Audiosignalen en de ESP32

de ESP-ADF-omgeving in de praktijk

Tam Hanna (Hongarije)

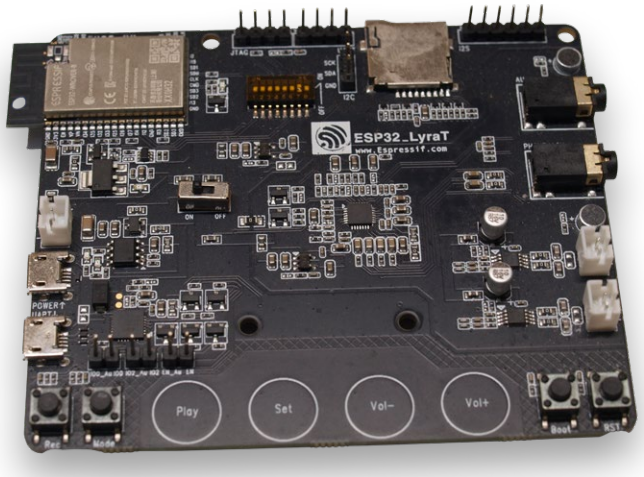
Het ontwikkelen van nieuwe consumentenelektronica is een echte uitdaging. Die producten hebben meestal een zeer korte levenscyclus, dus een snelle ontwikkeling is essentieel voor succes. Dat geldt zeker voor audiotoepassingen, waar algoritmes worden gebruikt voor het implementeren van verschillende standaard codecfuncties. Het Espressif Audio Development Framework (ESP-ADF) is een krachtig gereedschap met functies die tijd en inspanning besparen voor ontwikkelaars van zulke toepassingen.

Espressif's Audio Development Framework biedt een verzameling van algoritmen en codecs in een gestandaardiseerd formaat. Om een toepassing te ontwikkelen, hoeft de ontwerper alleen maar de juiste "software-IC's" aan elkaar te koppelen, zonder zich bezig te houden met de interne details van die bouwstenen.

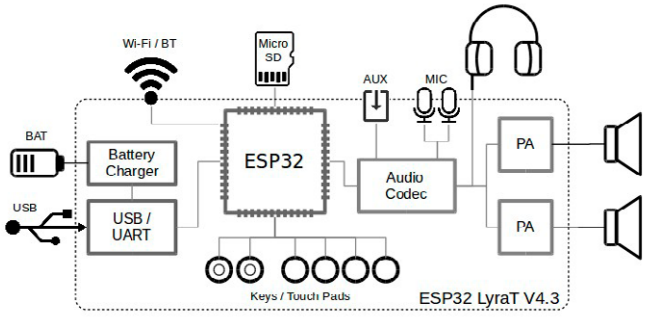
Het doel van dit artikel is om u enig begrip te geven van hoe ADF in de praktijk werkt. Ik heb Espressif's ADF-tool vaak gebruikt bij het implementeren van oplossingen volgens de ontwerpeisen van mijn klanten. Ik kan nu veel dingen met u delen, die ik graag had willen weten toen ik begon ADF te gebruiken.

Hardware to Go

Om te beginnen iets vanzelfsprekends. ESP-ADF is compatibel met alle ESP32-systemen. De library biedt een gestandaardiseerde driver-interface voor het "audiogedeelte" dat de data bewerkt.



Figuur 1: De kleine zwarte print bevat alles…



Figuur 2: …wat u nodig hebt om audiotoepassingen te ontwikkelen (bron: [6]).

Voordat u het hardwareontwerp gaat afronden, kunt u al veel van de software schrijven en testen met behulp van een standaard ontwikkelkaart. Espressif levert verschillende van die ontwikkelkaarten. Zelf gebruik ik graag het LyraT-board (**figuur 1 en 2**).

Het belangrijkste is het gedigitaliseerde “analog baseband”-signaal, dat wordt geleverd door de ES8388-chip (met behulp van twee analoge microfoons). De communicatie gaat via I2S en via een aantal gereserveerde pennen; het schema en de verbindingen zijn beschikbaar gesteld door Espressif en zijn eenvoudig te repliceren in het ontwerp van een apparaat. De chip is gemakkelijk verkrijgbaar bij UTSource en LCSC.

Bij het werken met de LyraT is het aantal beschikbare GPIO-lijnen van de ESP32 op de kaart tamelijk beperkt. Als het hardwareontwerp van het apparaat ingewikkelder wordt, kunnen dat er te weinig zijn. In zulke gevallen kan het nodig zijn om een tweede processor te overwegen voor de hardwarecommunicatie.

Installeren van ADF

Espressif levert ADF als een plugin voor de bestaande IDF-omgeving. ADF kan niet worden gebruikt met Arduino, en ook niet met enkele versies van Espressif’s eigen IDF.

De gemakkelijkste manier om de werkomgeving te installeren is de complete ESP-ADF-repository te downloaden. Om te beginnen gebruik ik de volgende commando’s om een subdirectory, *esp4*, aan te maken, waar de repository lokaal wordt opgeslagen:

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

Dan download ik de volledige repository met behulp van Git:

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

Aan de output (die hier niet is weergegeven), is te zien dat de ADF-repository verschillende referenties maakt. Daarom is downloaden via de in GitHub geïntegreerde browser niet mogelijk.

Er zit ook een versie van de Espressif IDF bij de ADF. Voor sommige componenten is de aanwezigheid van de omgevingsvariabele `ADF_PATH` nodig. Als u vanaf de commandoregel wilt werken aan ADF-toepassingen, moet u die omgevingsvariabele definiëren met de volgende commando’s:

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

De aanwezigheid van de variabele `ADF_PATH` beïnvloedt soms normale IDF-projecten, ik probeer dat meestal te vermijden door een ander terminalvenster te gebruiken voor “normaal” ESP32-werk.

Voor de geïntegreerde IDF moet het gebruikelijke installatiescript worden gedraaid voor de compilers en andere afhankelijkheden:

```
(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
```

Dan moet een script worden gedraaid voor het opzetten van de toolchain (de twee opeenvolgende punten zijn geen tikfout):

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

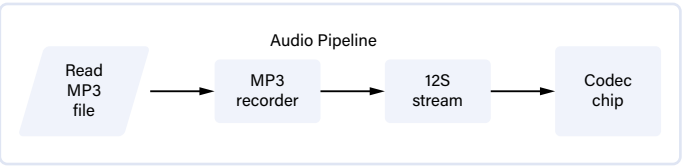
Het is aan te raden om nu een shell-script voor de parameterisatie aan te maken, zodat u met Bash automatisch kunt laten configureren als dat nodig is.

Softwarearchitectuur: de pijplijn

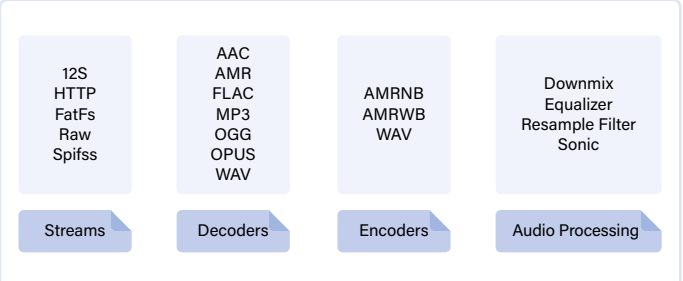
Na het downloaden en installeren van ADF is het tijd om na te denken over de theoretische structuur van het framework. Conform het concept van een “software-IC”, zoals ontwikkeld door The Stepstone Corporation, implementeert de ontwikkelaar met ADF in essentie een pijplijn die bestaat uit een reeks van verwerkingsstappen. In **figuur 3** ziet u een workflowmodel voor het implementeren van een MP3-speler.

In **figuur 4** ziet u de verschillende rollen en voorbeeldimplementaties die Espressif aanbiedt.

Het enige bijzondere aan het huidige systeem is dat het Unix-principe “alles is een file” hier ook van toepassing is op periferie.



Figuur 3: Deze pijplijn decodeert MP3’s (bron: [3]) .



Figuur 4: Pijplijnelementen zijn te verdelen in verschillende types (bron: [3]).

Met de API, die uitvoerig gedocumenteerd is onder [1] en het gemakkelijkst te begrijpen met behulp van de button-functies onder [2], kunt u "wrappers" aanmaken die u rechtstreeks kunt integreren in de pijplijn. Maar voor ontwikkelaars van toepassingen is dit alleen relevant omdat het idee om ook periferie te gebruiken als pijplijnelement soms niet vanzelfsprekend is.

API-documentatie voor de individuele pijplijnelementen is te vinden onder [3].

Softwarearchitectuur: bouw een MP3-speler

Laten we als praktische illustratie eens kijken naar één van de voorbeelden van Espressif: Als u een project wilt implementeren met behulp van ADF, kunt u het beste uitgaan van een soortgelijk, of minstens verwant, project tussen de voorbeelden. Ga naar de directory met voorbeelden met het commando:

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

In de volgende stappen zien we hoe een klassieke MP3-speler is te bouwen. Ga naar de map *~/esp4/esp-adf/examples/get-started/play_mp3_control/main* en open de file *play_mp3_control_example.c* met uw favoriete editor.

Het belangrijkste is het startpunt, waar enkele member-variabelen worden aangemaakt. Ons voorbeeld heeft een pijplijn-object nodig en twee element-handles voor de individuele elementen:

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

Aan de hardware wordt in ADF gerefereerd met de abstractie *Board*. De parameterisatie kiest u in het framework *Menuconfig*. De commando's in de achterliggende code verwerken voornamelijk de compilatieconstanten:

```
   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);
```

Het is interessant dat de HAL ook de volumeregeling bevat. Daarna wordt het pijplijnobject opgezet:

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

Dit zal ontwikkelaars met ervaring met ESP-IDF bekend voorkomen. Het pijplijnobject wordt gemaakt door een configuratiestruct mee te geven, dat dan van gebruiker naar gebruiker wordt doorgegeven.

Vervolgens worden de eigenlijke pijplijnelementen gegenereerd. Eerst de MP3-decoder:

```
   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);
```

Nu is de vraag (zeker als we kijken naar de pijplijn in figuur 3) hoe de data moet worden aangeleverd. Dat gaat met de methode `audio_element_set_read_cb`, die een functie als parameter krijgt. Deze functie heeft de volgende structuur:

```
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;
}
```

De callback levert de informatie die de codec moet verwerken via de buffer. De I2S-stream is dan eenvoudiger; die wordt doorgegeven van `AUDIO_STREAM_WRITER` en geparametriseerd als een output zodat de binnenkomende informatie naar de I2S-geluidshardware wordt geleid:

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

Het aanmaken van pijplijnelementen maakt niet automatisch dat ze worden gedeclareerd: Dat is ten behoeve van ontwikkelaars die meerdere pijplijnen tegelijk willen gebruiken. Het opzetten van de pijplijn vindt plaats door het registreren van de individuele delen, die elk worden geïdentificeerd door een string:

```
audio_pipeline_register(pipeline, mp3_decoder, "mp3");
audio_pipeline_register(pipeline, i2s_stream_writer,
"i2s");
```

Listing 1: Eventverwerking voor een MP3-speler.

```
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;
    }
}
```

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

De pijplijn wordt dan opgezet met behulp van een array waarin de individuele string-ID's in de juiste volgorde worden opgenomen. Voor de MP3-speler ontbreekt nu alleen nog het activeren van de pijplijn. Dat gaat als volgt:

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

Er is voor de MP3-speler ook nog een vrij uitgebreide eventverwerking om te reageren op verschillende gebeurtenissen (**listing 1**). De exacte eventverwerking is nu niet belangrijk, het is veel interessanter om de compilatie uit te voeren als test:

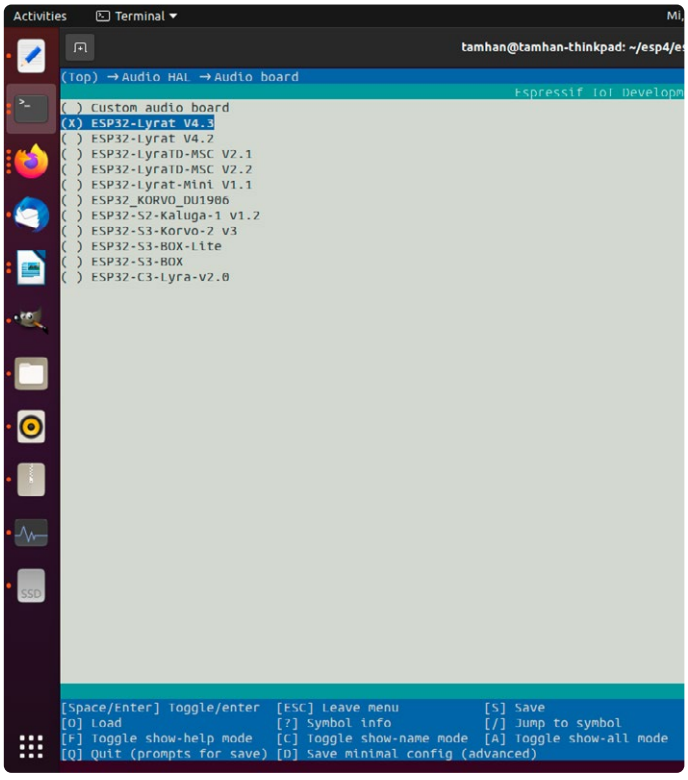
```
(base)tamhan@tamhan-thinkpad:~/esp4/esp-adf/examples/
get-started/play_mp3_control$ make build
```

Een opmerking hierover: om de privacy van mijn klanten te beschermen, heb ik bij het schrijven van dit artikel gewerkt op mijn reislaptop. De Ubuntu-versie die daarop draait werkt met versie 3.4.3 van CMake, en daarom kon ik idf.py niet draaien. In de praktijk is *idf.py* natuurlijk te prefereren boven het gebruik van `make`. En dat zal ook verplicht zijn in toekomstige versies van IDF.

Bij een maagdelijk project is de beloning voor het werk het *menuconfig*-scherm, dat enkele extra opties biedt vergeleken met dat van een gewoon IDF-project. De belangrijkste is de optie *(Top) Audio HAL Audio board*, waar u de board-configuratie kunt kiezen (**figuur 5**).

Na het opslaan begint het bouwproces op dezelfde manier als voor elk ander IDF-project. Het is heel belangrijk dat de kaart twee MicroUSB-poorten heeft: de POWER-poort is om de kaart te voeden en de UART-poort wordt alleen gebruikt voor datacommunicatie. Meestal gebruik ik gewoon twee aparte USB-kabels om mijn PC met de kaart te koppelen.

Het LyraT-board kan (in tegenstelling tot veel ander ontwikkelkaarten) niet automatisch opnieuw worden geprogrammeerd; zelfs voordat een verbinding kan worden opgebouwd, moet u de



Figuur 5: ADF breidt Menuconfig uit met enkele instellingen.

Boot-knop ingedrukt houden en dan kort op *RST* drukken. Dan ziet u als het goed is:

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

Als u niet op de juiste manier op de knoppen hebt gedrukt, ziet het systeem dit als een corrupt communicatiebericht… Probeer het opnieuw:

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

Als het flashen is gelukt, kunt u een koptelefoon aansluiten en opnieuw op *Reset* drukken. Als alles goed gegaan is, moet u een audio-track kunnen horen.

Vergeet niet dat het uitvoeren van het configuratiescript *export.sh* niet voldoende is om de omgeving te parameteriseren. U moet altijd eerst zorgen dat de omgevingsvariabele *ADF_PATH* goed is ingesteld:

```
(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'
. . .
```

Verwerken van inputdata

Het gebruik van de ESP-ADF-pijplijn is niet alleen zinvol bij het weergeven van geluid: We kunnen net zo goed berekeningen in de pijplijn zetten. Onlangs moest ik nog voor een klant microfoondata laten verwerken door een algoritme. De eigenlijke berekening zette ik in een pijplijnelement, waarvan ik de algemene structuur hier graag wil laten zien.

We beginnen met het initialiseren van de pijplijn, en omdat de audioinformatie binnenkomt via de I2S-bus, is een I2S-streamelement nodig in de eerste stap. Maar in dit geval bevat de configuratie nu de vlag AUDIO_STREAM_READER, die aangeeft dat het een input of een databron is:

```
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);
```

Naast de I2S-stream hebben we ook een element nodig waar het algoritme in zit. Dat wordt hier geïnitialiseerd:

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

De eigenlijke pijplijn wordt dan opgebouwd door strings toe te wijzen en die door te geven naar `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);
```

Nu kunnen we overgaan naar de headerfile die de elementen voor het gebruik van de "Tam"-taak bevat. Het belangrijkste is de configuratie-struct, die onder meer de meta-informatie voor FreeRTOS bevat (**listing 2**).

`TAMS_STREAM_CFG_DEFAULT` is een macro voor het aanmaken van de structs met default parameters. We hebben hier geen ruimte om de constanten weer te geven:

Listing 2: Configuratie van de pijplijnelementen.

```
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;
```

Listing 3: Initialisatie.

```
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, \
    . . .
}
```

Alleen voor `BUF_SIZE` maken we een uitzondering. De aangeleverde data is 16 bits breed, daarom hebben we de volgende constante nodig om blokken met een lengte van 1024 slots te verwerken:

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

Nu kunnen we overgaan naar de initialisatie, zie **listing 3**.

De waarde van `cfg.buffer_len` bepaalt het aantal datawoorden dat per run wordt geleverd. Voor de rest is er eigenlijk alleen standaard code voor de administratie:

```
    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;
```

De eigen audio-elementen van de ontwikkelaar moeten in een pijplijn worden geïntegreerd zoals figuur 3. `audio_element_init()` heeft daar wat informatie voor nodig: naast de thread-parameters zet de routine ook een aantal callbacks op om events te signaleren.

Zo is er bijvoorbeeld de methode `tams_open`, die zorgt voor de initialisatie van het element volgens het volgende schema:

```
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);
}
```

Omdat het huidige algoritme geen informatie in `audio_element_info_t`, nodig heeft is de implementatie beperkt tot het doorgeven van de parameterinformatie naar de pijplijn.

De events `read` en `write` zorgen voor de uitwisseling van gegevens. Onze stream wordt alleen gebruikt als een “write point” en kan niet worden gelezen, daarom genereert hij alleen een foutmelding in de logfile als iemand probeert hem uit te lezen:

```
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;
}
```

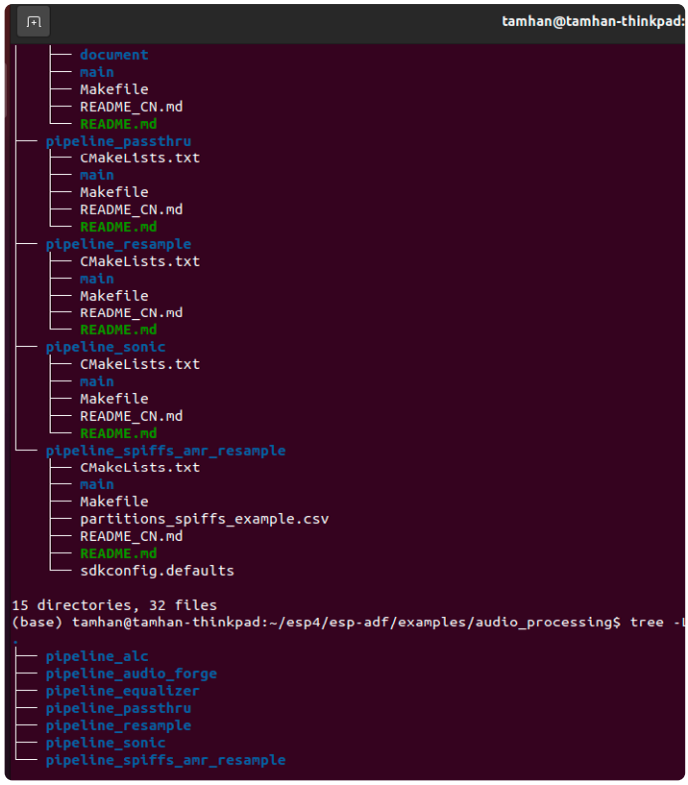
Schrijfoperaties zijn ook geïmplementeerd als “passthrough” en voeren geen echte operaties uit:

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

Voor de volgende stap is een inputbuffer van het volgende formaat nodig:

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

`_tams_process()` verzorgt de eigenlijke verwerking van de data.



Figuur 6: Er zijn heel veel pijplijnen om te verkennen!

Het belangrijkste hier is de call van `audio_element_input`, die de informatie naar de werkbuffer kopieert:

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

Als een eenvoudig voorbeeld van dataverwerking doe ik hier een eenvoudige normalisatie op een reeks van waarden:

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

Vergeet ook de administratie niet:

```
    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;
}
```

Operaties voor het vrijmaken van geheugen dat eerder was toegewezen voor streams:

```
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;
}
```

De rest van de code in de oplossing hoeft alleen nog de kern te stoppen met behulp van de volgende instructies:

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

De periodiek aangeroepen werkerfunctie in de pijplijn kan in feite doen wat hij wil. In dit project communiceert hij met I2C- of SPI-collega's als reactie op binnenkomende geluidsdata.

Meer pijplijnelementen

Er zijn veel meer audio-voorbeelden beschikbaar in de ESP-ADF (**figuur 6**). Ga naar *~/esp4/esp-adf/examples/audio_processing* . De algoritmen implementeren allerlei extra pijplijnoperaties.

Het voorbeeld *examples/audio_processing/pipeline_equalizer* vond ik bijzonder interessant. Dit implementeert een complete grafische equaliserfunctie en is een klasse in het ESP-ADF-raamwerk, dus heeft geen geavanceerde wiskunde nodig. Het gebruik ervan gaat op de volgende, bekende manier:

```
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);
```

Zie [4] voor meer over datavelden.

Integratie in de weergavestream gaat dan, zoals gewoonlijk, met `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");
```

Als een toepassing een bepaalde graad van complexiteit bereikt, schieten de aanwezige signaalverwerkingselementen tekort. Espressif levert dan hulp met de ESP-DSP-library, beschikbaar onder [5]:

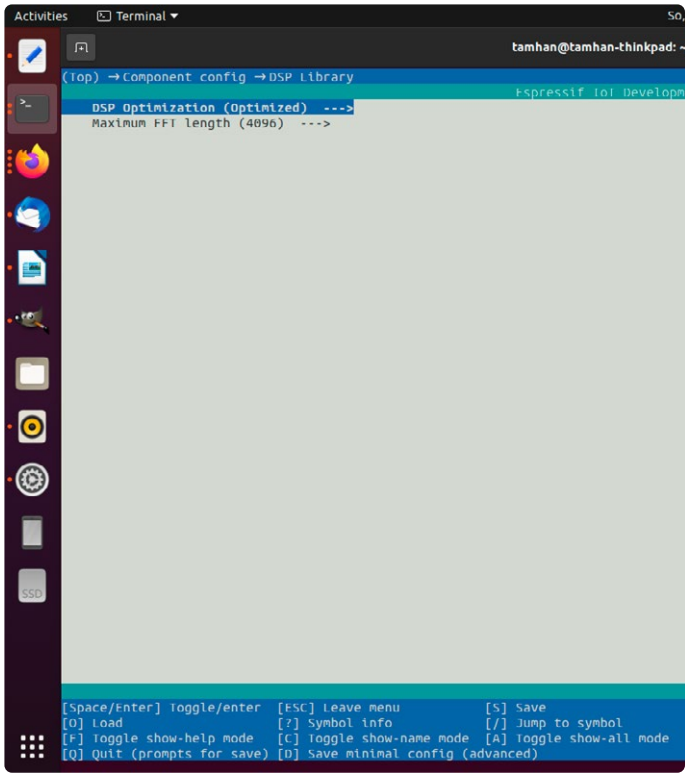
Het is een soort raamwerk met diverse DSP-algoritmen met een grote algoritmische stabiliteit en met optimalisaties voor de verschillende chips van Espressif.

Dankzij de mappenstructuur is elk ESP32-project "direct" ingericht voor integratie van extra ESP-IDF-componenten. De installatie van DSP is dus gemakkelijk. Maak eerst een kopie van een project:

```
(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/
```

Maak daar een map met de naam *components*. Daar komt dan de volledige versie van de library, die is te downloaden van 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
```



Figuur 7: ESP-DSP verschijnt onder (Top) Component config DSP Library in het compilatieproces.

Als u dan de compilatie-toolchain draait, ziet u een nieuwe optie voor het configureren van het gedrag van de DSP-library, zoals in **figuur 7**.

Samenvatting

Met de ondersteuning van ESP-ADF geeft Espressif ontwikkelaars een krachtig en flexibel raamwerk voor het vereenvoudigen van de implementatie van audiotoepassingen. Met die hulp heb ik in mijn eigen bedrijf al vele honderden uren aan ontwikkeltijd bespaard, ik kan het zeer aanbevelen.

220600-03

Vragen of opmerkingen?

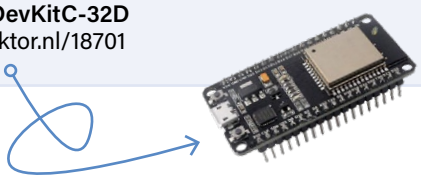
Als u vragen of opmerkingen hebt over dit artikel neem dan contact op met de auteur via tamhan@tamoggemon.com of met het redactieteam via redactie@elektor.com.

Over de auteur

Tam Hanna is freelance ontwikkelaar, auteur en journalist (www.instagram.com/tam.hanna). Hij ontwikkelt al meer dan 20 jaar elektronische producten, computers en software. In zijn vrije tijd ontwerpt en produceert hij 3D-geprinte oplossingen en hij is een liefhebber van goede sigaren.

Gerelateerde producten

> **ESP32-DevKitC-32D**
> www.elektor.nl/18701



WEBLINKS

[1] ESP-ADF Peripherals : https://elektor.link/ESPPeripherals
[2] ESP-ADF Button Peripheral: https://elektor.link/ESPButtonPeripheral
[3] ESP-ADF API Reference: https://elektor.link/ESPAPIReference
[4] ESP-ADF Equalizer: https://elektor.link/ESPEqualizer
[5] esp-dsp: https://github.com/espressif/esp-dsp
[6] ESP32-LyraT Getting Started Guide: https://elektor.link/ESP32LyraT