API DMA continue native de l'ESP-IDF pour l'ADC

Disponibilité des versions

L'API `adc_continuous` est apparue dans l'**ESP-IDF v4.4** (fin 2021) et a ensuite été largement améliorée :

- **ESP-IDF v4.4** : première version de l'API `adc_continuous`
- **ESP-IDF v5.0+** : API plus stable et aboutie
- **ESP-IDF v5.1+** : améliorations supplémentaires et corrections de bogues

Support du framework Arduino

État actuel : - **Arduino-ESP32 v2.x** (basé sur ESP-IDF 4.4) : API partiellement prise en charge, documentation limitée - **Arduino-ESP32 v3.x** (basé sur ESP-IDF 5.1+) : meilleure intégration de l'API ADC continue

Le framework Arduino-ESP32 encapsule l'ESP-IDF. En incluant les bons en-têtes, vous pouvez donc utiliser directement l'API native dans un projet Arduino.

Fonctionnement général de la nouvelle interface

Architecture de base

La nouvelle API fournit un mode ADC continu dédié avec un vrai support DMA, sans recourir à l'astuce I2S :

```
#include <Arduino.h>
#include "esp_adc/adc_continuous.h"

#define ADC_CHANNEL_INDEX ADC_CHANNEL_0   // À adapter selon le GPIO utilisé
#define ADC_ATTENUATION   ADC_ATTEN_DB_12 // L'atténuation 11 dB est dépréciée, 12 dB couvre ~3,3
#define SAMPLE_RATE_HZ    20000
#define FRAME_LENGTH      256             // La taille doit être un multiple de SOC_ADC_DIGI_RESU
#define NUM_CHANNELS      1

static adc_continuous_handle_t s_adc_handle = nullptr;

static void init_adc_continuous() {
  const adc_continuous_handle_cfg_t handle_cfg = {
    .max_store_buf_size = 1024,
    .conv_frame_size = FRAME_LENGTH,
  };
  ESP_ERROR_CHECK (adc_continuous_new_handle (&handle_cfg, &s_adc_handle));

  adc_digi_pattern_config_t pattern[NUM_CHANNELS] = {};
  pattern[0].atten = ADC_ATTENUATION;
  pattern[0].channel = ADC_CHANNEL_INDEX;
  pattern[0].unit = ADC_UNIT_1;
  pattern[0].bit_width = ADC_BITWIDTH_12;

  const adc_continuous_config_t adc_config = {
```

```
    .pattern_num = NUM_CHANNELS,
    .adc_pattern = pattern,
    .sample_freq_hz = SAMPLE_RATE_HZ,
    .conv_mode = ADC_CONV_SINGLE_UNIT_1,
    .format = ADC_DIGI_OUTPUT_FORMAT_TYPE1, // Passer en TYPE2 sur ESP32-C6/C5 si nécessaire
  };
  ESP_ERROR_CHECK (adc_continuous_config (s_adc_handle, &adc_config));
  ESP_ERROR_CHECK (adc_continuous_start (s_adc_handle));
}

void setup() {
  Serial.begin (115200);
  init_adc_continuous();
}

void loop() {
  uint8_t buffer[FRAME_LENGTH] = {0};
  uint32_t bytes_read = 0;

  const esp_err_t ret = adc_continuous_read (s_adc_handle, buffer, sizeof (buffer), &bytes_read,

  if (ret == ESP_OK) {
    for (uint32_t offset = 0; offset + SOC_ADC_DIGI_RESULT_BYTES <= bytes_read; offset += SOC_ADC
      const adc_digi_output_data_t *sample = reinterpret_cast<const adc_digi_output_data_t *> (&b
      const uint32_t channel = sample->type1.channel;
      const uint32_t value = sample->type1.data;
      Serial.printf ("CH[%lu] = %lu\n", channel, value);
    }
  }
  else if (ret != ESP_ERR_TIMEOUT) {
    Serial.printf ("adc_continuous_read failed: %s\n", esp_err_to_name (ret));
  }

  delay (100);
}
```

Exemple complet avec plusieurs canaux

```
#include "esp_adc/adc_continuous.h"

#define SAMPLE_RATE       20000
#define READ_LEN          256
#define NUM_CHANNELS      2

static adc_continuous_handle_t handle = NULL;
static TaskHandle_t task_handle = NULL;

// Callback appelé quand une trame de conversion est prête
static bool IRAM_ATTR adc_conv_done_cb(adc_continuous_handle_t handle,
```

```
                                        const adc_continuous_evt_data_t *edata,
                                        void *user_data) {
  BaseType_t mustYield = pdFALSE;
  vTaskNotifyGiveFromISR(task_handle, &mustYield);
  return (mustYield == pdTRUE);
}

void init_adc_continuous() {
  adc_continuous_handle_cfg_t adc_config = {
    .max_store_buf_size = 1024,
    .conv_frame_size = READ_LEN,
  };
  ESP_ERROR_CHECK(adc_continuous_new_handle(&adc_config, &handle));

  // Configuration de plusieurs canaux
  adc_digi_pattern_config_t adc_pattern[NUM_CHANNELS];

  // Canal 0 - GPIO36
  adc_pattern[0].atten = ADC_ATTEN_DB_11;
  adc_pattern[0].channel = ADC_CHANNEL_0;
  adc_pattern[0].unit = ADC_UNIT_1;
  adc_pattern[0].bit_width = ADC_BITWIDTH_12;

  // Canal 3 - GPIO39
  adc_pattern[1].atten = ADC_ATTEN_DB_11;
  adc_pattern[1].channel = ADC_CHANNEL_3;
  adc_pattern[1].unit = ADC_UNIT_1;
  adc_pattern[1].bit_width = ADC_BITWIDTH_12;

  adc_continuous_config_t dig_cfg = {
    .pattern_num = NUM_CHANNELS,
    .adc_pattern = adc_pattern,
    .sample_freq_hz = SAMPLE_RATE,
    .conv_mode = ADC_CONV_SINGLE_UNIT_1,
    .format = ADC_DIGI_OUTPUT_FORMAT_TYPE1,
  };

  ESP_ERROR_CHECK(adc_continuous_config(handle, &dig_cfg));

  adc_continuous_evt_cbs_t cbs = {
    .on_conv_done = adc_conv_done_cb,
  };
  ESP_ERROR_CHECK(adc_continuous_register_event_callbacks(handle, &cbs, NULL));

  ESP_ERROR_CHECK(adc_continuous_start(handle));
}

void adc_task(void *param) {
  uint8_t result[READ_LEN] = {0};
```

```
  uint32_t ret_num = 0;

  while (1) {
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

    while (1) {
      esp_err_t ret = adc_continuous_read(handle, result, READ_LEN, &ret_num, 0);

      if (ret == ESP_OK) {
        for (int i = 0; i < ret_num; i += SOC_ADC_DIGI_RESULT_BYTES) {
          adc_digi_output_data_t *p = (adc_digi_output_data_t*)&result[i];

          uint32_t chan = p->type1.channel;
          uint32_t data = p->type1.data;

          Serial.printf("CH[%d]: %d\n", chan, data);
        }
      } else if (ret == ESP_ERR_TIMEOUT) {
        break;
      }
    }
  }
}

void setup() {
  Serial.begin(115200);

  task_handle = xTaskGetCurrentTaskHandle();
  init_adc_continuous();

  xTaskCreate(adc_task, "adc_task", 4096, NULL, 2, &task_handle);
}

void loop() {
  delay(1000);
}
```

Fonctionnement interne

1. Flux matériel

`Périphérique ADC → Contrôleur DMA → Buffer circulaire → Application`

L'API gère automatiquement : - **Descripteurs DMA** : chaînés et configurés sans intervention manuelle - **Buffer circulaire** : stockage continu des données - **Interruptions** : notification quand des données sont disponibles

2. Format des échantillons

Chaque mesure est codée dans une structure (format TYPE1 sur ESP32) :

```c
typedef struct {
    union {
        struct {
            uint16_t data:     12;  // Valeur ADC (12 bits)
            uint16_t channel:   4;  // Numéro de canal
        } type1;
        uint16_t val;
    };
} adc_digi_output_data_t;
```

3. Modes de conversion

- **ADC_CONV_SINGLE_UNIT_1** : uniquement ADC1
- **ADC_CONV_SINGLE_UNIT_2** : uniquement ADC2 (ESP32 classique)
- **ADC_CONV_BOTH_UNIT** : ADC1 et ADC2 en multiplexage temporel
- **ADC_CONV_ALTER_UNIT** : alternance ADC1/ADC2

4. Gestion des buffers

```
Chaîne DMA :
[Buf1] → [Buf2] → [Buf3] → [Buf4] → [Buf1] (cercle)
  ↓       ↓       ↓       ↓
Buffer interne (max_store_buf_size)
  ↓
Lecture via adc_continuous_read()
```

Différences clés face à la méthode I2S

Caractéristique	Astuce I2S + ADC	API native `adc_continuous`
Complexité	Configuration verbeuse	API dédiée, plus claire
Multi-canaux	Démultiplexage manuel	Support intégré
Performances	Correctes	Optimisées pour l'ADC
Conflit Wi-Fi	Même conflit (ADC2)	Même restriction
Callbacks	À coder soi-même	Callbacks fournis
Gestion buffer	Manuelle	Buffer circulaire automatique

Avantages de la nouvelle API

1. **Conçue pour l'ADC** : évite le détournement du périphérique I2S audio
2. **Multi-canaux natif** : plan de conversion configurable
3. **Code plus propre** : moins de code d'initialisation à écrire
4. **Abstraction supérieure** : inutile de connaître les détails I2S
5. **Approche événementielle** : callbacks optionnels pour un traitement sans polling
6. **Buffering automatique** : gestion interne du ring buffer

Vérifier la version Arduino-ESP32

```c
void setup() {
  Serial.begin(115200);
```

```
  Serial.printf("Arduino-ESP32 version : %s\n", ARDUINO_ESP32_GIT_DESC);
  Serial.printf("Version ESP-IDF : %s\n", esp_get_idf_version());
}
```

Si vous utilisez une version Arduino-ESP32 plus ancienne, vous pouvez continuer avec l'approche I2S, ou passer à la v3.x pour profiter pleinement de l'API ADC continue.

Notes spécifiques à l'ESP32-C6

L'ESP32-C6 exploite la même API `adc_continuous`, mais son matériel diffère de l'ESP32 historique :

- Un seul ADC SAR est présent, `ADC_UNIT_1` est donc l'unique unité disponible. Il faut toujours choisir `ADC_CONV_SINGLE_UNIT_1` (les autres modes ne s'appliquent pas).
- Le contrôleur numérique expose 8 canaux externes. Les macros classiques pointent directement vers les GPIO correspondant :

```
ADC_CHANNEL_0 -> GPIO0
ADC_CHANNEL_1 -> GPIO1
ADC_CHANNEL_2 -> GPIO2
ADC_CHANNEL_3 -> GPIO3
ADC_CHANNEL_4 -> GPIO4
ADC_CHANNEL_5 -> GPIO5
ADC_CHANNEL_6 -> GPIO6
ADC_CHANNEL_7 -> GPIO7
```

- Certaines broches (par exemple GPIO0) sont également utilisées pour le boot ou partagées avec l'USB/UART. Tenez compte des contraintes de démarrage quand vous connectez des signaux analogiques.
- L'astuce I2S "ADC built-in" n'est pas implémentée sur ESP32-C6. L'API `adc_continuous` (ou le driver bas niveau `adc_digi`) constitue donc la voie privilégiée pour la capture continue avec DMA.