les signaux audios et l'ESP32

l'environnement ESP-ADF en pratique

Tam Hanna (Hongrie)

Le développement de tout nouvel appareil électronique grand public constitue incontestablement un défi. Ces produits ont en général un cycle de vie extrêmement court, de sorte que le temps consacré au développement est primordial pour le succès. Ceci est particulièrement vrai pour les applications audios où on utilise des algorithmes pour mettre en œuvre diverses fonctions de codecs standard. L'*Espressif Audio Development Framework* (ESP-ADF) est un outil puissant avec des ressources qui permettront aux développeurs de telles applications de gagner du temps et de l'énergie.

L'Espressif Audio Development Framework fournit une collection logicielle d'algorithmes et de codecs qui utilisent un format standardisé. Pour développer une application, le concepteur n'a qu'à relier en séquence les « circuits logiciels » appropriés sans avoir à s'occuper de leurs détails internes individuels.

Cet article a pour but d'expliquer les bases du fonctionnement pratique de l'ADF. J'ai beaucoup utilisé l'outil ADF d'Espressif pour m'aider à mettre en œuvre des solutions conformes aux exigences de conception de mes clients. Je peux maintenant partager avec vous beaucoup de choses que j'aurais aimé connaître lorsque j'ai commencé à utiliser l'ADF.

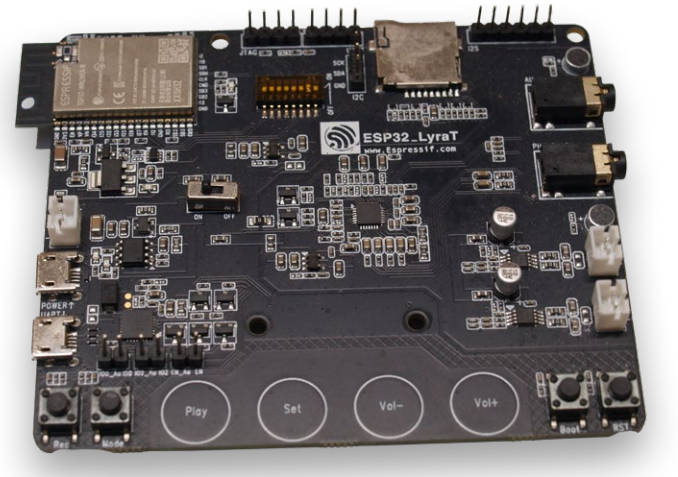

Figure 1. Le petit circuit imprimé noir comprend tout...

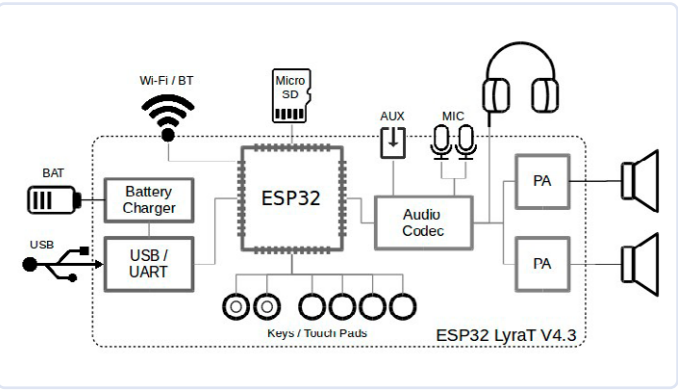

Figure 2. ... ce dont vous avez besoin pour prototyper des applications audio (source : [6]).

Matériel nécessaire

Soyons clairs : l'ESP-ADF est compatible avec tout système à base d'ESP32. Pour la « partie audio », la bibliothèque offre une interface de pilote standardisée qui prend en charge les données. Avant de finaliser la conception matérielle, une grande partie du logiciel peut être écrite à l'avance et testée sur une carte de développement

ordinaire. Espressif propose une gamme de cartes de ce type et pour mes développements maison j'apprécie la carte LyraT présentée dans les **figures 1** et **2**.

Le plus important est le signal analogique en « bande de base » numérisé, fourni par la puce ES8388 (avec deux microphones analogiques). La communication se fait à la fois via I2S et via un groupe de broches dédiées – le schéma de circuit et les interconnexions sont fournis par Espressif et peuvent être facilement reproduits pour la conception du dispositif final. La puce se trouve sans peine auprès de UTSource et LCSC.

Lorsqu'on travaille avec la LyraT, il ne reste pas beaucoup de GPIO disponibles sur l'ESP32 embarqué. Lorsque la conception du matériel de l'appareil devient plus complexe, vous pouvez rapidement manquer d'options d'interface. Dans ce cas, il peut être nécessaire d'envisager l'utilisation d'un second processeur dédié à la prise en charge de la communication matérielle.

Installation de l'ADF

Espressif fournit l'ADF comme un greffon pour son environnement IDF existant. L'ADF ne peut pas être utilisé dans Arduino, et n'est même pas supporté par certaines versions de l'IDF d'Espressif.

La façon la plus pratique d'installer l'environnement de travail est de télécharger le dépôt *ESP-ADF* complet. Je commence par les commandes suivantes pour créer un sous-dossier, *esp4*, où le dépôt sera stocké localement :

```
(base)tamhan@tamhan-thinkpad:~$ mkdir esp4
(base)tamhan@tamhan-thinkpad:~$ cd esp4/
```

Le dépôt complet peut maintenant être téléchargé en utilisant le client en ligne de commande de Git :

```
(base)tamhan@tamhan-thinkpad:~/esp4$ git clone
--recursive https://github.com/espressif/esp-adf.git
Cloning into 'esp-adf'...
```

Une inspection minutieuse de la sortie (non montrée ici) révèle que le dépôt ADF comporte plusieurs références externes – pour cette raison, un téléchargement via la fonction de téléchargement du navigateur intégrée à GitHub n'est pas possible.

Une version de l'IDF d'Espressif est incluse avec l'ADF. Certains des composants nécessitent la présence de la variable d'environnement `ADF_PATH`. Si vous souhaitez utiliser une fenêtre de ligne de commande pour traiter les applications basées sur l'ADF, vous devez définir la variable d'environnement à l'aide des commandes suivantes :

```
(base)tamhan@tamhan-thinkpad:~/esp4$ cd esp-adf
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf$ export
ADF_PATH=$PWD
```

La présence de la variable `ADF_PATH` peut affecter les projets IDF normaux : j'évite généralement cela en utilisant une fenêtre de terminal différente pour le travail « normal » sur ESP32.

Pour la version IDF intégrée, il faut exécuter l'installation habituelle à l'étape suivante pour obtenir les compilateurs et autres dépendances :

```
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf$ cd
$ADF_PATH/esp-idf
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/esp-idf$ ./
install.sh
Detecting the Python interpreter
. . .All done! You can now run:

  . ./export.sh
```

Il faut alors appeler un script pour la configuration de la chaîne d'outils (les deux points consécutifs ne sont pas une faute de frappe) :

```
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/esp-idf$ .
./export.sh
```

En pratique, il est conseillé à ce stade de créer un script shell pour le paramétrage – avec Bash, il peut alors se configurer automatiquement à la demande.

Architecture logicielle : le pipeline

Après avoir téléchargé et installé l'ADF, il est temps de commencer à se pencher sur sa structure théorique. Fidèle au concept de « circuit logiciel » développé à l'origine par The Stepstone Corporation, l'ADF consiste aussi pour le développeur à essentiellement mettre en œuvre un pipeline constitué d'une séquence d'étapes de traitement. À titre d'illustration, la **figure 3** présente un modèle de flux de travail pour l'implémentation d'un lecteur MP3.

Espressif ne semble pas trop s'étendre sur les éléments concrets du pipeline – la **figure 4** montre les rôles et exemples de mise en œuvre proposés.

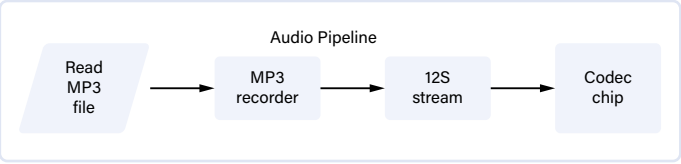

Figure 3. Ce pipeline décode les MP3 (source : [3]).

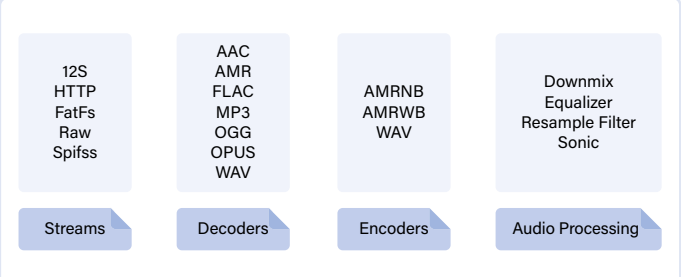

Figure 4. Les éléments des pipelines peuvent être divisés en plusieurs types (source : [3]).

La seule particularité du système actuel est que le concept Unix selon lequel « tout est fichier » s'applique aussi aux périphériques. L'API, documentée en détail sous [1] et dont la compréhension est facilitée par les fonctions de bouton fournies sous [2], permet la création d'« enveloppes » qu'on peut directement intégrer dans le pipeline. Pour les développeurs d'applications, ce n'est toutefois pertinent que dans la mesure où l'idée de considérer également les périphériques comme des éléments du pipeline peut nécessiter une réflexion approfondie.

La documentation de l'API pour les différents éléments du pipeline se trouve sous [3].

Architecture logicielle : construire un lecteur MP3

Pour mieux comprendre, examinons un des exemples fournis par Espressif : Si vous souhaitez mettre en œuvre un projet quelconque à l'aide de l'ADF, vous avez tout intérêt à commencer par rechercher un modèle similaire ou du moins apparenté parmi les exemples. Naviguez vers le répertoire des exemples en utilisant la commande suivante :

```
cd $ADF_PATH/examples/
```

Nous allons voir dans les prochaines étapes comment construire un lecteur MP3 classique. Allez dans le répertoire *~/esp4/esp-adf/examples/get-started/play_mp3_control/main* et ouvrez le fichier *play_mp3_control_example.c* avec votre éditeur préféré.

Le plus important est le point d'entrée, qui montre la création de certaines variables membres. Notre exemple a besoin d'un objet pipeline et deux descripteurs sont nécessaires pour les éléments individuels :

```
void app_main(void)
{
   audio_pipeline_handle_t pipeline;
   audio_element_handle_t i2s_stream_writer, mp3_decoder;
...
```

Le matériel est pris en compte dans l'ADF sous forme d'une abstraction intitulée *Board*. Votre sélection ou paramétrage s'effectue dans le cadre de *Menuconfig*. Les commandes à créer dans le code associé évaluent principalement les constantes de compilation :

```
   audio_board_handle_t board_handle = audio_board_init();
   audio_hal_ctrl_codec(board_handle->audio_hal,
   AUDIO_HAL_CODEC_MODE_BOTH, AUDIO_HAL_CTRL_START);
. . .
   int player_volume;
   audio_hal_get_volume(board_handle->audio_hal,
    &player_volume);
```

Il est intéressant de noter que le HAL contient aussi les paramètres de commande du volume. L'étape suivante est la configuration de l'objet principal du pipeline :

```
   audio_pipeline_cfg_t pipeline_cfg =
    DEFAULT_AUDIO_PIPELINE_CONFIG();
   pipeline = audio_pipeline_init(&pipeline_cfg);
   mem_assert(pipeline);
```

Les développeurs ayant une expérience antérieure d'ESP-IDF trouveront une grande partie de ces éléments familiers. L'objet pipeline est créé en passant une structure de configuration, qui peut ensuite être transmise d'un utilisateur à l'autre.

Vient ensuite la génération des éléments réels du pipeline. Le premier élément est le décodeur MP3 :

```
   mp3_decoder_cfg_t mp3_cfg =
    DEFAULT_MP3_DECODER_CONFIG();
   mp3_decoder = mp3_decoder_init(&mp3_cfg);
   audio_element_set_read_cb(mp3_decoder,
    mp3_music_read_cb, NULL);
```

La question qui se pose maintenant (surtout au vu du pipeline présenté à la **figure 3**) est de savoir comment les données seront fournies. La réponse à cette question est la méthode `audio_element_set_read_cb`, qui reçoit en paramètre une fonction (structurée selon le schéma suivant) :

```
int mp3_music_read_cb(audio_element_handle_t el,
        char *buf, int len,
        TickType_t wait_time, void *ctx) {
   int read_size = file_marker.end
      - file_marker.start - file_marker.pos;
   if (read_size == 0) {
       return AEL_IO_DONE;
   } else if (len < read_size) {
       read_size = len;
   }
   memcpy(buf, file_marker.start +
    file_marker.pos, read_size);
   file_marker.pos += read_size;
   return read_size;
}
```

La fonction de rappel fournit les informations à traiter par le codec via le tampon. Le flux I2S est alors simplifié ; il provient de `AUDIO_STREAM_WRITER` et paramétré en sortie afin que les informations entrantes soient dirigées vers le matériel de sonorisation I2S :

```
i2s_stream_cfg_t i2s_cfg = I2S_STREAM_CFG_DEFAULT();
i2s_cfg.type = AUDIO_STREAM_WRITER;
i2s_stream_writer = i2s_stream_init(&i2s_cfg);
```

La création d'éléments de pipeline ne les déclare pas automatiquement : Il s'agit d'une facilité de l'ESP-ADF pour les développeurs qui veulent utiliser plusieurs pipelines en même temps. La configuration du pipeline se fait à la place par la déclaration des différents éléments, chacun d'entre eux possédant aussi une chaîne qui sert d'identifiant (ID) :

```
audio_pipeline_register(pipeline, mp3_decoder, "mp3");
```

Listage 1. Traitement des événements du lecteur MP3.

```
while (1) {
    audio_event_iface_msg_t msg;
    esp_err_t ret = audio_event_iface_listen(evt, &msg, portMAX_DELAY);
    if (ret != ESP_OK) {
        continue;
    }
    if (msg.source_type == AUDIO_ELEMENT_TYPE_ELEMENT && msg.source == (void *) mp3_decoder
                && msg.cmd == AEL_MSG_CMD_REPORT_MUSIC_INFO) {
        audio_element_info_t music_info = ;
        audio_element_getinfo(mp3_decoder, &music_info);
        ESP_LOGI(TAG, "[ * ] Receive music info from mp3 decoder, sample_rates=%d, bits=%d, ch=%d",
                music_info.sample_rates, music_info.bits, music_info.channels);
        audio_element_setinfo(i2s_stream_writer, &music_info);
        i2s_stream_set_clk(i2s_stream_writer, music_info.sample_rates,
                music_info.bits, music_info.channels);
        continue;
    }
}
```

```
audio_pipeline_register(pipeline, i2s_stream_writer,
"i2s");
const char *link_tag[2] = {"mp3", "i2s"};
audio_pipeline_link(pipeline, &link_tag[0], 2);
```

Dans le deuxième acte, le pipeline est alors constitué à l'aide d'un tableau qui fournit les ID des chaînes individuelles dans le bon ordre. Pour le lecteur MP3, seule l'activation du pipeline manque à ce stade :

```
ESP_LOGI(TAG, "[ 5.1 ] Start audio_pipeline");
. . .
audio_pipeline_run(pipeline);
```

Dans le cas du lecteur MP3, il reste un traitement relativement important pour réagir à divers événements (**listage 1**). À ce stade, le traitement exact des événements n'est pas pertinent – il est plus intéressant d'exécuter le processus de compilation à titre de test :

```
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/examples/
get-started/play_mp3_control$ make build
```

Une remarque à ce sujet : Pour garder les données de mes clients professionnels séparées et sécurisées, j'écris cet article sur mon ordinateur portable de voyage. La version Ubuntu utilisée ici utilise la version 3.4.3 de CMake, c'est pourquoi il n'est pas possible d'exécuter idf.py. En pratique, idf.py est bien sûr toujours préférable à l'utilisation de `make` et sera également obligatoire dans les futures versions de l'IDF.

Avec un squelette de projet impeccable, la récompense de l'effort est l'écran *menuconfig*, qui offre quelques options supplémentaires par rapport à un projet IDF normal. La plus importante est l'option *(Top) Audio HAL Audio board*, où vous pouvez sélectionner la configuration de carte appropriée, comme le montre la **figure 5**.

Après la sauvegarde, le processus de construction commence de la même manière que pour tout projet IDF. Il est particulièrement important que la carte dispose de deux ports Micro-USB : le port POWER est utilisé pour alimenter la carte, tandis que le port UART est uniquement utilisé pour la communication de données. Normalement, j'utilise simplement deux câbles USB séparés pour connecter la station de travail à la carte.

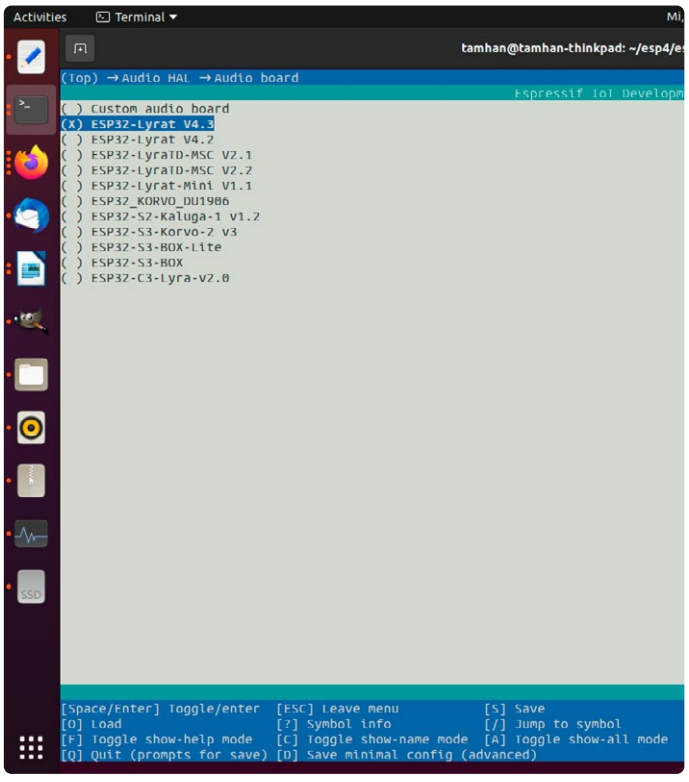

Figure 5. ADF étend Menuconfig avec plusieurs paramètres.

Une chose à noter à propos de la carte LyraT est que (contrairement à de nombreuses autres cartes de développement) elle ne peut pas être reprogrammée automatiquement ; avant même qu'une connexion puisse être établie, vous devez maintenir le bouton poussoir *Boot* enfoncé tout en appuyant brièvement sur RST. Vous devriez alors voir :

```
Serial port /dev/ttyUSB0
Connecting.....................................
```

Si vous vous êtes trompé dans la séquence de boutons-poussoirs, le système interprète qu'il s'agit d'un message de communication corrompu… essayez à nouveau :

```
A fatal error occurred: Failed to connect to ESP32:
Invalid head of packet (0x1B): Possible serial noise or
corruption.
```

Une fois le processus de flashage terminé avec succès, vous pouvez brancher un casque dans la prise phono et appuyer à nouveau sur *Reset*. Si tout est en ordre, vous devriez entendre une mélodie.

Enfin, il faut rappeler que l'exécution du script de configuration export.sh ne suffit pas à paramétrer complètement l'environnement. Avant de l'activer, il est toujours nécessaire de paramétrer correctement la variable d'environnement *ADF_PATH* :

```
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf$ export
ADF_PATH=$PWD
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/esp-idf$ .
./export.sh
Setting IDF_PATH to '/home/tamhan/esp4/esp-adf/esp-idf'
. . .
```

Traitement des données d'entrée

L'utilisation du pipeline ESP-ADF n'a pas de sens que dans le contexte de la lecture de médias : il est tout aussi légitime d'y intégrer des calculs. Par exemple, l'un de mes récents travaux pour un client nécessitait le traitement de données microphoniques par un algorithme. Le calcul proprement dit a été placé dans un élément du pipeline, dont je voudrais vous montrer brièvement la structure générale.

Nous commençons par initialiser le pipeline, et puisque les informations audios arrivent via le bus I2S, un élément de flux I2S est à nouveau requis dans la première étape. Ici, cependant, sa configuration inclut maintenant l'indicateur AUDIO_STREAM_READER, qui le marque comme une entrée ou une source de données :

```
i2s_stream_cfg_t i2s_cfg = I2S_STREAM_CFG_DEFAULT();
i2s_cfg.type = AUDIO_STREAM_READER;
#if defined CONFIG_ESP_LYRAT_MINI_V1_1_BOARD
    i2s_cfg.i2s_port = 1;
#endif
i2s_stream_reader = i2s_stream_init(&i2s_cfg);
```

Outre le flux I2S, li nous faut aussi son compagnon, qui encapsule l'algorithme propriétaire. Voici son initialisation :

```
tams_stream_cfg_t fatfs_cfg = TAMS_STREAM_CFG_DEFAULT();
fatfs_cfg.type = AUDIO_STREAM_WRITER;
tams_stream = tams_stream_init(&fatfs_cfg);
```

La construction proprement dite du pipeline se fait ensuite en assignant des chaînes et en les passant à `audio_pipeline_link()`:

```
audio_pipeline_register(pipeline, i2s_stream_reader,
"i2s");
audio_pipeline_register(pipeline, tams_stream, "tam");
audio_pipeline_link(pipeline, (const char *[]) {"i2s",
"tam"}, 2);
```

Avec cela, nous pouvons passer au fichier d'en-tête qui fournit les éléments nécessaires à la mise en service de la tâche « Tam ». Le plus important est la structure de configuration, qui contient, entre autres, les méta-informations requises pour FreeRTOS (**listage 2**).

`TAMS_STREAM_CFG_DEFAULT` est une macro de commodité qui permet la création des structures avec les paramètres par défaut. Pour des raisons de place, nous ne pouvons pas imprimer les constantes ici :

Listage 2. Configuration des éléments du Pipeline.

```
typedef struct {
   audio_stream_type_t     type;          /*!< Stream type */
   int                     buf_sz;        /*!< Audio Element Buffer size */
   int                     out_rb_size;   /*!< Size of output ring buffer */
   int                     task_stack;    /*!< Task stack size */
   int                     task_core;     /*!< Task running in core (0 or 1) */
   int                     task_prio;     /*!< Task priority (based on freeRTOS priority) */
} tams_stream_cfg_t;
```

Listage 3. Initialisation.

```
audio_element_handle_t tams_stream_init(tams_stream_cfg_t *config)
{
    audio_element_handle_t el;
    tams_stream_t *fatfs = audio_calloc(1, sizeof(tams_stream_t));

    AUDIO_MEM_CHECK(TAG, fatfs, return NULL);

    audio_element_cfg_t cfg = DEFAULT_AUDIO_ELEMENT_CONFIG();
    cfg.open = _tams_open;
    cfg.close = _tams_close;
    cfg.process = _tams_process;
    cfg.destroy = _tams_destroy;
    cfg.task_stack = config->task_stack;
    cfg.task_prio = config->task_prio;
    cfg.task_core = config->task_core;
    cfg.out_rb_size = config->out_rb_size;
    cfg.buffer_len = config->buf_sz;
    if (cfg.buffer_len == 0) {
       cfg.buffer_len = TAMS_STREAM_BUF_SIZE;
    }
    cfg.tag = "file";
...
}
```

```
#define TAMS_STREAM_CFG_DEFAULT() {\
    .task_prio = TAMS_STREAM_TASK_PRIO, \
    . . .
}
```

La seule exception concerne `BUF_SIZE`. Les données fournies ont une largeur de 16 bits, c'est pourquoi nous avons besoin de la valeur constante suivante pour traiter les blocs d'une longueur de 1024 slots :

```
#define TAMS_STREAM_BUF_SIZE   (1024*2)
```

Avec cela, nous pouvons passer à l'initialisation, voir le **listage 3**.

La valeur de `cfg.buffer_len` détermine le nombre de mots de données à fournir par cycle. À part ça, il n'y a essentiellement que du code de gestion banal :

```
    if (config->type == AUDIO_STREAM_WRITER) {
        cfg.write = _tams_write;
    } else {
        cfg.read = _tams_read;
    }
    el = audio_element_init(&cfg);

    AUDIO_MEM_CHECK(TAG, el, goto _tams_init_exit);
    audio_element_setdata(el, fatfs);
    return el;
_tams_init_exit:
    audio_free(fatfs);
    return NULL;
```

Les éléments audios créés par le développeur doivent s'intégrer dans un pipeline, comme le montre la **figure 3**. Pour cela `audio_element_init()` a besoin de certaines informations : outre les paramètres liés au thread, la routine définit également un ensemble de fonctions de rappel que le superviseur audio peut utiliser pour informer l'élément des événements qui se produisent.

À titre d'exemple, examinons la méthode `tams_open` qui se charge de l'initialisation de l'élément selon le schéma suivant :

```
static esp_err_t _tams_open(audio_element_
handle_t self) {
    audio_element_info_t info;
    audio_element_getinfo(self, &info);

    initTamsWorkerAlgo();

     return audio_element_setinfo(self,
&info);
}
```

Comme le présent algorithme n'a pas besoin des informations contenues dans `audio_element_info_t`, l'implémentation se limite à transmettre au pipeline les informations sur les paramètres.

Les événements `read` et `write` sont responsables de l'échange de données. Notre flux n'est utilisé qu'en tant que « point d'écriture » et ne peut pas être lu, il ne fait donc qu'émettre un message d'erreur dans le journal du système lorsqu'une tentative de lecture est effectuée :

```
static int _tams_read(audio_element_handle_t self,
char *buffer, int len,
TickType_t ticks_to_wait, void *context)
{
    ESP_LOGE(TAG, "CENSORED");
    return 0;
}
```

Les opérations d'écriture sont aussi implémentées en « transparent » et n'effectuent aucune opération réelle :

```
static int _tams_write(audio_element_handle_t self,
char *buffer, int len,
TickType_t ticks_to_wait, void *context) {
    return len;
}
```

L'étape suivante nécessite un tampon d'entrée structuré selon le format suivant :

```
int16_t DspBuf[4096];
```

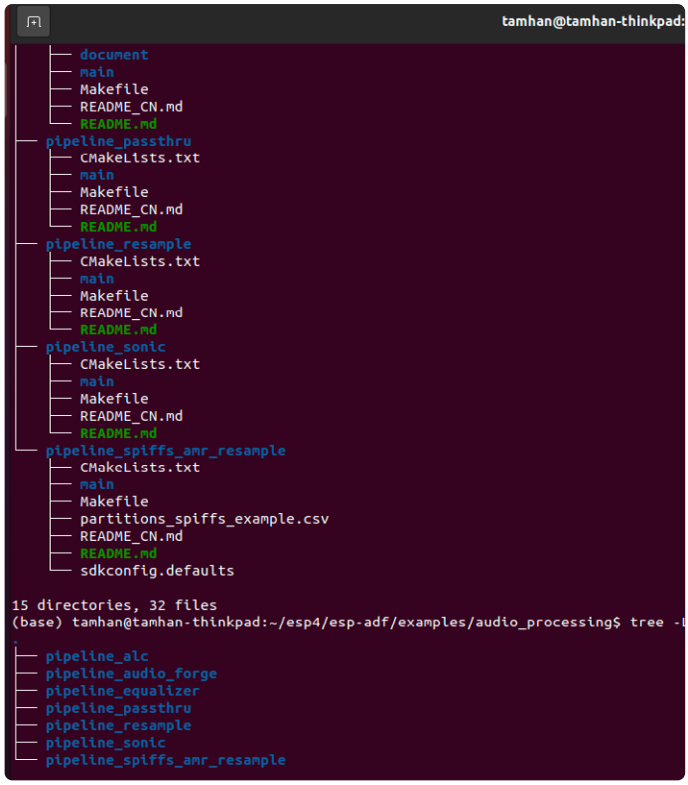

Figure 6. Tout un tas de pipelines à explorer !

`_tams_process()` se charge ensuite du traitement réel des données. La chose la plus importante ici est l'appel à `audio_element_input`, qui copie les informations dans le tampon de travail :

```
static int _tams_process(audio_element_handle_t self,
char *in_buffer, int in_len) {
    audio_element_input(self, (char *)DspBuf, in_len);
```

Comme exemple simple de traitement de données, j'ai inclus une simple opération de normalisation sur une plage de valeurs :

```
    for (int i=0 ; i< in_len ; i++) {
        y_cf[i] = ((float)DspBuf[i]) /  (float)32768;;
    }
```

Enfin, n'oubliez pas les utilités :

```
    int r_size = audio_element_input(self,
          in_buffer, in_len);
    int w_size = 0;
    if (r_size > 0) {
        w_size = audio_element_output(self,
          in_buffer, r_size);
    } else {
        w_size = r_size;
    }
    return w_size;
}
```

Opérations visant à libérer la mémoire précédemment affectée aux flux :

```
static esp_err_t _tams_close(audio_element_handle_t self)
{
    return ESP_OK;
}
static esp_err_t _tams_destroy(audio_element_handle_t
self) {
    tams_stream_t *fatfs =
    (tams_stream_t *)audio_element_getdata(self);
    audio_free(fatfs);
    return ESP_OK;
}
```

Le reste du code fonctionnel de la solution doit seulement arrêter le noyau en utilisant les instructions suivantes :

```
audio_pipeline_run(pipeline);
printf("halting.\n");
for(;;);
```

Notez que la fonction `worker` appelée périodiquement dans le pipeline peut faire ce qu'elle veut – dans ce projet, selon la configuration, elle communique avec des pairs I²C ou SPI en réponse aux données audios entrantes.

Plus d'éléments de pipelines

De nombreux autres exemples liés à l'audio existent dans l'ESP-ADF (**figure 6**). Allez dans *~/esp4/esp-adf/examples/audio_processing* – les algorithmes implémentent un grand nombre d'autres opérations en pipeline.

J'ai trouvé l'exemple *examples/audio_processing/pipeline_equalizer* particulièrement intéressant. Il implémente une fonction complète d'égaliseur graphique et est une classe incluse dans l'environnement ESP-ADF, donc ne nécessite pas de mathématiques avancées. Sa mise en œuvre s'effectue selon le schéma suivant, bien connu :

```
equalizer_cfg_t eq_cfg = DEFAULT_EQUALIZER_CONFIG();
int set_gain[] = {-13, -13, -13, -13, -13, -13, -13,
-13, -13, -13, -13, -13, -13, -13, -13, -13, -13, -13,
-13, -13};
eq_cfg.set_gain = set_gain;
equalizer = equalizer_init(&eq_cfg);
```

Pour en savoir plus sur les champs de données, consultez la rubrique [4].

L'intégration dans le flux de lecture s'effectue ensuite comme d'habitude avec `audio_pipeline_register()` :

```
audio_pipeline_register(pipeline, fatfs_stream_reader,
"file_read");
audio_pipeline_register(pipeline, wav_decoder, "wavdec");
audio_pipeline_register(pipeline,  equalizer,
"equalizer");
audio_pipeline_register(pipeline, i2s_stream_writer,
"i2s");
```

Lorsqu'une application atteint un certain degré de complexité, on arrive à un point où les traitements du signal faits maison ne suffisent plus. Espressif fournit ici de l'aide avec la bibliothèque

ESP-DSP disponible sous [5] : c'est une sorte de canevas qui fournit divers algorithmes DSP avec une grande stabilité algorithmique et diverses optimisations pour les différentes puces communes d'Espressif.

Grâce à l'organisation de sa structure de dossiers, chaque projet ESP32 est « directement » équipé pour accepter l'intégration de composants ESP-IDF supplémentaires. L'installation du DSP est donc facile. Dans un premier temps, créez une copie d'un projet :

```
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/examples$
cp -r  audio_processing/pipeline_equalizer/ tamsdsptest1
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/examples$
cd tamsdsptest1/
```

Créez-y un dossier portant le nom de *components*. Celui-ci accepte alors la version complète de la bibliothèque, qui peut être téléchargée sur GitHub :

```
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/examples/
tamsdsptest1$ mkdir components
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/examples/
tamsdsptest1$ cd components/
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/examples/
tamsdsptest1/components$ git clone https://github.com/
espressif/esp-dsp.git
```

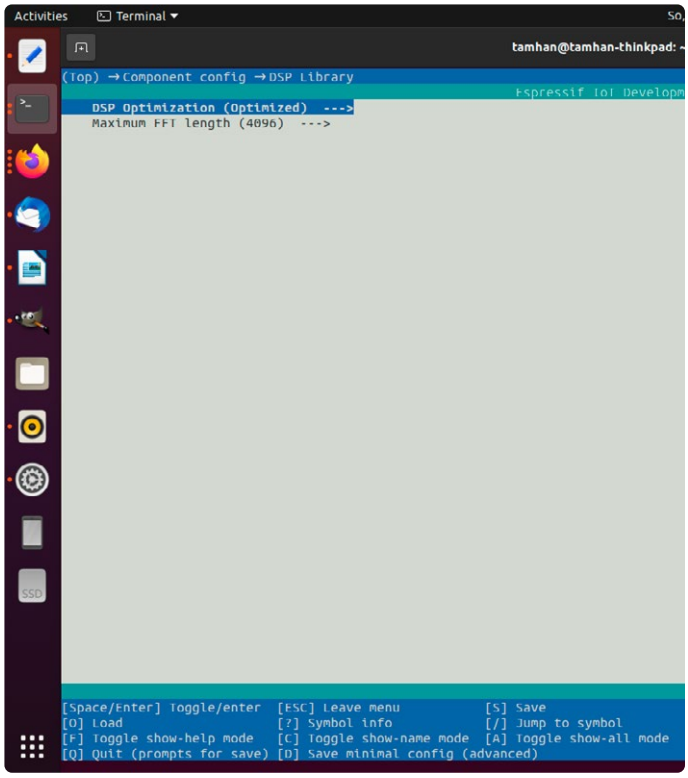

Figure 7. ESP-DSP s'imbrique sous (Top) → Component config → DSP Library dans le processus de compilation.

Lors de la prochaine exécution de la chaîne d'outils de compilation, vous verrez apparaître une nouvelle option permettant de configurer le comportement de la bibliothèque DSP, comme le montre la **figure 7.**

Résumé

Avec le support fourni par l'ESP-ADF, Espressif offre aux développeurs un environnement puissant et flexible pour simplifier la mise en œuvre d'applications audio. Grâce à lui, ma propre entreprise a déjà pu économiser plusieurs centaines d'heures de temps de développement, je peux fortement recommander son utilisation.

220600-04 — VF : Denis Lafourcade

Des questions, des commentaires ?

Envoyez un courriel à l'auteur à (tamhan@tamoggemon.com) ou contactez Elektor (redaction@elektor.fr).

À propos de l'auteur

L'ingénieur Tam Hanna est un développeur, auteur et journaliste indépendant (www.instagram.com/tam.hanna). Il développe des produits électroniques, des ordinateurs et des logiciels depuis plus de 20 ans maintenant. Pendant son temps libre, Tam conçoit et réalise des produits en impression 3D et, entre autres choses, se passionne pour le commerce et la dégustation de cigares haut de gamme.

LIENS

[1] Périphériques ESP-ADF : https://elektor.link/ESPPeripherals
[2] ESP-ADF Button Peripheral : https://elektor.link/ESPButtonPeripheral
[3] Référence API ESP-ADF : https://elektor.link/ESPAPIReference
[4] Égaliseur ESP-ADF : https://elektor.link/ESPEqualizer
[5] esp-dsp : https://github.com/espressif/esp-dsp
[6] Guide de démarrage de l'ESP32-LyraT : https://elektor.link/ESP32LyraT