ESP32forth et la librairie RMT 2.0

Marc PETREMANN

Version 1.0 - 08/02/26

Table des matières

Préambule

Le but de ce manuel est de vous donner toutes les informations, ainsi que des exemples pratiques, pour exploiter pleinement la librairie RMT 2.0.

Le passage au **RMT 2.0** (introduit avec l'ESP-IDF v5) marque une transition d'un driver "statique" vers une architecture **dynamique et orientée objet**. Auparavant, on manipulait des numéros de canaux fixes (0-7) et des registres de diviseurs d'horloge manuels via `rmt_set_clk_div`. Désormais, tout repose sur des **Handles** (`rmt_channel_handle_t`) alloués dynamiquement par le système, garantissant une meilleure portabilité entre les puces (ESP32, S3, C3).

La gestion des signaux a également radicalement changé : on ne remplit plus simplement un tableau d'items, on utilise des **Encodeurs** (`RMT Encoder`) qui traduisent les données brutes en ondes en temps réel. Cette version introduit aussi le **Sync Manager** pour synchroniser parfaitement plusieurs canaux, et une intégration native du **DMA** pour libérer le processeur lors de transferts massifs. En résumé, le RMT 2.0 est plus complexe à initialiser, mais beaucoup plus robuste, flexible et économe en ressources CPU pour les protocoles complexes.

Les structures

rmt_sync_manager_config_t

La structure **`rmt_sync_manager_config_t`** est une nouveauté majeure du driver RMT 2.0 (Next Gen).

Elle sert à une chose précise : **la synchronisation de plusieurs canaux**.

Dans l'ancien driver, il était très difficile de déclencher l'envoi de deux signaux sur deux pins différentes exactement au même moment. Avec le **Sync Manager**, vous pouvez regrouper plusieurs handles de canaux TX et leur dire : "Partez tous ensemble sur le prochain top horloge".

C'est indispensable si vous construisez :

- Un contrôleur de matrice de LEDs géante (plusieurs rubans en parallèle).
- Un protocole de communication parallèle propriétaire.
- Une commande de moteurs pas à pas où les impulsions doivent être parfaitement alignées.

rmt_transmit_config_t

rmt_tx_channel_config_t

rmt_tx_event_callbacks_t

Les mots du vocabulaire rmt

Liste de tous les mots définis :

```
rmt_new_tx_channel rmt_enable rmt_transmit rmt-builtins
```

rmt_del_sync_manager

rmt_disable (handle --)

rmt_enable (handle --)

Le mot **`rmt_enable`** est une étape de transition indispensable dans le cycle de vie d'un canal RMT 2.0. Elle permet de faire passer le canal de l'état "Configuré" à l'état "Prêt à émettre".

Lorsque vous créez un canal avec `rmt_new_tx_channel`, le système alloue les ressources (mémoire, canal matériel), mais le périphérique est maintenu dans un état de **basse consommation**.

`rmt_enable` effectue les actions suivantes :

- **Active l'horloge** interne du périphérique RMT.
- **Alimente le bloc matériel** (Power Domain).
- **Autorise les interruptions** liées à ce canal.
- **Place le GPIO** dans son état initial (`init_level`).

Il est important de comprendre où elle se situe dans votre code Forth :

1. **`rmt_new_tx_channel`** : Création (Le canal est **IDLE** / Inactif).
2. **`rmt_enable`** : Activation (Le canal est **READY** / Prêt).
3. **`rmt_transmit`** : Action (Le canal est **BUSY** / En cours d'envoi).
4. **`rmt_disable`** : Sommeil (Le canal repasse en **IDLE** pour économiser l'énergie).

Cette séparation permet une gestion fine de l'énergie. Sur un ESP32 alimenté par batterie, vous pouvez créer vos canaux au démarrage, mais ne les "activer" (`enable`) que juste avant d'envoyer une trame, puis les "désactiver" (`disable`) immédiatement après pour couper la consommation des horloges.

Si vous tentez d'appeler `rmt_transmit` sur un canal qui n'a pas été "enabled", la fonction retournera une erreur de type `ESP_ERR_INVALID_STATE` (0x103) et rien ne sortira sur votre broche GPIO.

rmt_new_sync_manager

rmt_new_tx_channel (rmtTxConfig retHandle -- fl)

esp_err_t rmt_new_tx_channel(const rmt_tx_channel_config_t *config, rmt_channel_handle_t *ret_chan);

Le mot **`rmt_new_tx_channel`** est le point d'entrée indispensable de la nouvelle API RMT (v2.0 / Next Gen) présente dans votre Core 3.3.5. Il remplace l'ancien `rmt_config`.

Son rôle est simple : il demande au système d'allouer une ressource matérielle RMT libre et de lui appliquer votre configuration.

Les paramètres :

1. **`config`** : Un pointeur vers la structure que nous avons définie en Forth. Elle contient le GPIO, la résolution, et les fameux "flags".
2. **`ret_chan`** : C'est ici que la fonction va écrire le "Handle" (l'identifiant) du canal qu'elle vous a attribué. C'est ce handle que vous utiliserez pour toutes les autres fonctions (envoyer des données, arrêter le canal, etc.).

Contrairement à l'ancienne version où vous choisissiez manuellement `RMT_CHANNEL_0`, cette fonction est plus intelligente :

- **Gestion des ressources** : Elle cherche un canal libre (l'ESP32 en a 8, mais le S3 n'en a que 4).
- **Arbitrage d'horloge** : Elle vérifie que la source d'horloge demandée est compatible avec les autres canaux déjà ouverts.
- **Initialisation du GPIO** : Elle configure le multiplexeur interne (GPIO Matrix) pour relier le périphérique RMT à votre broche physique.
- **Protection mémoire** : Elle prépare l'accès à la RAM RMT (les blocs de symboles).

Le passage à cette fonction marque la fin du modèle "statique".

- **Avant** : On configurait un canal fixe, souvent avec des variables globales rigides.
- **Maintenant** : C'est une approche **dynamique**. On peut créer et supprimer des canaux à la volée, ce qui est beaucoup plus propre pour un langage interactif comme Forth.

Une fois le canal créé avec cette fonction, il est en état "IDLE" (repos). Pour envoyer un signal, vous ne pourrez plus envoyer de simples "items" directement ; vous devrez obligatoirement créer un **Encoder** et l'attacher à ce handle.

rmt_sync_reset

rmt_transmit

esp_err_t rmt_transmit(rmt_channel_handle_t tx_channel, rmt_encoder_handle_t encoder, const void *payload, size_t payload_bytes, const rmt_transmit_config_t *config);

rmt_tx_register_event_callbacks

rmt_tx_switch_gpio

rmt_tx_wait_all_done

Exemples RMT :
https://github.com/espressif/arduino-esp32/tree/master/libraries/ESP32/examples/RMT

RMT Write Neo Pixel

```
// Copyright 2023 Espressif Systems (Shanghai) PTE LTD
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at

//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

/**
 * @brief This example demonstrates usage of RGB LED driven by RMT
 *
 * The output is a visual WS2812 RGB LED color moving in a 8 x 4 LED matrix
 * Parameters can be changed by the user. In a single LED circuit, it will just blink.
 */

// The effect seen in (Espressif devkits) ESP32C6, ESP32H2, ESP32C3, ESP32S2 and ESP32S3
is like a Blink of RGB LED
#ifdef PIN_NEOPIXEL
#define BUILTIN_RGBLED_PIN   PIN_NEOPIXEL
#else
#define BUILTIN_RGBLED_PIN   21   // ESP32 has no builtin RGB LED (PIN_NEOPIXEL)
#endif

#define NR_OF_LEDS   8*4
#define NR_OF_ALL_BITS 24*NR_OF_LEDS

//
// Note: This example uses Neopixel LED board, 32 LEDs chained one
//      after another, each RGB LED has its 24 bit value
//      for color configuration (8b for each color)
//
//      Bits encoded as pulses as follows:
//
//      "0":
//         +-------+              +--
//         |       |              |
//         |       |              |
//         |       |              |
//      ---|       |--------------|
//         +       +              +
//         | 0.4us |   0.85 0us   |
//
//      "1":
//         +-------------+       +--
//         |             |       |
//         |             |       |
//         |             |       |
//         |             |       |
//      ---+             +-------+
//         |    0.8us    | 0.4us |

rmt_data_t led_data[NR_OF_ALL_BITS];

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

```
    if (!rmtInit(BUILTIN_RGBLED_PIN, RMT_TX_MODE, RMT_MEM_NUM_BLOCKS_1, 10000000)) {
        Serial.println("init sender failed\n");
    }
    Serial.println("real tick set to: 100ns");
}

int color[] =  { 0x55, 0x11, 0x77 };  // Green Red Blue values
int led_index = 0;

void loop() {
    // Init data with only one led ON
    int led, col, bit;
    int i=0;
    for (led=0; led<NR_OF_LEDS; led++) {
        for (col=0; col<3; col++ ) {
            for (bit=0; bit<8; bit++){
                if ( (color[col] & (1<<(7-bit))) && (led == led_index) ) {
                    led_data[i].level0 = 1;
                    led_data[i].duration0 = 8;
                    led_data[i].level1 = 0;
                    led_data[i].duration1 = 4;
                } else {
                    led_data[i].level0 = 1;
                    led_data[i].duration0 = 4;
                    led_data[i].level1 = 0;
                    led_data[i].duration1 = 8;
                }
                i++;
            }
        }
    }
    // make the led travel in the pannel
    if ((++led_index)>=NR_OF_LEDS) {
        led_index = 0;
    }
    // Send the data and wait until it is done
    rmtWrite(BUILTIN_RGBLED_PIN, led_data, NR_OF_ALL_BITS, RMT_WAIT_FOR_EVER);
    delay(100);
}
```

Le projet WS2812

Le projet consiste à piloter une couronne de 60 LEDs. Pourquoi 60 LEDs ? Tout simplement parce qu’il y a 60 minutes dans une heure, ou 60 secondes dans une minute. Le challenge est donc de pîloter ces LEDs pour s’en servir comme horloge analogique !