

# ESP32forth et la librairie RMT 2.0

Marc PETREMANN



Version 1.0 - 09/02/26

# Table des matières

<b>Préambule.....</b>	<b>3</b>
<b>Les structures.....</b>	<b>4</b>
rmt_sync_manager_config_t.....	4
rmt_transmit_config_t.....	4
rmt_tx_channel_config_t.....	5
rmt_tx_event_callbacks_t.....	6
<b>Les mots du vocabulaire rmt.....</b>	<b>7</b>
rmt_apply_carrier ( handle freq duty_percent -- err ).....	7
rmt_del_channel ( handle -- err ).....	7
rmt_del_sync_manager ( sync_handle -- err ).....	8
rmt_disable ( handle -- err ).....	9
rmt_enable ( handle -- err ).....	10
rmt_new_sync_manager.....	10
rmt_new_rx_channel ( config -- err ).....	10
rmt_new_tx_channel ( rmtTxConfig retHandle -- fl ).....	11
rmt_receive ( rx_channel buffer bufferSize config -- err ).....	12
rmt_rx_register_event_callbacks.....	13
rmt_wait_rx_done ( handle timeout_ms -- err ).....	14
rmt_sync_reset ( sync_handle -- err ).....	14
rmt_transmit ( handle encoder payload len config -- err ).....	15
rmt_tx_register_event_callbacks.....	16
rmt_tx_switch_gpio.....	16
rmt_tx_wait_all_done.....	16
<b>RMT Write Neo Pixel.....</b>	<b>17</b>
<b>Le projet WS2812.....</b>	<b>19</b>
Structure rmt_tx_channel_config_t.....	19
Les Timings WS2812B (Internes à l'encodeur).....	19
Configuration de la transmission (rmt_transmit_config_t).....	19

# 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 économique en ressources CPU pour les protocoles complexes.

**Note** – documentation détaillée :

<https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/rmt.html#rmt-resource-allocation>

# 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

La structure **rmt\_transmit\_config\_t** est une petite structure de configuration utilisée uniquement au moment de l'appel à **rmt\_transmit**. Elle permet de définir **comment** le message spécifique doit être envoyé (répétition, comportement de fin, etc.).

Dans le driver RMT 2.0 (IDF v5.x), elle est très légère. Voici son anatomie :

Champ C	Type	Description
<b>loop_count</b> int		Nombre de répétitions du message. <ul style="list-style-type: none"><li>• 0 : Envoi simple (défaut).</li><li>• -1 : Boucle infinie.</li><li>• &gt;0 : Nombre spécifique de répétitions.</li></ul>
<b>flags</b>	struct	Contient des réglages spécifiques sous forme de bits.

À l'intérieur du champ **flags**, on trouve principalement :

- **eot\_zealous** (1 bit) :
  - Si mis à 1 (true), le canal forcera le signal à l'état inactif immédiatement après le dernier symbole.
  - Si mis à 0, il peut y avoir une légère latence selon l'état du matériel.
  - *Utilité* : Crucial pour les protocoles très sensibles au timing de fin, comme certaines bandes de LEDs ou l'infrarouge.

Si vous voulez manipuler cette structure directement depuis Forth, voici comment la définir pour qu'elle corresponde à l'alignement mémoire 32 bits du processeur :

Extrait de code

```
struct rmt_transmit_config_t
    u32 field ->loop_count
    u32 field ->flags_packed \ Bit 0 est eot_zealous
end-struct
```

Dans 90% des cas, on souhaite envoyer une trame une seule fois sans réglages exotiques. C'est pour cela que la fonction `rmt_transmit` accepte un pointeur **NULL** (ou `0` en Forth) à la place de cette structure.

**Conseil pratique :** Si vous n'avez pas besoin de faire boucler votre signal (comme pour envoyer un code Infrarouge en continu), ne vous embêtez pas à créer cette structure. Passez simplement `0` comme dernier argument à votre macro `rmt_transmit`.

Si vous voulez envoyer un signal en boucle 5 fois :

```
variable ma-config
rmt_transmit_config_t allot ma-config !

5 ma-config ->loop_count !
0 ma-config ->flags_packed ! \ Pas de flags particuliers

\ Passage à la macro :
\ handle encoder payload len ma-config rmt_transmit
```

## rmt\_tx\_channel\_config\_t

La structure `rmt_tx_channel_config_t` est le "plan de fabrication" de votre canal de transmission. C'est elle qui définit les caractéristiques matérielles permanentes du canal au moment de sa création avec `rmt_new_tx_channel`.

Voici le détail des champs pour le **Core 3.x (RMT 2.0)**, car ils diffèrent grandement des anciennes versions.

Configuration Matérielle (Les bases) :

- **gpio\_num (int)** : Le numéro du GPIO physique où sortira le signal.
- **clk\_src (enum)** : La source d'horloge. Généralement mis à `0` (`RMT_CLK_SRC_DEFAULT`) pour utiliser l'horloge système (APB).
- **resolution\_hz (uint32\_t)** : **Le champ le plus important.** C'est la fréquence de votre "tick" d'horloge.
  - *Exemple* : Si vous mettez `10 000 000` (10 MHz), chaque unité de temps de vos futurs signaux vaudra  $1/107=100\text{ns}$ .

Gestion de la Mémoire et File d'attente :

- **mem\_block\_symbols (size\_t)** : Taille du tampon mémoire interne (en nombre de symboles RMT). Plus ce chiffre est grand, plus vous pouvez envoyer de longues trames sans interruption.
- **trans\_queue\_depth (size\_t)** : Nombre de transmissions que vous pouvez "mettre en attente" (piler) avant que `rmt_transmit` ne devienne bloquant.

- **intr\_priority** (int) : Priorité de l'interruption matérielle (0 pour laisser le système choisir).

Le bloc des "Flags" (Structure compacte), ces champs sont regroupés dans un seul mot de 32 bits en mémoire :

- **invert\_out** : Inverse le signal (le High devient Low et inversement).
- **with\_dma** : Indique si le canal doit utiliser le contrôleur DMA (Direct Memory Access). Indispensable pour les très longues trames (milliers de LEDs).
- **io\_loop\_back** : Mode de test où la sortie est réinjectée en interne vers l'entrée.
- **io\_od\_mode** : Configure la sortie en "Open Drain" (Collecteur ouvert).
- **allow\_pd** : Autorise la mise hors tension du périphérique en mode veille profonde pour économiser l'énergie.
- **init\_level** : Définit si la pin est à 0 ou 1 quand le canal est au repos.

Pour que votre **struct** Forth fonctionne avec le C++, voici l'ordre exact que vous devez respecter dans votre définition :

Champ Forth	Type C	Taille
->gpio_num	int	4 octets
->clk_src	int	4 octets
->resolution_hz	uint32_t	4 octets
->mem_block_symbols	size_t	4 octets
->trans_queue_depth	size_t	4 octets
->intr_priority	int	4 octets
->flags_packed	bitfield	4 octets

**Attention :** Si vous oubliez le champ **intr\_priority** ou si vous séparez les **flags**, le décalage mémoire sera faux et **rmt\_new\_tx\_channel** recevra des données incohérentes (comme une résolution de 0 Hz ou un GPIO impossible).

## rmt\_tx\_event\_callbacks\_t

# Les mots du vocabulaire rmt

Liste de tous les mots définis :

```
rmt_new_rx_channel rmt_new_tx_channel rmt_disable rmt_enable rmt_transmit  
rmt_receive rmt_apply_carrier rmt_rx_wait_done rmt_delete rmt_del_sync_manager  
rmt_rx_register_event_callbacks rmt_sync_reset rmt_new_ws2812_encoder  
rmt_builtins
```

## rmt\_apply\_carrier ( handle freq duty\_percent -- err )

Le mot **rmt\_apply\_carrier** est l'outil qui permet de moduler votre signal RMT avec une **fréquence porteuse**.

C'est une fonction essentielle si vous travaillez sur de la **domotique Infrarouge (IR)**. Les récepteurs IR (comme le célèbre TSOP) ne réagissent pas à une simple lumière continue, mais à une lumière qui clignote très vite (généralement à **38 kHz**).

Au lieu d'avoir un signal "carré" simple (High ou Low), la porteuse va découper chaque état "High" en une multitude de petites impulsions ultra-rapides.

- **Sans porteuse** : Le GPIO reste à 3.3V pendant toute la durée du symbole "High".
- **Avec porteuse** : Le GPIO oscille (0V -> 3.3V -> 0V) à la fréquence choisie (ex: 38 000 fois par seconde) pendant la durée du symbole "High".

Pour utiliser cette fonction, vous devez lui passer une structure qui définit les règles du "clignotement" :

Champ	Type	Description
<b>frequency_hz</b>	uint32_t	La fréquence de la porteuse (ex: 38000 pour l'IR).
<b>duty_cycle</b>	float	Le rapport cyclique (souvent 0.33 ou 0.5).
<b>flags</b>	struct	<b>polarity_active_low</b> : inverse la porteuse.

Une fois le canal créé, vous appelez cette fonction pour "armer" la modulation.

- Si **config** est un pointeur valide : La porteuse est **activée**.
- Si **config** est **NULL** (ou 0 en Forth) : La porteuse est **désactivée**.

Cas d'usage typique, si vous voulez piloter une télévision :

1. Vous créez un canal TX classique.
2. Vous appliquez une porteuse de 38 kHz avec **rmt\_apply\_carrier**.
3. Vous envoyez vos données avec **rmt\_transmit**. Le matériel s'occupe tout seul de "hacher" le signal à la bonne fréquence sans que votre CPU n'ait à lever le petit doigt.

## rmt\_del\_channel ( handle -- err )

Le mot **rmt\_del\_channel** est la dernière étape du cycle de vie d'un canal RMT. C'est elle qui assure le "nettoyage" et la libération des ressources matérielles et logicielles.

Lorsque vous appelez cette fonction, le driver effectue les actions suivantes :

- **Libère la mémoire vive (RAM)** allouée pour les structures de contrôle du canal.
- **Désalloue le bloc de mémoire interne** de l'ESP32 (SRAM RMT) qui était réservé pour vos symboles.
- **Détache le GPIO** : La broche physique n'est plus contrôlée par le périphérique RMT.
- **Désinstalle les interruptions** : Le CPU ne surveillera plus les événements de ce canal.

Cause fréquente d'erreurs : il est **interdit** de supprimer un canal qui est encore actif. Pour que `rmt_del_channel` réussisse, le canal doit obligatoirement être dans l'état "**Disabled**".

**L'ordre correct est toujours :**

1. `rmt_disable`
2. `rmt_del_channel`

En Forth, on a tendance à tester des petits bouts de code, à redéfinir des variables et à relancer des scripts.

- Si vous créez un canal (`rmt_new_tx_channel`) à chaque fois que vous lancez votre code sans supprimer l'ancien, vous allez **épuiser les ressources** de l'ESP32.
- L'ESP32 possède un nombre limité de canaux RMT (généralement 4 ou 8 selon le modèle). Une fois qu'ils sont tous alloués, `rmt_new_tx_channel` renverra systématiquement 0 (erreur) tant que vous n'aurez pas fait de `delete`.

## **`rmt_del_sync_manager ( sync_handle -- err )`**

Le mot **`rmt_del_sync_manager`** est une fonction de nettoyage spécifique au **Sync Manager** (Gestionnaire de Synchronisation) du driver RMT 2.0.

Elle intervient uniquement si vous avez utilisé la fonctionnalité de synchronisation de plusieurs canaux RMT.

Dans certains projets complexes, vous avez besoin que plusieurs canaux RMT (par exemple pour piloter 4 rubans de LEDs différents) qui commencent à émettre **exactement en même temps**, au cycle d'horloge près. Pour cela, on crée un "groupe de synchronisation" avec `rmt_new_sync_manager`.

Tout comme `rmt_del_channel` libère un canal, `rmt_del_sync_manager` :

- **Libère les ressources logicielles** liées au groupe de synchronisation.
- **Dissocie les canaux** qui étaient liés à ce manager.
- **Rend la mémoire** consommée par cet objet au système.

Si vous avez fini d'utiliser un groupe de canaux synchronisés et que vous voulez libérer de la mémoire ou réorganiser vos canaux de manière indépendante, vous devez supprimer le manager.

**Note** : La suppression du Sync Manager ne supprime pas les canaux eux-mêmes. Vous devrez toujours appeler `rmt_del_channel` pour chaque canal individuellement.

Si vous pilotez un seul périphérique (un ruban LED, un capteur, ou un émetteur IR), vous n'utiliserez probablement jamais le Sync Manager. C'est une fonction avancée pour la synchronisation multi-canaux.

## rmt\_disable ( handle -- err )

La fonction **rmt\_disable()** est le commutateur qui permet de mettre un canal de transmission (ou de réception) en mode "veille" ou "repos" après son utilisation.

Voici ses rôles et son fonctionnement dans le driver RMT 2.0 :

Lorsque vous appelez **rmt\_disable(handle)**, le driver effectue les opérations suivantes en coulisses :

- **Coupure de l'horloge (Clock Gating)** : Il arrête la source d'horloge (APB, XTAL ou RC) qui alimente le canal. Cela stoppe immédiatement toute consommation d'énergie inutile.
- **Libération des Interruptions** : Il désactive les vecteurs d'interruption associés au canal pour que le CPU ne soit plus sollicité par des événements de fin de transmission.
- **Protection de l'état** : Le canal n'est plus "prêt" à envoyer. Si vousappelez **rmt\_transmit** sur un canal désactivé, vous recevez une erreur **ESP\_ERR\_INVALID\_STATE**.

Dans un programme bien structuré, **rmt\_disable** se situe entre l'utilisation active et la destruction :

1. **Enable** : Le canal "se réveille" et consomme du courant.
2. **Transmit** : Le canal travaille.
3. **Wait Done** : On attend la fin du travail.
4. **Disable** : Le canal "s'endort" (Économie d'énergie).

Contrairement aux anciens microcontrôleurs simples où un périphérique est toujours "allumé", l'ESP32 gère très finement son énergie.

- **Sur batterie** : Laisser un canal RMT activé inutilement empêche certains modes de veille légère (Light Sleep) et consomme quelques milliampères en continu.
- **Sécurité du signal** : Désactiver le canal garantit qu'aucune impulsion parasite ne sera envoyée sur le GPIO si la mémoire est manipulée par erreur.

Il ne faut pas confondre **Disable** et **Delete** :

- **rmt\_disable** : Éteint le moteur, mais la voiture reste garée (la configuration reste en RAM).
- **rmt\_del\_channel** : Envoie la voiture à la casse (libère totalement la mémoire et le handle).

**Note** : Vous devez obligatoirement appeler **rmt\_disable** avant de pouvoir faire un **rmt\_del\_channel**.

## rmt\_enable ( handle -- err )

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\_rx\_channel ( config -- err )

Cette fonction est le pendant de **rmt\_new\_tx\_channel**, mais pour la **réception**. Elle demande au système d'allouer les ressources matérielles nécessaires pour écouter un signal entrant sur un GPIO et le transformer en symboles RMT.

Dans le Core 3.x (RMT 2.0), la réception est devenue beaucoup plus puissante, notamment grâce à une meilleure gestion du DMA et des filtres de bruit.

Les paramètres :

- **config** : Un pointeur vers la structure de configuration (GPIO, horloge, filtres). C'est ici que vous définissez "comment" vous voulez recevoir.

- **ret\_chan** : Un pointeur vers une variable de type `rmt_channel_handle_t`. Si l'allocation réussit, le système y écrit l'identifiant unique du canal de réception.

Pour que cela fonctionne dans **ESP32forth**, vous devez définir cette structure avec précision. Elle diffère légèrement de la version TX :

Champ Forth	Type C	Taille	Description
<code>-&gt;gpio_num</code>	<code>int</code>	4 octets	Le GPIO d'entrée.
<code>-&gt;clk_src</code>	<code>int</code>	4 octets	Source d'horloge (0 par défaut).
<code>-&gt;resolution_hz</code>	<code>uint32_t</code>	4 octets	Précision de la mesure du temps (ex: 1MHz = 1µs).
<code>-&gt;mem_block_symbols</code>	<code>size_t</code>	4 octets	Taille du buffer de réception.
<code>-&gt;trans_queue_depth</code>	<code>size_t</code>	4 octets	Nombre de réceptions pouvant être en attente.
<code>-&gt;intr_priority</code>	<code>int</code>	4 octets	Priorité d'interruption.
<code>-&gt;flags_packed</code>	<code>u32</code>	4 octets	<b>Bit 0:</b> Inversion, <b>Bit 1:</b> DMA, <b>Bit 2:</b> Power Down.

Les nouveautés du RMT 2.0 pour la réception, contrairement à l'ancienne API, la réception 2.0 est conçue pour être "confortable" :

1. **Filtre anti-rebond (Glitch Filter)** : Très important pour ignorer les parasites. (Se configure via une autre fonction : `rmt_apply_rx_glitch_filter`).
2. **Seuil d'inactivité (Idle Threshold)** : Vous définissez après combien de temps de silence (sans changement d'état) on considère que la trame est terminée.
3. **Gestion par événements** : Vous pouvez attacher des "callbacks" qui s'exécutent dès qu'une trame est reçue, sans que le Forth n'ait besoin de scruter (polling) le périphérique en permanence/

## **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\_receive ( rx\_channel buffer bufferSize config -- err )

Le mot **rmt\_receive** est le moteur de la capture de données dans le driver RMT 2.0. Elle permet de lancer une opération d'écoute sur un canal de réception préalablement configuré.

Les Paramètres :

- **rx\_channel** : Le handle du canal RX (créé avec `rmt_new_rx_channel`).
- **buffer** : L'adresse mémoire (en Forth, l'adresse de votre tableau) où les données reçues seront stockées.
  - *Attention* : Contrairement au TX, le RMT écrit ici des **rmt\_symbol\_word\_t**. Chaque symbole occupe 4 octets (2 octets pour la durée, 1 bit pour le niveau).
- **buffer\_size** : La taille maximale du buffer en octets.
- **config** : Un pointeur vers une structure `rmt_receive_config_t`. Vous pouvez passer `NULL` (ou `0` en Forth) pour les réglages par défaut.

Contrairement au TX qui utilise un "Encoder", le RX est plus direct : il transforme les changements d'état sur la broche GPIO en une suite de **durées**.

1. **Lancement** : Vous appelez `rmt_receive`. Le canal se met aux aguets.
2. **Capture** : Dès qu'un front (montant ou descendant) est détecté, le timer interne mesure le temps jusqu'au prochain front.
3. **Stockage** : Chaque segment (High ou Low) est converti en un symbole et écrit dans votre buffer.

4. **Fin** : La réception s'arrête soit quand le buffer est plein, soit quand le signal reste stable plus longtemps que le "seuil d'inactivité" `idle_threshold_ticks`.

Caractéristiques importantes :

- **Non-bloquante** : Comme pour la transmission, cette fonction lance la réception et rend la main immédiatement au Forth. Elle ne "bloque" pas l'interpréteur en attendant le signal.
- **Synchronisation** : Pour savoir quand la réception est terminée, vous devez utiliser une fonction d'attente `rmt_wait_rx_done`, sinon vous lirez un buffer vide ou incomplet.

Il est crucial de comprendre ce que vous allez lire dans votre buffer Forth après un `rmt_receive`.

Un symbole RMT est codé sur 32 bits :

- **15 bits** : Durée de la première partie du symbole.
- **1 bit** : Niveau logique de la première partie (0 ou 1).
- **15 bits** : Durée de la seconde partie.
- **1 bit** : Niveau logique de la seconde partie.

## **rmt\_rx\_register\_event\_callbacks**

C'est la fonction qui permet de rendre votre réception "intelligente". Comme nous l'avons vu, `rmt_receive` est asynchrone : il lance la capture et s'en va.

`rmt_rx_register_event_callbacks` permet de dire à l'ESP32 : "*Dès que tu as fini de recevoir une trame, exécute cette fonction spécifique*".

C'est le pont entre le matériel (le signal qui arrive) et votre logique logicielle.

Sans callbacks, vous seriez obligé de faire du "polling" (boucler sans fin pour vérifier si c'est fini), ce qui gaspille du CPU et manque de précision. Avec un callback :

1. Le matériel reçoit le signal.
2. Une interruption est déclenchée.
3. Le driver appelle votre fonction (le callback).
4. Votre fonction peut lever un drapeau (flag) ou libérer un sémaphore que votre code Forth surveille.

Vous ne passez pas une fonction seule, mais une structure `rmt_rx_event_callbacks_t` qui peut contenir plusieurs événements :

- **on\_recv\_done** : Appelé quand une transaction de réception est terminée avec succès.
- **on\_recv\_error** : Appelé si une erreur survient (ex: buffer trop petit).

Les paramètres :

- **rx\_channel** : Votre handle de réception.
- **cbs** : Pointeur vers la structure contenant les adresses de vos fonctions

- **user\_data** : Un pointeur optionnel vers vos propres données (très utile en Forth pour passer l'adresse d'une variable **variable** ou d'un **value**).

Une fois enregistré, vous n'avez plus besoin de faire une boucle **while** aveugle. Vous faites :

1. **rmt\_receive**
2. Attendre que **rx\_done\_flag** passe à **true** (en Forth : `BEGIN rx-flag @ UNTIL`).
3. Traiter les données et remettre **rx-flag** à **false**.

## **rmt\_wait\_rx\_done ( handle timeout\_ms -- err )**

Contrairement à la transmission (TX), où il existe une fonction dédiée pour attendre, la **réception (RX)** dans le driver moderne d'Espressif (v5.x) repose sur un système de **Notification de Tâche** ou de **File d'attente (Queue)**. Il n'y a effectivement pas de fonction native **rmt\_rx\_wait\_done** directement dans le header.

Ici le mot **rmt\_wait\_rx\_done** est une solution simple pour FORTH.

Quand vous créez un canal RX, vous pouvez lui demander de vous envoyer un signal (via une queue FreeRTOS) dès qu'une trame est prête.

**Note** : ce mot est à utiliser avec précaution.

## **rmt\_sync\_reset ( sync\_handle -- err )**

Le mot **rmt\_sync\_reset** est une commande de "remise à zéro synchronisée". Elle est indissociable du **Sync Manager** dont nous avons parlé précédemment.

Son rôle est de s'assurer que les compteurs de temps (timers) de plusieurs canaux RMT sont parfaitement alignés sur la même valeur de départ avant de lancer une émission.

Normalement, chaque canal RMT possède son propre compteur interne. Si vous lancez quatre canaux les uns après les autres en Forth, il y aura toujours quelques microsecondes de décalage entre le premier et le dernier à cause du temps d'exécution du CPU.

Pour des applications de précision (comme piloter des matrices de LEDs géantes ou des signaux de commande moteur synchronisés), ce décalage est inacceptable.

Lorsque vous appelez **rmt\_sync\_reset**, le périphérique RMT :

1. **Bloque** les horloges des canaux appartenant au groupe.
2. **Réinitialise** tous leurs compteurs à zéro simultanément.
3. **Libère** les horloges pour qu'elles repartent exactement au même "tick".

Le flux de travail avec ce mot ressemble à ceci :

1. Créer les canaux et les ajouter au Sync Manager.
2. Appeler **rmt\_sync\_reset**.
3. Lancer les transmissions (elles partiront toutes sur le même front d'horloge).

Si vous utilisez la synchronisation, vous aurez besoin de ce trio :

- `rmt_new_sync_manager` : Créer le groupe.
- `rmt_sync_reset` : Aligner les horloges (votre question ici).
- `rmt_del_sync_manager` : Détruire le groupe.

**Petite précision importante :** Si vous ne gérez qu'un seul canal (une seule sortie GPIO), cette fonction est inutile. Elle n'est là que pour coordonner une "armée" de canaux.

## **rmt\_transmit ( handle encoder payload len config -- err )**

C'est la fonction **centrale** du driver RMT 2.0. Si `rmt_new_tx_channel` prépare le moteur, `rmt_transmit` est la pédale d'accélérateur : c'est elle qui envoie physiquement les données vers la sortie GPIO.

Voici le détail de ses 5 paramètres, essentiels pour ton implémentation Forth :

### 1. **tx\_channel (Le canal)**

C'est le **handle** (l'identifiant) que tu as récupéré lors de la création du canal. Il indique au driver quel périphérique RMT utiliser (et donc sur quelle broche GPIO envoyer le signal).

### 2. **encoder (Le traducteur)**

C'est la grande nouveauté du RMT 2.0. L'encodeur est un objet qui sait transformer tes données brutes (ton **payload**) en impulsions électriques (temps haut/temps bas).

Exemple : Si tu envoies des octets pour des LEDs WS2812, l'encodeur transforme le bit "1" en 0.8µs High / 0.4µs Low.

### 3. **payload (Les données)**

C'est un pointeur vers tes données en mémoire Forth. Cela peut être n'importe quoi : un tableau d'octets, une chaîne de caractères ou une structure complexe. La fonction ne cherche pas à comprendre ce qu'il y a dedans, elle passe simplement ce pointeur à l'encodeur.

### 4. **payload\_bytes (La taille)**

Indiquer ici la taille totale (en octets) des données pointées par le **payload**. Cela permet au driver de savoir quand s'arrêter.

### 5. **config**

Dans les options de transmission ; on passe un pointeur vers une structure `rmt_transmit_config_t`.

- Elle permet par exemple de définir si l'envoi doit se faire en boucle (`loop_count`).
- **Astuce :** on peut passer NULL (ou 0 en Forth) pour utiliser les réglages par défaut (envoi simple, une seule fois).

Contrairement à l'ancienne version, `rmt_transmit` **n'attend pas** que le signal soit fini pour rendre la main au code Forth.

1. Elle place la demande dans une file d'attente (queue).
2. Elle lance le transfert (souvent via DMA).

3. Elle redonne immédiatement le contrôle à l'interprète Forth.

Comme la fonction est non-bloquante, si on modifie le **payload** (le tableau de données) immédiatement après avoir appelé **rmt.send** en Forth, on risque de corrompre le signal en cours d'envoi.

Il faut utiliser **rmt\_tx\_wait\_all\_done** pour être sûr que l'envoi est terminé avant de passer à la suite.

## **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
#ifndef 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 piloter ces LEDs pour s'en servir comme horloge analogique !

Pour piloter 60 LEDs WS2812B avec le driver RMT 2.0, le "secret" réside dans la précision de l'horloge. Le protocole WS2812B code les bits par des durées très précises (un cycle total d'environ 1.25µs).

Voici les paramètres optimaux pour les structures :



## Structure rmt\_tx\_channel\_config\_t

C'est la fondation. On va régler l'horloge à **10 MHz** (10000000 Hz) pour avoir une résolution de **100 ns** par tick, ce qui rend les calculs très simples.

Champ	Valeur conseillée	Explication
<code>gpio_num</code>	(Ton GPIO)	La pin reliée au fil "Data" de la couronne.
<code>clk_src</code>	<code>RMT_CLK_SRC_DEFAULT</code>	Utilise l'horloge système (80 MHz).
<code>resolution_hz</code>	<b>10 000 000</b>	<b>1 tick = 100 ns.</b> Crucial pour le WS2812B.
<code>mem_block_symbols</code>	<b>64</b>	Standard. Pour 60 LEDs, le DMA fera le reste.
<code>trans_queue_depth</code>	<b>4</b>	Permet d'envoyer la trame suivante pendant que la première finit.
<code>flags.with_dma</code>	<b>1 (True)</b>	<b>Obligatoire</b> pour 60 LEDs (60 x 24 bits = 1440 symboles).

Dans le RMT 2.0, on n'écrit plus les temps à la main. On utilise un **Led Strip Encoder** qui transforme les octets RGB en symboles RMT automatiquement.

Champ      Valeur      Pourquoi ?  
**resolution** **10 000 000** Doit être **identique** à celle du canal TX.

## Les Timings WS2812B (Internes à l'encodeur)

Si on devait les configurer manuellement (ou pour comprendre ce que fait l'encodeur), voici les valeurs standards converties en **ticks de 100 ns** :

- **Bit "0"** : 400ns High / 850ns Low → **4 ticks High / 9 ticks Low**.
- **Bit "1"** : 800ns High / 450ns Low → **8 ticks High / 5 ticks Low**.
- **Reset** : >50µs Low → **500 ticks Low**.

## Configuration de la transmission (rmt\_transmit\_config\_t)

Pour une couronne de LEDs, on reste simple :

- **loop\_count** : 0 (On n'envoie la couleur qu'une fois).
- **eot\_zealous** : 1 (Pour s'assurer que le signal revient bien à zéro immédiatement après la dernière LED, sinon la première LED pourrait scintiller).

```
\ Créer un canal TX à 10MHz
1000000 rmt_new_ws2812_encoder  \ Résolution 1MHz sur la pile
                                \ Retourne le handle de l'encodeur ou 0

\ Pour utiliser avec rmt_transmit :
\ channel encoder data_ptr byte_count config_ptr rmt_transmit
```