire pour générer la liste.

Graphe de collaboration de ogg_physical_stream :

Champs de donnée

ogg_logical_stream_t *	first	Pointeur vers le premier des flux logiques (tête de liste chaînée).
uint32_t	nb_streams	Nombre de flux logiques contenus dans le flux physique.

4.10.2.2 struct ogg_logical_stream

Structure représentant un flux logique Ogg.

Graphe de collaboration de ogg_logical_stream :

Champs de donnée

ogg_codec_t	codec	Format/codec du flux logique
ogg_logical_stream_t *	next	Pointeur vers le prochain élément de la liste des flux logiques, NULL si c'est le dernier.
uint32_t	stream_id	Identifiant du flux logique au sein du flux physique.

4.10.2.3 struct ogg_page_hdr

En-tête de page Ogg, sans sa table des segments.

Sa définition (ordre et taille des champs, voir section 4.4.1 du sujet) correspond exactement à l'en-tête présent dans le fichier. Ceci permet, dans le cas où l'architecture utilisée est little-endian, une copie directe du contenu du fichier vers cette structure. Dans le cas contraire big-endian, il faut en plus intervertir les octets des champs multi-octets.

Champs de donnée

uint32_t	crc	
int64_t	gran_pos	
uint8_t	magic[4]	
uint8_t	nb_segs	
uint32_t	page_id	
uint32_t	stream_id	
uint8_t	type	
uint8_t	version	

4.10.2.4 struct internal_ogg_logical_stream

Represention interne d'un flux logique.

Cette structure est une extension de la structure ogg_logical_stream. Elle contient des champs supplémentaires nécessaires pour la lecture du flux mais qui n'ont pas besoin d'être vus par les autres modules.

Les trois champs *header*, *table* et *data* doivent être cohérents entre eux : *table* et *data* doivent être de même taille, et valent NULL si la page courante ne contient aucun segment.

Graphe de collaboration de internal_ogg_logical_stream :

Champs de donnée

ogg_logical_stream_t	base	Partie publique de cette structure
uint8_t *	data	Pointeur vers les segments de donnée de la page.
ogg_page_hdr_t *	header	Pointeur vers l'entête (sans la page des segments) du flux logique.

ogg_packet_handler_t *	packet	Pointeur vers une structure de gestion de packet
ogg_physical_stream_t *	phy	Flux physique auquel appartient le flux logique
uint8_t *	table	Table des segments : un tableau contenant la taille de chacun des segments de la page.

4.10.3 Documentation des définitions de type

4.10.3.1 typedef enum ogg_codec ogg_codec_t

4.10.3.2 typedef struct ogg_logical_stream ogg_logical_stream_t

4.10.3.3 typedef struct ogg_physical_stream ogg_physical_stream_t

4.10.4 Documentation du type de l'énumération

4.10.4.1 enum ogg_codec

Identifiant de format/codec d'un flux logique Ogg.

Valeurs énumérées

OGG_UNKNOWN

OGG_VORBIS

4.10.5 Documentation des fonctions

4.10.5.1 struct ogg_page_hdr __attribute__ ((__packed__))

4.10.5.2 ogg_status_t ogg_decode (ogg_logical_stream_t * *lstream,* pcm_handler_t * *pcm_hdler*)

Decode un flux logique au format Vorbis.

Le flux logique à décoder doit faire partie de la liste des flux logiques du flux physique, construite par la fonction ogg_init. (ou n'est pas appelée, c'est à la discrétion du module main).

Le décodage proprement dit est délégué à la fonction decode_stream du module vorbis_main. Celle-ci doit pouvoir faire appel à ogg_packet_read et récupérer ainsi les premiers octets du premier paquet du flux logique. Pour celà, un ogg_packet_handler aura été préalablement attaché au flux *lstream*. Il sera détaché une fois le décodage terminé. Ces opérations d'attacher/détacher sont réalisées par le module ogg_packet.

Note

Cette fonction ne peut être appelée qu'une seule fois par physical_stream. Dans ce projet, il n'est en effet pas possible de décoder plusieurs flux logiques d'un même flux physique.
Elle peut aussi ne jamais être appelée, c'est à la discrétion du module main.

Paramètres

in	*lstream*	flux logique à décoder
in	*pcm_hdler*	module de traitement des échantillons PCM décodés dans le flux Vorbis.

Renvoie

OGG_OK en cas de succès, OGG_ERR_ILL_OP si la fonction a déjà été appelée sur le flux *lstream*, un statut d'erreur adéquat sinon.

Références internal_ogg_logical_stream : :base, ogg_logical_stream : :codec, decode_stream(), ogg_packet_attach(), ogg_packet_detach(), OGG_UNKNOWN, internal_ogg_logical_stream : :phy, et _phy_stream_t : :todec.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.10.5.3 ogg_status_t ogg_get_next_page (internal_ogg_logical_stream_t * *lstream*)

Fonction permettant d'avancer à la page suivante d'un flux logique.

Seuls les champs *header*, *table*, *data* du flux sont modifiés. S'il n'y a plus de page dans le flux logique, la fonction retourne OGG_END et mets les 3 champs à NULL.

Paramètres

lstream	le flux logique en cours de décodage

Renvoie

OGG_OK en cas de succès, OGG_END si le flux était déjà sur la dernière page, un statut d'erreur adéquat sinon.

Références internal_ogg_logical_stream : :base, _phy_stream_t : :buffer, _phy_stream_t : :buffer_crc, ogg_page_hdr : :crc, internal_ogg_logical_stream : :data, _phy_stream_t : :file, internal_ogg_logical_stream : :header, ogg_page_hdr : :nb_segs, internal_ogg_logical_stream : :phy, ogg_page_hdr : :stream_id, ogg_logical_stream : :stream_id, internal_ogg_logical_stream : :table, et _phy_stream_t : :todec.

Voici le graphe des appelants de cette fonction :

4.10.5.4 ogg_status_t ogg_init (FILE * *file,* ogg_physical_stream_t ** *ppstream*)

Alloue et initialise un flux physique à partir d'un fichier, en remplissant la liste des flux logiques qu'il contient.

Comme expliqué au chapitre 4 du sujet, les premières pages des flux logiques (1 seule page par flux, celle qui a le flag BOS) sont les unes à la suite des autres au début du fichier Ogg. Cette fonction lit donc les premières pages afin de lister tous les flux logiques que contient le flux physique.

Paramètres

in	*file*	descripteur de fichier ouvert en lecture
out	*ppstream*	pointeur pour le retour de l'adresse de la structure allouée et initialisée dans cette fonction.

Renvoie

OGG_OK en cas de succès, OGG_END si le flux physique ne contient aucun flux logique, un statut d'erreur adéquat sinon.

Références internal_ogg_logical_stream : :base, _phy_stream_t : :buffer, _phy_stream_t : :buffer_crc, ogg_logical_stream : :codec, ogg_page_hdr : :crc, internal_ogg_logical_stream : :data, _phy_stream_t : :file, ogg_physical_stream : :first, internal_ogg_logical_stream : :header, ogg_page_hdr : :nb_segs, ogg_physical_stream : :nb_streams, ogg_logical_stream : :next, internal_ogg_logical_stream : :packet, internal_ogg_logical_stream : :phy, _phy_stream_t : :public, ogg_page_hdr : :stream_id, ogg_logical_stream : :stream_id, internal_ogg_logical_stream : :table, _phy_stream_t : :todec, et ogg_page_hdr : :version.

Voici le graphe des appelants de cette fonction :

4.10.5.5 ogg_status_t ogg_term (ogg_physical_stream_t * *pstream*)

Libérer la mémoire d'une structure de flux physique Ogg.

Cette fonction peut être appelée directement après l'appel à la fonction ogg_decode, ou directement après ogg_init si on ne veut décoder aucun flux.

Paramètres

in	*pstream*	pointeur sur la zone mémoire à libérer.

Renvoie

OGG_OK en cas de succès, un statut d'erreur adéquat sinon.

Références _phy_stream_t : :buffer.

Voici le graphe des appelants de cette fonction :

4.11 Ogg_packet

Fichiers

— fichier ogg_packet.h

Structures de données

— struct ogg_packet_handler

Fonctions

— ogg_status_t ogg_packet_attach (internal_ogg_logical_stream_t *lstream)
— ogg_status_t ogg_packet_detach (internal_ogg_logical_stream_t *lstream)
— ogg_status_t ogg_packet_read (ogg_logical_stream_t *lstream, uint8_t *buf, uint32_t nbytes, uint32_t *nbytes_read)
— ogg_status_t ogg_packet_next (ogg_logical_stream_t *lstream)
— uint32_t ogg_packet_size (ogg_logical_stream_t *lstream)
— ogg_status_t ogg_packet_position (ogg_logical_stream_t *lstream, int64_t *position)

Variables

— struct ogg_packet_handler __attribute__

4.11.1 Description détaillée

Module de gestion des paquets Ogg. Ce module permet la reconstitution des paquets d'un flux logique à partir de ses pages. Il contient des fonctions utilisables par le module ogg_core et d'autres par les modules Vorbis.

Toutes les fonctions travaillent à partir d'un objet ogg_logical_stream, qui est en réalité du type étendu internal_ogg_logical_stream

4.11.2 Documentation des structures de données

4.11.2.1 struct ogg_packet_handler

Structure interne de gestion d'un packet Ogg.

Cette structure est simplement une structure "de typage" pour le champ *packet* d'un internal_ogg_logical_stream.

Conceptuellement, elle devrait être vide et ne contenir aucun champ. Cependant, les structures vides sont inscrites dans la norme C comme étant "compiler dependant" (pour des raisons de taille de type principalement). GCC les gère bien, mais ce n'est pas le cas de tous les compilateurs. La structure est donc ici définie avec un champ inutile, le plus petit possible.

Naturellement vous devez étendre cette structure en y ajoutant les informations nécessaires à votre implémentation des modules ogg.

Champs de donnée

uint8_t	dummy	Champ inutile, pour respecter la norme C sur les structures vides.

4.11.3 Documentation des fonctions

4.11.3.1 ogg_status_t ogg_packet_attach (internal_ogg_logical_stream_t * *lstream*)

Fonction permettant d'associer un gestionaire de paquet ogg_packet_handler à un flux logique.

Elle doit :

— créer un ogg_packet_handler puis l'initialiser de manière à ce que le paquet courant soit le premier paquet du flux logique. Un appel à la fonction ogg_packet_read doit renvoyer les premiers octets du premier paquet.
— attacher cet objet au flux logique passé en paramètre, via son champs *packet* (*lstream* est en réalité du type étendu internal_ogg_logical_stream), initialement NULL.

Lors de l'appel de cette fonction la première page du flux logique doit avoir été mise dans les champs *header*, *table* et *data* du flux.

Paramètres

stream	le flux logique auquel on attache un gestionnaire de paquet.

Renvoie

OGG_OK en cas de succès, un statut d'erreur adéquat sinon.

Références internal_ogg_logical_stream : :header, internal_ogg_logical_stream : :packet, ogg_packet_handler_int- : :page_id, et ogg_page_hdr : :page_id.

Voici le graphe des appelants de cette fonction :

4.11.3.2 ogg_status_t ogg_packet_detach (internal_ogg_logical_stream_t ∗ *lstream*)

Fonction libérant le champ *packet* du flux passé en argument, qui est ensuite mis à NULL.

Paramètres

stream	le flux logique duquel on détache un gestionnaire de paquet.

Renvoie

OGG_OK en cas de succès, un statut d'erreur adéquat sinon.

Références internal_ogg_logical_stream : :packet.

Voici le graphe des appelants de cette fonction :

4.11.3.3 ogg_status_t ogg_packet_next (ogg_logical_stream_t ∗ *lstream*)

Fonction permettant d'avancer au paquet suivant.

Tous les octets éventuellement restant dans le paquet courant sont sautés. L'appel suivant de ogg_packet_read copiera les premiers octets du paquet nouvellement sélectionné.

Un appel à cette fonction sur le dernier paquet n'est pas une erreur, mais le devient une fois que le dernier paquet a été sauté.

Paramètres

in	*lstream*	flux logique en cours de lecture

Renvoie

OGG_OK en cas de succès, OGG_END si le paquet courant est le dernier paquet du flux, un statut d'erreur adéquat sinon.

Références ogg_packet_handler_int : :byte_id, internal_ogg_logical_stream : :header, ogg_page_hdr : :nb_segs, ogg_get_next_page(), internal_ogg_logical_stream : :packet, ogg_packet_handler_int : :page_id, ogg_page_hdr- : :page_id, ogg_packet_handler_int : :seg_base, ogg_packet_handler_int : :seg_id, et internal_ogg_logical_stream- : :table.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.11.3.4 ogg_status_t ogg_packet_position (ogg_logical_stream_t ∗ *lstream*, int64_t ∗ *position*)

Fonction permettant d'accéder à la position absolue dans le flux logique.

Cette position est définie dans la section 4.4.1 sur les en-tête de page. Si le paquet courant est le dernier de la page et qu'il se termine sur cette page, alors la position rechercher est la valeur *granule position* de l'en-tête. Sinon, la position du paquet courant n'est pas (encore) connue, et vaut -1. Quand le dernier paquet est sauté, la position indiquée doit être celle du dernier paquet.

Paramètres

lstream[in]	flux logique en cours de lecture.
position[out]	la position abosule du paquet courant, -1 si elle ne peut pas être déterminée

Renvoie

OGG_OK en cas de succès, un statut d'erreur adéquat sinon.

Références internal_ogg_logical_stream : :header, ogg_page_hdr : :nb_segs, et internal_ogg_logical_stream : :packet.

Voici le graphe des appelants de cette fonction :

4.11.3.5 ogg_status_t ogg_packet_read (ogg_logical_stream_t * *lstream*, uint8_t * *buf*, uint32_t *nbytes*, uint32_t * *nbytes_read*)

Fonction permettant de lyre des octets du paquet courant d'un flux logique.

Le nombre d'octets lus par cette fonction est normalement égal à *nbytes*, mais il peut être inférieur dans le cas où le paquet ne contient plus assez d'octets. Demander plus d'octets qu'il n'en reste dans le paquet courant n'est pas une erreur. En particulier l'appel de ogg_packet_read sur un paquet entièrement lu (il ne reste plus d'octets dedans), doit simplement retourner 0 puisqu'elle n'a pu lire aucun octet.

Une erreur est par contre d'appeler cette fonction si le dernier paquet du flux a été sauté avec la fonction ogg_packet_next.

Paramètres

in	*lstream*	flux logique en cours de lecture
in	*buf*	buffer de sortie contenant les octets lus, qui doit être déjà alloué avec une capacité au moins égale à nbytes.
in	*nbytes*	nombre de bytes à lire dans lstream
out	*nbytes_read*	le nombre d'octets réellement lus

Renvoie

OGG_OK en cas de succès, un statut d'erreur adéquat sinon.

Références ogg_packet_handler_int : :byte_id, internal_ogg_logical_stream : :data, internal_ogg_logical_stream : :header, ogg_page_hdr : :nb_segs, ogg_get_next_page(), internal_ogg_logical_stream : :packet, ogg_packet_handler_int : :page_id, ogg_page_hdr : :page_id, ogg_packet_handler_int : :pkt_offset, ogg_packet_handler_int : :seg_base, ogg_packet_handler_int : :seg_id, et internal_ogg_logical_stream : :table.

Voici le graphe d'appel pour cette fonction :

4.11.3.6 uint32_t ogg_packet_size (ogg_logical_stream_t * *lstream*)

Fonction estimant la taille en octets du paquet courant.

Cette taille peut être estimée à partir d'informations sur le paquet courant (à rajouter dans votre structure ogg_packet_handler) : par exemple le segment courant et le nombre d'octets déjà lus. Si le paquet est terminé, la fonction doit retourner la taille réelle. Sinon, la valeur retournée doit être strictement suérieure au nombre d'octets lus jusqu'à présent dans le paquet.

Cette fonction est surtout utile pour le débogage, afin de vérifier si le paquet entier a été lu.

Paramètres

in	*lstream*	flux logique en cours de lecture.

Renvoie

taille en octets du paquet courant.

Références internal_ogg_logical_stream : :packet, ogg_packet_handler_int : :pkt_offset, ogg_packet_handler_int- : :seg_id, et internal_ogg_logical_stream : :table.

Voici le graphe des appelants de cette fonction :

4.11.4 Documentation des variables

4.11.4.1 struct ogg_packet_handler __attribute__

4.12 Pcm_handler

Structures de données

— struct pcm_handler

Définitions de type

— typedef struct pcm_handler pcm_handler_t

Fonctions

— void pcm_handler_list (const char ∗prefix)
— pcm_handler_t ∗ pcm_handler_create (const char ∗format, const char ∗arg)
— void pcm_handler_delete (pcm_handler_t ∗hdlr)

4.12.1 Description détaillée

Module de gestion des sorties PCM. Ce module sert à traiter les échantillons audios décodés par Vorbis. Plusieurs sorties sont possibles pour les échantillons : par exemple les stocker dans un fichier, ou les envoyer sur les hauts parleurs. Ce module permet de gérer plusieurs sorties. Notre version en propose 2 :

1. "wav" : met les échantillons dans un fichier au format WAV,
2. "raw" : met les échantillons directement dans un fichier sans traitement.

4.12.2 Documentation des structures de données

4.12.2.1 struct pcm_handler

Cette structure représente un traitant d'échantillons qui est utilisé par les modules vorbis.

Elle contient 3 champs qui sont des pointeurs sur des fonctions qui sont spécifiées ci-dessous.

Cette fonction permet d'initialiser le traitant avec les paramètres du flux audio (fréquence d'échantillonage et nombre de canaux). La taille d'un échantillon n'est pas paramétrable, seuls des échantillons signés sur 16 bits sont utilisés dans ce projet. Cette fonction doit être appelée une seule fois avant tout appel à la fonction *process*.

init prend en argument le traitant d'échantillon, la fréquence d'échantillonage (le nombre d'échantillon qui doivent lus par seconde pour chaque canal audio) et le nombre de canaux audios (1 pour du mono, 2 pour de la stéréo, etc).

init retourne 0 en cas de succès et un nombre négatif en cas d'erreur.

La fonction *process* permet de traiter des échantillons audios. En plus du traitant, elle prend en argument le nombre d'échantillons par canal, ainsi que les échantillons. Cette fonction peut être appelée autant de fois que nécessaire

Les échantillons sont donnés dans un tableau de tableaux. La première dimension est le canal, la deuxième est le numéro de l'échantillon. Les échantillons sont ordonnés par ordre croissant dans le temps.

process retourne 0 en cas de succès et un nombre négatif en cas d'erreur.

La fonction *finalize* permet de finaliser le traitement des échantillons. Une fois appelée, il n'est plus possible de traiter des échantillons avec *process*.

finalisze retourne 0 en cas de succès et un nombre négatif en cas d'erreur.

Champs de données

— int(∗ init)(pcm_handler_t ∗hdlr, unsigned int sampl, unsigned int nchan)
— int(∗ process)(pcm_handler_t ∗hdlr, unsigned int num, int16_t ∗∗samples)
— int(∗ finalize)(pcm_handler_t ∗hdlr)

Documentation des champs

4.12.2.1.1 int(∗ pcm_handler : :finalize)(pcm_handler_t ∗hdlr)

Pointeur sur la fonction finalize correspondante

4.12.2.1.2 int(∗ pcm_handler : :init)(pcm_handler_t ∗hdlr, unsigned int sampl, unsigned int nchan)

Pointeur sur la fonction init correspondante

4.12.2.1.3 int(∗ pcm_handler : :process)(pcm_handler_t ∗hdlr, unsigned int num, int16_t ∗∗samples)

Pointeur sur la fonction process correspondante

4.12.3 Documentation des définitions de type

4.12.3.1 typedef struct pcm_handler pcm_handler_t

4.12.4 Documentation des fonctions

4.12.4.1 pcm_handler_t∗ pcm_handler_create (const char ∗ *format,* const char ∗ *arg*)

Cette fonction alloue et renvoie un `pcm_handler_t` du format *format* demandé. Le deuxième argument est un argument pour le gestionnaire d'échantillon. Les chaînes de caractère *format* correctes sont celles listées par la fonction *pcm_handler_list*. Dans le cas d'un gestionnaire stockant les échantillons dans un fichier (par exemple **"wav"**), c'est le nom de ce fichier qui est mis dans cet argument.

Les trois champs du `pcm_handler_t` renvoyé doivent être valides. Ainsi il sera nécessaire d'avoir aussi implémenté dans ce module les 3 fonctions à mettre dans les champs du `pcm_handler_t` pour pouvoir les initialiser.

Paramètres

in	*format*	est le nom du format souhaité.
in	*arg*	est un argument passé au gestionnaire d'échantillons.

Renvoie

un pointeur vers le `pcm_handler_t` créé en cas de succès ou NULL en cas d'erreur.

Références _pcm_handler_format_t : :create, et _pcm_handler_desc_t : :format.

Voici le graphe des appelants de cette fonction :

4.12.4.2 void pcm_handler_delete (pcm_handler_t ∗ *hdlr*)

Cette fonction supprime toute zone mémoire allouée pour le `pcm_handler_t` dont le pointeur est passé en argument.

Paramètres

in	*hdlr*	pointe la zone à libérer.

Références _pcm_handler_format_t : :delete, et _pcm_handler_desc_t : :format.

Voici le graphe des appelants de cette fonction :

4.12.4.3 void pcm_handler_list (const char ∗ *prefix*)

Cette fonction permet de lister tous les formats de sortie disponibles. Elle affiche sur la sortie standard la liste de ces formats. Chaque nom de format est affiché sur une ligne différente précédé de la chaîne de caractères *prefix* passée en argument. Dans le cas d'un module ne gérant que le format de sortie **"wav"**, un appel à cette fonction avec comme argument la chaîne vide **""** doit uniquement imprimer **"wav\n"** sur la sortie standard.

Paramètres

in	*prefix*	est une chaîne de caractère contenant le préfixe d'affichage.

4.13 Residue

Structures de données

- struct residues_setup
- struct residue

Définitions de type

- typedef struct residues_setup residues_setup_t
- typedef struct residue residue_t

Fonctions

- status_t residues_setup_init (vorbis_stream_t *stream, residues_setup_t **pres)
- void residues_free (residues_setup_t *set)
- status_t residue_decode (vorbis_stream_t *stream, residue_t *residue, int ch, int64_t N2, sample_t **v, uint8_t *do_not_decode)

4.13.1 Description détaillée

Module de gestion des residues.

4.13.2 Documentation des structures de données

4.13.2.1 struct residues_setup

Structure contenant la configuration lue dans l'entête 3.

Cette structure rassemble l'ensemble des définitions de residues du flux vorbis lu, dans le tableau *residues*. Ce tableau est de dimension *residue_count* et est constitué de pointeur sur des structures de type `residue_t`.

Graphe de collaboration de residues_setup :

Champs de donnée

uint8_t	residue_count	Nombre de residues
residue_t **	residues	Tableau des residues

4.13.2.2 struct residue

Structure de description d'un residue.

Cette structure permet d'identifier les différents residues utilisés dans le flux vorbis décodé.

Champs de données

- int type
- status_t(* decode)(vorbis_stream_t *stream, residue_t *resid, uint8_t ch, uint16_t N2, sample_t **v, uint8_t *do_not_decode)
- void(* free)(residue_t *res)

Documentation des champs

4.13.2.2.1 status_t(* residue : :decode)(vorbis_stream_t *stream, residue_t *resid, uint8_t ch, uint16_t N2, sample_t **v, uint8_t *do_not_decode)

Pointeur vers une fonction permettant la lecture du residue dans le paquet audio vorbis.

4.13.2.2.2 void(* residue : :free)(residue_t *res)

Fonction de libération associée Pointeur vers une fonction permettant la libération de la structure courante

4.13.2.2.3 int residue : :type

Type de residue. Dans la norme actuelle, il existe 3 types de residues (0, 1 et 2), mais cette norme est extensible, et ce nombre n'est pas figé. [0-2]

4.13.3 Documentation des définitions de type

4.13.3.1 typedef struct residue residue_t

4.13.3.2 typedef struct residues_setup residues_setup_t

4.13.4 Documentation des fonctions

4.13.4.1 status_t residue_decode (vorbis_stream_t * *stream,* residue_t * *residue,* int *ch,* int64_t *N2,* sample_t ** *v,* uint8_t * *do_not_decode*)

Cette fonction est responsable du décodage dans les paquets audio residues.

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en cours de lecture.
in	*residue*	est un pointeur sur la configuration de residue à utiliser pour obtenir les coefficients permettant de calculer la courbe de grain fin.
in	*ch*	donne le nombre de canaux à décoder (il peut être inférieur au nombre réel de canaux).
in	*N2*	donne la taille en nombre de coefficient de chaque canal (tous les canaux ont la même taille).
out	*v*	correspond aux vecteurs qui permettront de conserver les residues décodés dans le paquet.
in	*do_not_decode*	correspond au vecteur de fanions correspondant aux canaux n'ayant pas de residue à décoder.

Renvoie

VBS_SUCCESS en cas de success, VBS_BADSTREAM en cas d'erreur de lecture dans le flux, ou VBS_EOP en cas de fin de paquet impromptu.

Références decode.

4.13.4.2 void residues_free (residues_setup_t * *set*)

Cette fonction libère la mémoire allouée pour le stockage des configurations de residues.

Paramètres

in	*set*	est le pointeur sur la structure à libérer.

Références free, residues_setup : :residue_count, et residues_setup : :residues.

Voici le graphe des appelants de cette fonction :

4.13.4.3 status_t residues_setup_init (vorbis_stream_t * *stream,* residues_setup_t ** *pres*)

Cette fonction alloue et initialise la structure de type `residues_setup_t`, peuple le champ *residues* et la renvoie par le biais du pointeur de pointeur *pres*. Cette fonction lit la partie de l'entête 3 relative aux residues (c.f. la section Header3/residue de la documentation et la section Residue/décodage). NB : Cette fonction est largement similaire à la fonction `floors_setup_init`.

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en cours de lecture.
out	*pres*	est un pointeur permettant de renvoyer un pointeur sur la zone allouée et initialisée.

Renvoie

VBS_SUCCESS en cas de success, VBS_OUTOFMEMORY en cas d'erreur d'allocation ou VBS_BADSTREAM en cas d'erreur de lecture dans le flux.

Références residues_setup : :residue_count, et residues_setup : :residues.

Voici le graphe des appelants de cette fonction :

4.14 Time_domain

Structures de données

— struct time_domain_transform

Définitions de type

— typedef struct time_domain_transform time_domain_transform_t

Fonctions

— status_t time_domain_transforms_setup_init (vorbis_stream_t *stream, time_domain_transform_t **ptdt)
— void time_domain_transforms_free (time_domain_transform_t *tdt)
— status_t time_domain_transform_process (time_domain_transform_t *tdt, sample_t *fsamp, sample_t *tsamp, sample_t *filter, int mode)

4.14.1 Description détaillée

Module de gestion des transformations de domaine de temps. Ce module est responsable de la gestion des transformées en domaine temporel. Cette partie de la norme Vorbis est une extension possible pour les futures versions de la norme, mais n'est pas utilisée aujourd'hui, puisque la MDCT est utilisée systématiquement. En revanche, plusieurs algorithmes existent pour effectuer l'iMDCT. Dans un premier temps, vous implanterez l'algorithme décrit au paragraphe 3.4 du sujet. NB : Le module `time_domain.o` qui vous est fourni implante une iMDCT rapide (fast_imdct), il est donc normal que votre version soit (beaucoup) plus lente.

4.14.2 Documentation des structures de données

4.14.2.1 struct time_domain_transform

Structure de base de Time Domain Transform.

Cette structure devra être étendue pour stocker les informations nécessaires à l'implantation de l'iMDCT choisie. Par exemple pour l'implantation initiale, vous aurez besoin des tailles des fenêtres (stream->codec->blocksize). Rappel : vous ne devez PAS modifier ce fichier `time_domain.h` ! L'extension de type se fera dans vos fichiers .c, comme illustré dans `floor1_example.c` .

Champs de donnée

int	type	Type de Time Domain Transform, utile si vous tentez diverses implantations de l'iMDCT

4.14.3 Documentation des définitions de type

4.14.3.1 typedef struct time_domain_transform time_domain_transform_t

4.14.4 Documentation des fonctions

4.14.4.1 status_t time_domain_transform_process (time_domain_transform_t * *tdt*, sample_t * *fsamp*, sample_t * *tsamp*, sample_t * *filter*, int *mode*)

Fonction réalisant l' opération de changement de domaine (fréquentiel -> temporel) incluant l'iMDCT et le filtre. Dans un premier temps, vous implanterez l'algorithme iMDCT décrit au paragraphe 5.3.4 du sujet. Cet algorithme nécessite de connaître la taille N de la fenêtre courante, il faut donc avoir étendu le type time_domain_transform_t afin de stocker cette information, comme discuté ci-dessus.

Paramètres

in	*tdt*	est un pointeur sur la structure définissant la TDT
in	*fsamp*	est le vecteur des échantillons fréquentiels (de taille N/2)
out	*tsamp*	est le vecteur de sortie des échantillons temporels, il doit avoir été préalablement alloué (de taille N)
in	*filter*	est le vecteur des coefficients du filtre de l'iMDCT dans notre cas ils correspondent à la fonction de fenêtrage (de taille N)
in	*mode*	définit le type de fenêtre (petite 0 ou grande 1)

Renvoie

VBS_SUCCESS

Références tdt_int : :im.

Voici le graphe des appelants de cette fonction :

4.14.4.2 void time_domain_transforms_free (time_domain_transform_t * *tdt*)

Fonction réalisant la libération de la structure de time_domain_transform.

Paramètres

in	*tdt*	est un pointeur sur la structure à libérer.

Références tdt_int : :im.

Voici le graphe des appelants de cette fonction :

4.14.4.3 status_t time_domain_transforms_setup_init (vorbis_stream_t * *stream*, time_domain_transform_t ** *ptdt*)

Fonction responsable de la lecture de la partie de l'entête 3 qui est prévue pour cette extension (time_domain_transform), et l'initialisation de la TDT. Voir le paragraphe 5.2.3.b du sujet, ainsi que les commentaires ci-dessus concernant l'extension du type `time_domain_transform`.

Paramètres

in	*stream*	est un pointeur sur le flux Vorbis en cours de lecture.
out	*ptdt*	est un pointeur pour le retour de l'adresse de la structure allouée et initialisée dans cette fonction.

Renvoie

VBS_SUCCESS en cas de succes, VBS_OUTOFMEMORY en cas d'erreur d'allocation ou VBS_BADSTREAM en cas d'erreur de lecture dans le flux.

Références vorbis_codec : :blocksize, vorbis_stream : :codec, et tdt_int : :im.

Voici le graphe des appelants de cette fonction :

4.15 Vorbis_main

Structures de données

— struct vorbis_codec
— struct vorbis_stream

Définitions de type

— typedef struct vorbis_stream vorbis_stream_t
— typedef struct vorbis_codec vorbis_codec_t

Fonctions

— status_t decode_stream (ogg_logical_stream_t *ogg_stream, pcm_handler_t *pcm_hdler)
— vorbis_codec_t * vorbis_codec_new (void)
— void vorbis_codec_free (vorbis_codec_t *codec)
— vorbis_stream_t * vorbis_new (ogg_logical_stream_t *ogg, pcm_handler_t *pcm_hdler)
— void vorbis_free (vorbis_stream_t *vorbis)

4.15.1 Description détaillée

Ce module regroupe la définition des structures transversales utilisées dans le décodeur, ainsi que les fonctions liées au décodage des en-têtes et du flux principal Vorbis.

4.15.2 Documentation des structures de données

4.15.2.1 struct vorbis_codec

La structure vorbis_codec rassemble les informations de configuration glanées lors de la lecture des en-têtes.

Les champs de cette structure peuvent être rassemblés en deux groupes, le premier correspondant aux informations récupérées lors de la lecture de l'en-tête 1, le second correspondant à celles de l'en-tête 3. On notera que les informations de l'en-tête 2 ne sont pas stockées par notre décodeur, mais directement affichées sur le terminal.

Graphe de collaboration de vorbis_codec :

Champs de donnée

uint8_t	audio_channels	Nombre de canaux audio
uint32_t	audio_sample_-rate	Fréquence d'échantillonnage de ces canaux
uint32_t	bitrate_-maximum	Débit maximum
uint32_t	bitrate_minimum	Débit minimum
uint32_t	bitrate_nominal	Débit nominal
uint16_t	blocksize[2]	Taille des fenêtres (petite et grande)
codebook_-setup_t *	codebooks_desc	Informations de configuration des codebooks
floors_setup_t *	floors_desc	Informations de configuration des floors
mappings_-setup_t *	mappings_desc	Informations de configuration des mappings

window_modes-_setup_t *	modes_desc	Informations de configuration des modes
residues_setup-_t *	residues_desc	Informations de configuration des residues
time_domain_-transform_t *	tdt_desc	Informations de configuration de la TDT
uint32_t	vorbis_version	Version de la norme Vorbis utilisée

4.15.2.2 struct vorbis_stream

La structure vorbis_stream est la structure principale utilisée transversalement dans tout le décodeur Vorbis.

Elle contient à la fois le point d'entrée permettant de lire le flux à décoder, mais aussi la configuration du décodeur et enfin les échantillons PCM produits par notre décodeur. Cette structure n'est en aucun cas à étendre : elle garantit, avec vorbis_codec, la compatibilité des modules fournis avec ceux que vous développerez.

Graphe de collaboration de vorbis_stream :

Champs de donnée

vorbis_codec_t *	codec	Configuration du décodeur, remplie lors de la lecture des différents entêtes.
vorbis_io_t *	io_desc	Point d'entrée pour la lecture du flux.
pcm_handler_t *	pcm_hdler	Echantillons PCM produits après décodage du flux

4.15.3 Documentation des définitions de type

4.15.3.1 typedef struct vorbis_codec vorbis_codec_t

4.15.3.2 typedef struct vorbis_stream vorbis_stream_t

4.15.4 Documentation des fonctions

4.15.4.1 status_t decode_stream (ogg_logical_stream_t * *ogg_stream*, pcm_handler_t * *pcm_hdler*)

Fonction de décodage principal du flux Vorbis.

Cette fonction gère la machine d'état globale du décodage du flux Vorbis. Elle est chargée en particulier :
- — des initialisations nécessaires aux moments adéquats (avant ou après la lecture des entêtes) ;
- — de la lecture des entêtes ;
- — de la lecture de l'ensemble des paquets audio :
  - — avancement dans les paquets à l'aide du module vorbis_io ;
  - — lancement du décodage des paquets, via le module vorbis_packet ;
  - — traitement des PCM décodés via le module pcm_handler.
  - — Attention cependant à ne pas en produire trop ! Utiliser la fonction `vorbis_io_limit` pour connaître la limite d'échantillons à produire et tronquer la fin du flux le cas échéant.
- — libérations nécessaires

Paramètres

ogg_stream	est le pointeur sur le flux logique OGG contenant le flux Vorbis
pcm_hdler	est le pointeur sur le gestionnaire d'échantillons PCM à utiliser.

Renvoie

VBS_SUCCESS en cas de succès, VBS_FATAL en cas de passage de mauvais arguments ou VBS_BADSTREAM en cas d'erreur de lecture dans le flux.

Références vorbis_codec : :audio_channels, vorbis_codec : :audio_sample_rate, vorbis_codec : :blocksize, vorbis_stream : :codec, pcm_handler : :finalize, pcm_handler : :init, vorbis_stream : :io_desc, vorbis_packet : :pcm, vorbis_stream : :pcm_hdler, pcm_handler : :process, vorbis_free(), vorbis_header1_decode(), vorbis_header2_decode(), vorbis_header3_decode(), vorbis_io_limit(), vorbis_io_next_packet(), vorbis_new(), vorbis_packet_decode(), vorbis_packet_free(), et vorbis_packet_init().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.15.4.2 void vorbis_codec_free (vorbis_codec_t * *codec*)

Fonction qui permet de libérer la mémoire d'une structure vorbis_codec.

Paramètres

in	*codec*	est un pointeur sur la zone mémoire à libérer.

Références vorbis_codec : :codebooks_desc, codebooks_free(), vorbis_codec : :floors_desc, floors_free(), vorbis_codec : :mappings_desc, mappings_free(), vorbis_codec : :modes_desc, vorbis_codec : :residues_desc, residues_free(), vorbis_codec : :tdt_desc, time_domain_transforms_free(), et window_modes_free().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.15.4.3 vorbis_codec_t* vorbis_codec_new (void)

Fonction permettant d'allouer une structure vorbis_codec pour stocker les informations lues dans les entêtes.

Les champs de la structure ne sont pas initialisés ; ils le seront au fur et à mesure de la lecture (des entêtes) du flux Ogg.

Renvoie

un pointeur sur la structure nouvellement allouée ou `NULL` en cas de problème d'allocation.

Voici le graphe des appelants de cette fonction :

4.15.4.4 void vorbis_free (vorbis_stream_t * *vorbis*)

Fonction qui permet de libérer la mémoire d'une structure vorbis_stream_t.

Paramètres

in	*vorbis*	est un pointeur sur la zone mémoire à libérer.

Références vorbis_stream : :codec, vorbis_stream : :io_desc, vorbis_codec_free(), et vorbis_io_free().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.15.4.5 vorbis_stream_t* vorbis_new (ogg_logical_stream_t * *ogg*, pcm_handler_t * *pcm_hdler*)

Fonction permettant d'allouer et d'initaliser la structure principale vorbis_stream du décodeur.

Paramètres

in	*ogg*	le flux Ogg dans lequel est lu le flux Vorbis (seul le premier flux logique du container est lu dans ce projet).
in	*pcm_hdler*	le module de gestion des échantillons PCM décodés dans le flux Vorbis.

Renvoie

un pointeur sur la structure nouvellement allouée ou `NULL` en cas de problème d'allocation.

Références vorbis_stream : :codec, vorbis_stream : :io_desc, vorbis_stream : :pcm_hdler, vorbis_codec_new(), et vorbis_io_init().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.16 Common_header

Fonctions

— status_t vorbis_common_header (vorbis_stream_t ∗stream, uint8_t ∗header_type)

4.16.1 Description détaillée

Module transverse de lecture des en-têtes.

4.16.2 Documentation des fonctions

4.16.2.1 status_t vorbis_common_header (vorbis_stream_t ∗ *stream,* uint8_t ∗ *header_type*)

Fonction effectuant l'opération de décodage commune aux trois en-têtes, à savoir les 7 octets contenant l'identification du type de paquet, du type de header et du flux ("vorbis"), comme indiqué en section 5.2 du sujet. Elle doit donc être appelée au début de chaque fonction décodant un type d'en-tête particulier (header1_decode, header2-_decode, header3_decode).

Paramètres

in	*stream*	est le pointeur sur le flux Vorbis à decoder
out	*header_type*	contient le type de header décodé par cette fonction

Renvoie

VBS_SUCCESS en cas de succès, VBS_BADSTREAM en cas d'erreur de lecture dans le flux

Voici le graphe des appelants de cette fonction :

4.17 Header1

Fonctions

— status_t vorbis_header1_decode (vorbis_stream_t ∗stream)

4.17.1 Description détaillée

Module de lecture de l'en-tête 1.

4.17.2 Documentation des fonctions

4.17.2.1 status_t vorbis_header1_decode (vorbis_stream_t ∗ *stream*)

Fonction effectuant l'opération de décodage de l'en-tête 1 du flux Vorbis, en-tête d'identification. En pratique, cette fonction lit sur le flux, effectue des vérifications sur les informations lues comme détaillé en section 5.2.1 du sujet, et remplit les champs du codec (inclus dans stream) concernés.

Paramètres

in,out	*stream*	est le pointeur sur le flux Vorbis à decoder. La structure codec qu'il contient est déjà allouée avant l'appel à vorbis_header1_decode et est remplie en fonction des informations lues par cette fonction.

Renvoie

VBS_SUCCESS en cas de succès, VBS_BADSTREAM en cas d'erreur de lecture dans le flux

Références vorbis_codec : :audio_channels, vorbis_codec : :audio_sample_rate, vorbis_codec : :bitrate_maximum, vorbis_codec : :bitrate_minimum, vorbis_codec : :bitrate_nominal, vorbis_codec : :blocksize, vorbis_stream : :codec, vorbis_common_header(), et vorbis_codec : :vorbis_version.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.18 Header2

Fonctions

— status_t vorbis_header2_decode (vorbis_stream_t ∗stream)

4.18.1 Description détaillée

Module de lecture de l'en-tête 2.

4.18.2 Documentation des fonctions

4.18.2.1 status_t vorbis_header2_decode (vorbis_stream_t ∗ *stream*)

Fonction effectuant l'opération de décodage de l'en-tête 2 du flux Vorbis, en-tête de commentaires. En pratique, cette fonction ne fait que lire sur le flux. Les informations lues ne sont pas stockées, mais affichées sur le terminal de façon synthétique. Cette fonction implémente le pseudo-code présenté en section 5.2.2 du sujet.

Paramètres

in	*stream*	est le pointeur sur le flux Vorbis à decoder

Renvoie

VBS_SUCCESS en cas de succès, VBS_BADSTREAM en cas d'erreur de lecture dans le flux

Références vorbis_common_header().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.19 Header3

Fonctions

— status_t vorbis_header3_decode (vorbis_stream_t ∗stream)

4.19.1 Description détaillée

Module de lecture de l'en-tête 3.

4.19.2 Documentation des fonctions

4.19.2.1 status_t vorbis_header3_decode (vorbis_stream_t ∗ *stream*)

Fonction effectuant l'opération de décodage de l'en-tête 3 du flux Vorbis, en-tête de configuration. En pratique, cette fonction est en charge d'appeler les fonctions d'allocation/initialisation des structures de données de chacun des éléments du Vorbis (mode, mapping, codebook, residue, floor et transformée). Ces fonctions, comprenant la lecture du flux pour mettre à jour les champs des structures codebooks_desc, floors_desc, residues_desc, mappings_desc et modes_desc, sont quant à elles implémentées indépendamment, dans chacun des modules concernés. La section 5.2.3 du sujet détaille dans quel ordre les fonctions d'allocation/initialisation de ces modules doivent être appelées.

Paramètres

in,out	*stream*	est le pointeur sur le flux Vorbis à decoder. Elle contient les structures xxx_setup_t (via l'objet codec) qui sont remplies par cette fonction.

Renvoie

VBS_SUCCESS en cas de succès, VBS_BADSTREAM en cas d'erreur de lecture dans le flux

Références codebook_setup_init(), vorbis_codec : :codebooks_desc, codebooks_free(), vorbis_stream : :codec, vorbis_codec : :floors_desc, floors_free(), floors_setup_init(), vorbis_stream : :io_desc, vorbis_codec : :mappings_desc, mappings_free(), mappings_setup_init(), vorbis_codec : :modes_desc, vorbis_io : :readbits, vorbis_codec : :residues_desc, residues_free(), residues_setup_init(), vorbis_codec : :tdt_desc, time_domain_transforms_setup_init(), vorbis_common_header(), et window_modes_setup_init().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.20 Vorbis_io

Définitions de type

— typedef struct vorbis_io vorbis_io_t

Fonctions

— vorbis_io_t ∗ vorbis_io_init (ogg_logical_stream_t ∗ogg_desc)
— void vorbis_io_free (vorbis_io_t ∗io)
— status_t vorbis_read_nbits (uint32_t nb_bits, uint32_t ∗dst, vorbis_io_t ∗io, uint32_t ∗p_count)
— status_t vorbis_io_next_packet (vorbis_io_t ∗io)
— int64_t vorbis_io_limit (vorbis_io_t ∗io)

4.20.1 Description détaillée

Module de gestion des lectures binaires dans le flux.

4.20.2 Documentation des définitions de type

4.20.2.1 typedef struct vorbis_io vorbis_io_t

4.20.3 Documentation des fonctions

4.20.3.1 void vorbis_io_free (vorbis_io_t ∗ *io*)

Fonction qui permet de libérer une structure de travail de lecture binaire.

Paramètres

in,out	*io*	est le pointeur sur la structure à libérer

Voici le graphe des appelants de cette fonction :

4.20.3.2 vorbis_io_t∗ vorbis_io_init (ogg_logical_stream_t ∗ *ogg_desc*)

Fonction qui permet d'allouer et initialiser une structure de travail de lecture binaire.

Paramètres

in	*ogg_desc*	est un pointeur sur le flux OGG correspondant.

Renvoie

un pointeur sur la zone nouvellement allouée et initialisée, NULL en cas d'erreur.

Références ogg_desc.

Voici le graphe des appelants de cette fonction :

4.20.3.3 int64_t vorbis_io_limit (vorbis_io_t ∗ *io*)

Fonction permet de récupérer une information mise à la disposition du codec par le conteneur (OGG en l'occurence). Cette information est la limite du nombre d'échantillon à produire grâce au `ogg_packet_position`

Paramètres

in	*io*	est le pointeur sur la structure de travail

Renvoie

la limite du flux courant en terme d'échantillons

Références ogg_desc, et ogg_packet_position().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.20.3.4 status_t vorbis_io_next_packet (vorbis_io_t * *io*)

Fonction qui permet de passer au paquet OGG suivant, pour celà il interagit avec le flux OGG.

Paramètres

in	*io*	est le pointeur sur la structure de travail

Renvoie

VBS_SUCCESS en cas de succes, VBS_EOS si la fin du flux est atteinte, VBS_FATAL en cas d'erreur.

Références buffer, nbits, offset, ogg_desc, ogg_packet_next(), ogg_packet_size(), readbits, et status.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.20.3.5 status_t vorbis_read_nbits (uint32_t *nb_bits*, uint32_t * *dst*, vorbis_io_t * *io*, uint32_t * *p_count*)

Fonction principale du module, elle permet à l'ensemble des autres modules de vorbis d'effectuer des opérations de lecture au bit, *nb_bits* à partir du container OGG.

Paramètres

in	*nb_bits*	est le nombre de bits à lire (max : 32)
out	*dst*	est un pointeur sur le mots mémoire de 32bits où placer les données lues
in	*io*	est le pointeur sur la structure de travail
out	*p_count*	est un pointeur sur un mot de 32 bits où sera placé le nombre de bits effectivement lus

Renvoie

VBS_SUCCESS en cas de success, VBS_EOP en cas de fin de paquet ou VBS_FATAL en cas d'erreur.

Références buffer, nbits, offset, ogg_desc, readbits, et status.

4.21 Vorbis_packet

Structures de données

— struct vorbis_packet

Définitions de type

— typedef struct vorbis_packet vorbis_packet_t

Fonctions

— vorbis_packet_t ∗ vorbis_packet_init (uint16_t ∗blocksize, uint8_t nb_chan)
— void vorbis_packet_free (vorbis_packet_t ∗pkt)
— status_t vorbis_packet_decode (vorbis_stream_t ∗stream, vorbis_packet_t ∗pkt, uint16_t ∗nb_samp)

4.21.1 Description détaillée

Module de gestion des paquets Vorbis.

4.21.2 Documentation des structures de données

4.21.2.1 struct vorbis_packet

Structure de travail pour le décodage des paquets Vorbis.

Cette structure contient plusieurs types d'informations :
— générales, mises à jour au fur et à mesure du décodage : *nb_chan*, *pcm_offset*
— spécifiques au paquet courant (en cours de décodage) : la taille de fenêtre *N* du paquet, et des buffers de travail *spectral*, *dec_residues*, *residues*, *temporal* ainsi que *no_residue* et *do_not_decode*. Ces espaces mémoires sont partagés pour éviter de les allouer puis les désallouer à chaque fois qu'un nouveau paquet est décodé. Ils sont alloués une seule fois lors de l'initialisation de la structure, avec une taille n_max correspondant à la plus grande des tailles de fenêtre (taille de la "grande fenêtre").

Champs de donnée

sample_t ∗∗	dec_residues	Buffers pour les residues décodés du paquet (dans l'ordre des submaps). Taille : nombre_de_canaux x (n_max/2) Attention : le vecteur pour le canal 0 doit pouvoir contenir un residue de type 2. Sa taille est donc de nb_chan∗n/2.
uint8_t ∗	do_not_decode	Tableau des fanions de non décodage. Taille : nombre_de_canaux
uint8_t	nb_chan	Nombre de canaux
uint8_t ∗	no_residue	Tableau des fanions d'absence de residue. Taille : nombre_de_canaux
int16_t ∗∗	pcm	Buffers de sortie des PCM décodés dans un paquet. Taille : nombre_de_canaux x (n_max)
sample_t ∗∗	residues	Buffers des residues réordonnés après l'étape de découplage. Taille : nombre_de_canaux x (n_max/2)
uint16_t	size	Taille du paquet courant (i.e. de la fenêtre correspondante)
sample_t ∗∗	spectral	Buffers de la représentation spectrale : les courbes gros grain après le décodage des *floor*, puis les courbes spectrales issues du produit scalaire. Taille : nombre_de_canaux x (n_max/2)

sample_t ∗∗	temporal	Buffers de représentation temporelle. Taille : nombre_de_canaux x (n_-max)

4.21.3 Documentation des définitions de type

4.21.3.1 typedef struct vorbis_packet vorbis_packet_t

4.21.4 Documentation des fonctions

4.21.4.1 status_t vorbis_packet_decode (vorbis_stream_t ∗ *stream,* vorbis_packet_t ∗ *pkt,* uint16_t ∗ *nb_samp*)

Fonction qui effectue le décodage d'un paquet audio du flux Vorbis, comme décrit en section 5.3. Elle regroupe différentes opérations :

- lecture du *mode* ;
- décodage des informations dans le flux (floors, residues, ...) puis couplage inverse des residues. Ces étapes sont en fait déléguées au *mapping* associé au paquet courant.
- génération du signal, du *dot_product* à la production des échantillons PCM.

Remarques importantes :

- La chaîne de traitement (décodage) fonctionne sur des `sample_t`, or les PCM produits devront être de type `int16_t`. Une conversion est donc nécessaire pour produire les échantillons, mais vous devez faire attention à la saturation.
- La fonction `vorbis_packet_decode` est à l'image des fonctions `vorbis_header1_decode`, `vorbis_header2_decode` et `vorbis_header3_decode` : elle ne s'occupe que de la lecture d'un paquet, et ne prend pas en charge le changement de paquet qui doit être géré au niveau supérieur.

Paramètres

in	*stream*	est le pointeur sur le flux Vorbis en cours de décodage
in	*pkt*	est le pointeur la structure de travail de décodage de paquets Vorbis
out	*nb_samp*	renvoie le nombre d'échantillons produits lors de la lecture de ce paquet

Renvoie

VBS_SUCCESS dans le cas où le paquet a été décodé avec succès, VBS_BADSTREAM en cas d'erreur lors de la lecture dans le flux.

Références vorbis_packet_int : :cached, vorbis_stream : :codec, envelope : :curr_window, dot_product(), vorbis-_packet_int : :envelope, envelope_overlap_add(), vorbis_packet_int : :filter, envelope : :initialized, nb_chan, pcm, vorbis_packet_int : :produced, residues, size, spectral, vorbis_codec : :tdt_desc, temporal, et time_domain_-transform_process().

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.21.4.2 void vorbis_packet_free (vorbis_packet_t ∗ *pkt*)

Fonction qui permet de libérer la structure de travail de type `vorbis_packet_t`

Paramètres

in	*pkt*	est un pointeur sur la structure de travail à libérer, il doit bien sûr être non nul.

Références vorbis_packet_int : :cached, dec_residues, do_not_decode, vorbis_packet_int : :envelope, envelope-_free(), vorbis_packet_int : :filter, nb_chan, no_residue, pcm, vorbis_packet_int : :produced, residues, spectral, et temporal.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

4.21.4.3 vorbis_packet_t* vorbis_packet_init (uint16_t * *blocksize*, uint8_t *nb_chan*)

Fonction qui permet d'allouer et d'initialiser la structure de travail de type `vorbis_packet_t`.

Paramètres

in	*blocksize*	est un tableau contenant les tailles des deux types de fenêtres.
in	*nb_chan*	est le nombre de canaux présents dans le flux en cours de décodage.

Renvoie

un pointeur sur la structure fraichement allouée et initialisée, NULL en cas d'erreur.

Références vorbis_packet_int : :blocksizes, vorbis_packet_int : :cached, dec_residues, do_not_decode, vorbis_packet_int : :envelope, envelope_init(), vorbis_packet_int : :filter, nb_chan, no_residue, pcm, vorbis_packet_int : :produced, residues, size, spectral, et temporal.

Voici le graphe d'appel pour cette fonction :

Voici le graphe des appelants de cette fonction :

5 Documentation des structures de données

5.1 Référence de la structure _pcm_handler_format_t

`#include "pcm_handler_internal.h"`

Graphe de collaboration de _pcm_handler_format_t :

Champs de données

- const char ∗ name
- pcm_handler_desc_t ∗(∗ create)(const char ∗arg)
- void(∗ delete)(pcm_handler_desc_t ∗hdlr)

5.1.1 Documentation des champs

5.1.1.1 pcm_handler_desc_t∗(∗ _pcm_handler_format_t : :create)(const char ∗arg)

5.1.1.2 void(∗ _pcm_handler_format_t : :delete)(pcm_handler_desc_t ∗hdlr)

5.1.1.3 const char∗ _pcm_handler_format_t : :name

La documentation de cette structure a été générée à partir du fichier suivant :
- pcm_handler_internal.h

5.2 Référence de la structure residue_type0_2

`#include "residue_type0.h"`

Graphe de collaboration de residue_type0_2 :

Champs de données

- residue_t base
- uint32_t begin
- uint32_t end
- uint32_t partition_size
- uint8_t classifications
- codebook_t ∗ classbook
- uint8_t ∗ cascade
- codebook_t ∗∗∗ books
- status_t(∗ partition_decode)(vorbis_stream_t ∗stream, residue_type0_2_t ∗residue, codebook_t ∗vqbook, int offset, sample_t ∗v)

5.2.1 Documentation des champs

5.2.1.1 residue_t residue_type0_2 : :base

5.2.1.2 uint32_t residue_type0_2 : :begin

5.2.1.3 codebook_t∗∗∗ residue_type0_2 : :books

5.2.1.4 uint8_t∗ residue_type0_2 : :cascade

5.2.1.5 codebook_t∗ residue_type0_2 : :classbook

5.2.1.6 uint8_t residue_type0_2 : :classifications

5.2.1.7 uint32_t residue_type0_2 : :end

5.2.1.8 status_t(∗ residue_type0_2 : :partition_decode)(vorbis_stream_t ∗stream, residue_type0_2_t ∗residue, codebook_t ∗vqbook, int offset, sample_t ∗v)

5.2.1.9 uint32_t residue_type0_2 : :partition_size

La documentation de cette structure a été générée à partir du fichier suivant :
— residue_type0.h

Index