ESP32forth
et la librairie RMT

Marc PETREMANN

Version 1.0 - 16/02/26

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.

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
`loop_count`	`int`	Nombre de répétitions du message. • `0` : Envoi simple (défaut). • `-1` : Boucle infinie. • `>0` : Nombre spécifique de répétitions.
`flags`	`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

Infos : https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/peripherals/rmt.html

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=100ns.

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

Champ Forth	Type C	Taille
`->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
`frequency_hz`	`uint32_t`	La fréquence de la porteuse (ex: `38000` pour l'IR).
`duty_cycle`	`float`	Le rapport cyclique (souvent `0.33` ou `0.5`).
`flags`	`struct`	`polarity_active_low` : 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 vous appelez `rmt_transmit` sur un canal désactivé, vous recevrez 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
`->gpio_num`	`int`	4 octets	Le GPIO d'entrée.
`->clk_src`	`int`	4 octets	Source d'horloge (0 par défaut).
`->resolution_hz`	`uint32_t`	4 octets	Précision de la mesure du temps (ex: 1MHz = 1µs).
`->mem_block_symbols`	`size_t`	4 octets	Taille du buffer de réception.
`->trans_queue_depth`	`size_t`	4 octets	Nombre de réceptions pouvant être en attente.

Champ Forth	Type C	Taille	Description
`->intr_priority`	`int`	4 octets	Priorité d'interruption.
`->flags_packed`	`u32`	4 octets	**Bit 0**: Inversion, **Bit 1**: DMA, **Bit 2**: 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 buffSize 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)
   Iindiquer 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 modifiee 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

Les mots pour WS2812

Ce sont des mots définis pour une application précise. Ils sont définis dans le vocabulaire `rmt` mais sont des extensions de l'API RMT20.

rmt_new_ws2812_encoder (resolution_hz -- encoder|0)

Ce code est crucial car il transforme le périphérique RMT (qui est un simple générateur d'impulsions) en un contrôleur intelligent capable de parler le "langage" des LEDs WS2812.

Le premier paramètre récupéré depuis la pile Forth est la résolution en Hz (par exemple **1 000 000** pour 1 MHz).

- Le code calcule les durées des impulsions en se basant sur cette résolution.
- **Attention mathématique** : Dans un code, `(res / 1000000)` vaut **1** si la résolution est de 1 MHz. Les multiplications par `0.4` ou `0.85` risquent d'être tronquées si le compilateur traite cela comme des entiers. Pour plus de précision, il est souvent préférable de multiplier avant de diviser.

Le protocole WS2812 utilise la durée d'une impulsion **HAUTE** suivie d'une impulsion **BASSE** pour différencier un `0` d'un `1`.

Symbole	Durée Totale	État Haut (Level 1)	État Bas (Level 0)
Bit 0	~1.25 µs	`t0h` (0.4 µs)	`t0l` (0.85 µs)
Bit 1	~1.25 µs	`t1h` (0.8 µs)	`t1l` (0.45 µs)

L'encodeur "Bytes" est un traducteur : il prend un octet (8 bits) et, pour chaque bit, il demande au RMT de générer les deux durées définies :

- `level0 = 1` / `level1 = 0` : Cela signifie que la première partie de l'impulsion est à 3.3V (High) et la seconde à 0V (Low).
- `msb_first = 1` : Les LEDs WS2812 attendent les données dans l'ordre du bit le plus significatif au moins significatif (souvent Vert, puis Rouge, puis Bleu).

La fonction `rmt_new_bytes_encoder` crée l'objet en mémoire.

- Si pas d'erreur, empile l'adresse de l'encodeur (le **Handle**).
- Sinon, empile `0`.

prepare_ws2812_config (gpio -- handle|0)

@TODO : à rédiger

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 !

Les LEDs adressables

Les LEDs adressables sont des diodes électroluminescentes dont chaque unité peut être contrôlée indépendamment, ce qui permet d'afficher des couleurs et des effets lumineux variés sur une seule bande ou un seul panneau. Voici une explication simple de leur fonctionnement :

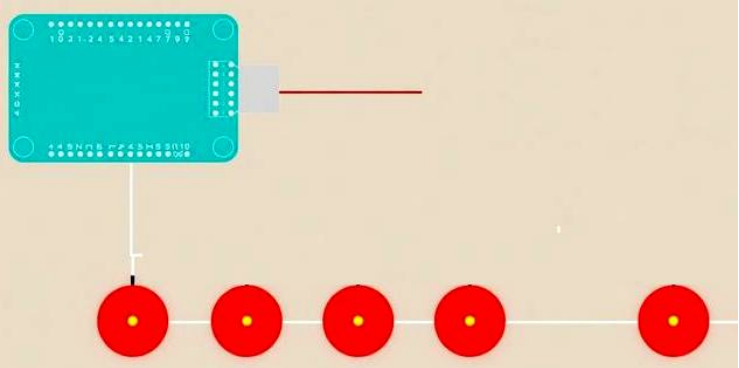
1. **Structure interne** : chaque LED adressable comprend généralement une petite puce de contrôle intégrée (comme WS2812, SK6812, WS2801, etc.) à l'intérieur, en plus de la diode lumineuse. Cette puce permet de recevoir et de traiter des données pour contrôler la couleur et la luminosité.
2. **Contrôle par un signal numérique** : les LED sont connectées en série. Le signal de contrôle (données) est envoyé par une seule ligne depuis un microcontrôleur (Arduino, Raspberry Pi, etc.). Ce signal contient des instructions pour chaque LED, une après l'autre.
3. **Transmission en cascade** : le microcontrôleur envoie une chaîne de bits représentant la couleur (rouge, vert, bleu) pour la première LED. La première LED lit ses bits, enregistre ses paramètres, puis transmet le reste du signal à la LED suivante, qui fait de même. Ainsi, chaque LED reçoit son propre ensemble de données et affiche la couleur spécifiée.
4. **Mise à jour et effets lumineux** : en modifiant le signal envoyé, on peut changer les couleurs de chaque LED individuellement, créer des animations, des dégradés, des effets de scintillement, etc.
5. **Alimentation** : les LEDs nécessitent une alimentation électrique adaptée, souvent 5V, et il faut faire attention à la gestion du courant lorsque beaucoup de LEDs sont allumées en même temps.

En résumé, grâce à leur puce de contrôle intégrée et à une communication en série, les LEDs adressables permettent un contrôle précis et flexible de chaque pixel lumineux dans une bande ou un panneau.

En pratique , voici la description du processus :

- Une source de contrôle (microcontrôleur) envoie un signal de données.

- La première LED reçoit le signal, lit ses données, affiche la couleur et transmet le reste du signal à la LED suivante.
- Chaque LED possède une puce de contrôle interne.
- La chaîne continue jusqu'à la dernière LED.



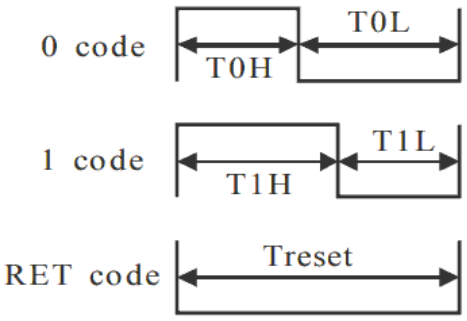
Apport de la librairie RMT dans le contrôle des LEDs adressables

Les LEDs adressables (comme WS2812) nécessitent des signaux avec des timings très précis pour distinguer les bits "0" et "1". La librairie RMT peut générer ces signaux avec une précision microsecondes, évitant ainsi les erreurs dues à la gestion logicielle classique.

Avec les mots RMT, le microcontrôleur peut déléguer la génération du signal à un matériel dédié, libérant le processeur pour d'autres tâches. Cela permet d'envoyer des données à une grande quantité de LEDs sans surcharge CPU.

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).

Pour avoir cette précision, le vocabulaire `rmt` intègre le mot `rmt_new_ws2812_encoder`. Ce mot encode en dur tous les paramètres spécifiques au pilotage de LEDs adressables avec leur composant WS2812.

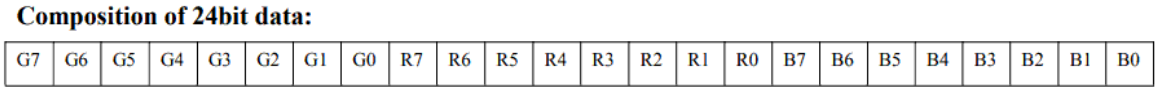


Les valeurs T0H, T0L, T1H et T1L sont prédéfinies dans le mot `rmt_new_ws2812_encoder`.

Le seul paramètre nécessaire pour `rmt_new_ws2812_encoder` est la valeur en Hz de la résolution nécessaire pour la génération du signal. En retour, on récupère un handle à conserver pour utilisation ultérieure.

Encodage pour une LED adressable

Pour contrôler une LED, il faut paramétrer les données à transmettre dans un espace de 24 bits, réparti comme suit :



Détail :

- bits G7..G0 qui déterminent la couleur verte (G = Green)
- bits R7..R0 qui déterminent la couleur rouge (R = Red)
- bits B7..B0 qui déterminent la couleur bleue (B = Blue)

L'ordre des couleurs dans une séquence 24 bits transmise au module WS2812 est **GRB**. Ceci est très important, car la majorité des systèmes de coloration font référence aux couleurs RGB (Red Green Blue). On verra plus tard comment gérer cette différence de codage des couleurs.

Pour contrôler n LEDs, on réservera un espace mémoire de n x 3octets. Pour la couronne LEDs, on réservera donc 180 octets :

```
60 constant NB_LEDS         \ defined for ring with 60 LEDs
create LEDS                 \ array for these LEDs
    NB_LEDS RGB_struct * allot
```

On utilise la structure `RGB_struct` comme multiplicateur pour arriver à réserver nos 180 octets. Définition de cette structure :

```
\ define RGB structure
struct RGB_struct
    i8 field ->g
    i8 field ->r
    i8 field ->b
```

Notez que nos accesseurs sont définis dans l'ordre des couleurs **GRB**. C'est le seul endroit, dans tout notre code source, où on définit les couleurs dans cet ordre. Pour tous les autres mots FORTH, on utilisera l'ordre des couleurs **RGB**.

Définition d'une couleur de LED

Pour récupérer les valeurs RGB d'une couleur définie en mémoire, on définit `RGB@` :

```
\ get values from RGB address
```

```
: RGB@ { addr -- r g b }
    addr ->r c@
    addr ->g c@
    addr ->b c@
  ;
```

Oui, nos accesseurs de structure sont bien dans l'ordre **RGB** ! C'est là que réside toute la force des structures. Seul l'ordre de définition des accesseurs est important. Ensuite, on peut modifier l'ordre comme ci-dessus. Si je précise une adresse, l'exécution de `RGB@` va chercher les données stockées dans l'ordre **GRB** et les empiler dans l'ordre **RGB**.

L'intensité des LEDs est géré par ces définitions ;

```
\ determine LED intensity in range 0..100
100 value INTENSITY

: setIntensity ( n -- )
    100 min 0 max
    to INTENSITY
  ;

\ modify LED intensity
: adjustIntensity ( color -- color' )
    INTENSITY 100 */
  ;
```

J'ai fait le choix d'étaler la luminosité d'une couleur dans un intervalle 0..100. Cette valeur d'intensité lumineuse est stockée dans la valeur `INTENSITY`.

Le mot `setIntensity` va affecter une autre valeur à `INTENSITY`. Exemple, si on exécute 20 setIntensity, toutes les valeurs **RGB** seront ajustées à 20 % de leur valeur brute. C'est ce qui est effectué dans cette définition :

```
\ store r g b in RGB address
: RGB! { r g b addr -- }
    r adjustIntensity addr ->r c!
    g adjustIntensity addr ->g c!
    b adjustIntensity addr ->b c!
  ;
```

Cette définition ne servira que pour stocker une couleur RGB et son intensité dans la zone des données réservées à la transmission vers le module WS2812.

Avant d'en arriver là, voici comment définir quelques couleurs de base :

```
\ Define named color
: RGBcolor: ( comp: r g b -- <name> | exec: -- r g b )
    create
        >r >r c,
        r> c,
        r> c,
```

```
    does>
        RGB@
  ;

  0   0   0 RGBcolor: RGB_Black
255   0   0 RGBcolor: RGB_Red
  0 255   0 RGBcolor: RGB_Green
  0   0 255 RGBcolor: RGB_Blue
255 255   0 RGBcolor: RGB_Yellow
  0 255 255 RGBcolor: RGB_Cyan
255   0 255 RGBcolor: RGB_Magenta
255 255 255 RGBcolor: RGB_White
255 165   0 RGBcolor: RGB_Orange
128   0 128 RGBcolor: RGB_Purple
255 192 203 RGBcolor: RGB_Pink
```

Ici, le mot `RGB_Red` est une sorte de constante de couleur RGB de base. Pour stocker cette couleur avec une intensité à 50 %;

```
50 setIntensity
RGB_Red LEDS RGB!
```

Les valeurs **GRB** stockées dans l'espace mémoire réservée de LEDS sont automatiquement ajustées en fonction de l'intensité demandée.

Gestion des LEDS de la couronne LEDs

On revient sur la définition de l'espace mémoire LEDS destiné à recevoir les couleurs à transmettre. Pour 60 LEDs, on réserve 60x3 octets :

```
create LEDS                 \ array for these LEDs
    NB_LEDS RGB_struct * allot
```

A partir de là, il est aisé de créer le mot qui va stocker les trois couleurs d'une LED à la position n :

```
: nLED!  ( r g b position -- )
    NB_LEDS 1- min            \ sécurité anti-débordement
    RGB_struct * LEDS +     \ calcul adresse réelle
    RGB!
  ;
```

Dans cette définition, la séquence `NB_LEDS 1- min` est une sécurité évitant le débordement de la position pour la cantonner dans l'intervalle 00..59 (donc 60 positions au total).

Enfin, on définit le mot qui nettoie notre espace mémoire réservé aux 60 LEDs :

```
\ set all LEDs to zero
: resetLEDs ( -- )
    LEDS NB_LEDS RGB_struct * 0 fill
```

```
  ;
```

Transmission des données LEDs

La toute première opération consiste à définir le pin qui transmet les données en mode sortie :

```
: initGPIO ( -- )
    RMT_GPIO OUTPUT pinMode
  ;
```

Ensuite, on initialise le canal de transmission RMT :

```
also rmt

\ *** Initialisation RMT ***

0 value channelHandle
0 value encoderHandle

: initWS2812 ( -- )
    initGPIO    \ defined in fastleds.fs
    RMT_GPIO RMT_RES prepare_ws2812_config
    ?dup if
        to channelHandle
    else
        ." erreur initialisation canal RMT" cr
    then
    \ Créer un encodeur à 10 Mhz
    RMT_RES rmt_new_ws2812_encoder  \ Résolution 1MHz sur la pile
                                    \ Retourne le handle de l'encodeur ou 0
    ?dup if
        to encoderHandle
    else
        ." erreur initialisation encodeur RMT" cr
    then
  ;
```

Le mot **`initWS2812`** définit le canal dont le handle est conservé dans **`channelHandle`**, puis définit l'encodeur dont le handle est stocké dans **`encoderHandle`**.

Pour des raisons purement techniques, on définit aussi cette structure mais qui restera vide.

```
\ Define a empty structure for rmt_transmit
create TRANSMIT_CONFIG
    rmt_transmit_config_t allot
    TRANSMIT_CONFIG rmt_transmit_config_t 0 fill

only FORTH
```

Et enfin on définit le le mot qui transmet les valeurs de nos 60 LEDs :

```
also rmt

\ transmit LEDs stored in LEDS array
: transmitLEDS ( -- )
    channelHandle 0= if
        initWS2812
        channelHandle rmt_disable drop    \ On s'assure que le canal est
propre
        channelHandle rmt_enable drop     \ Activation ---
    then
    channelHandle encoderHandle
    LEDS NB_LEDS RGB_struct * TRANSMIT_CONFIG rmt_transmit
    ?dup if
        ." Erreur transmission LEDs" cr
    then
  ;

only FORTH
```

Le cœur de cette définition c'est le mot **`rmt_transmit`**. Les explications de ce mot se trouvent plus haut dans ce document. Cette définition fait beaucoup appel à des valeurs et constantes. Cette méthode de programmation permet de réutiliser cette même définition quasiment sans modification pour n'importe quel type de LEDs adressables.

Définition de l'horloge à LEDs adressables

On commence par définir une nouvelle structure destinée à décrire la couleur **ET** l'intensité d'une LED affectée à une utilisation précise dans l'ensemble des 60 LEDs de notre couronne de LEDs

```
structures also

\ define CLOCK structure for seconds, hours....
struct CLOCK_struct
    RGB_STRUCT field ->rgb
            i8 field ->intensity

only FORTH
```

Cette nouvelle structure reprend cette utilisée pour la définition des couleurs de base, mais en rajoutant l'accesseur **`->intensity`**. Le mot suivant va récupérer les données d'une couleur de l'horloge :

```
\ get values from CLOCK address
: CLOCK@ { addr -- r g b intensity }
    addr ->r c@
    addr ->g c@
    addr ->b c@
```

```
    addr ->intensity c@
  ;
```

Voici comment on définit chaque point de notre horloge :

```
\ Define CLOCK color
: CLOCKcolor: ( comp: r g b intensity -- <name> | exec: -- r g b intensity )
    create
        >r >r >r c,
        r> c,
        r> c,
        r> c,
    does>
        CLOCK@
  ;
```

Le mot **`CLOCKcolor:`** traite quatre paramètes et est suivi du mot à définir. Les quatres paramètres correspondent à la couleur de base du point de l'horloge et son intensité :

```
RGB_Blue 1 CLOCKcolor: CLOCK_minutes
RGB_Blue 8 CLOCKcolor: CLOCK_hours
```

Voyons maintenant comment utiliser ces points pour mettre en évidence les minutes et les heures fixes sur le fond de l'horloge à LEDs adressables :

```
\ couronne en bleu très faible pour marquer les minutes
: setInitMinutes ( -- )
    CLOCK_minutes setIntensity drop drop drop
    60 0 do
       CLOCK_minutes drop i nLED!
    loop
  ;

\ couronne en bleu très faible pour marquer les minutes
: setInitHours ( -- )
    CLOCK_hours setIntensity drop drop drop
    60 0 do
         CLOCK_hours drop i nLED!
    5 +loop
  ;
```

Le mot **`setInitMinutes`** va colorer toutes les LEDs dans la teinte définie par **`CLOCK_minutes`**. De même, le mot **`setInitHours`** va colorer seulement une LED sur cinq, c'est à dire les LEDs marquant les heures, dans la teinte définie par **`CLOCK_hours`**. On exécute **`setInitMinutes`** avant **`setInitHours`**.

Mise en place des aiguilles

Une fois le fond de l'horloge défini, on regarde comment placer les « aiguilles » de notre horloge numérique à affichage analogique.

Ce premier mot **calc-led-hh** va positionner l'aiguille des heures sur une des 60 positions disponibles dans notre horloge ;

```
\ calculate HH position on clock
: calc-led-hh  { hh mm -- iLed }
    hh 12 mod 5 *       \ Calcul de la base (ex: 2h -> 10)
    mm 6 + 12 /         \ Calcul de l'avance des minutes avec arrondi
    + 60 mod            \ Addition et sécurité cycle 60
  ;
```

On définit maintenant la couleur de chaque aiguille :

```
RGB_Cyan   20 CLOCKcolor: CLOCK_SS
RGB_Green  20 CLOCKcolor: CLOCK_MM
RGB_red    20 CLOCKcolor: CLOCK_HH
```

L'aiguille des secondes sera de la couleur cyan, celles des minutes en vert, celle des heures en rouge, toutes les trois avec une intensité lumineuse à 20 %.

Les trois mots suivants positionnent chaque aiguille sur notre horloge :

```
: setSS { sec -- }
    CLOCK_SS setIntensity
    sec nLED!
  ;

: setMM { min -- }
    CLOCK_MM setIntensity
    min nLED!
  ;

: setHH { hour min -- }
    hour min calc-led-hh { position }
    CLOCK_HH setIntensity
    position nLED!
  ;
```

Ces trois actions sont regroupées dans **setHMS** :

```
: setHMS ( -- )
    RTC.get-time
    setSS
    dup setMM
    setHH
  ;
```

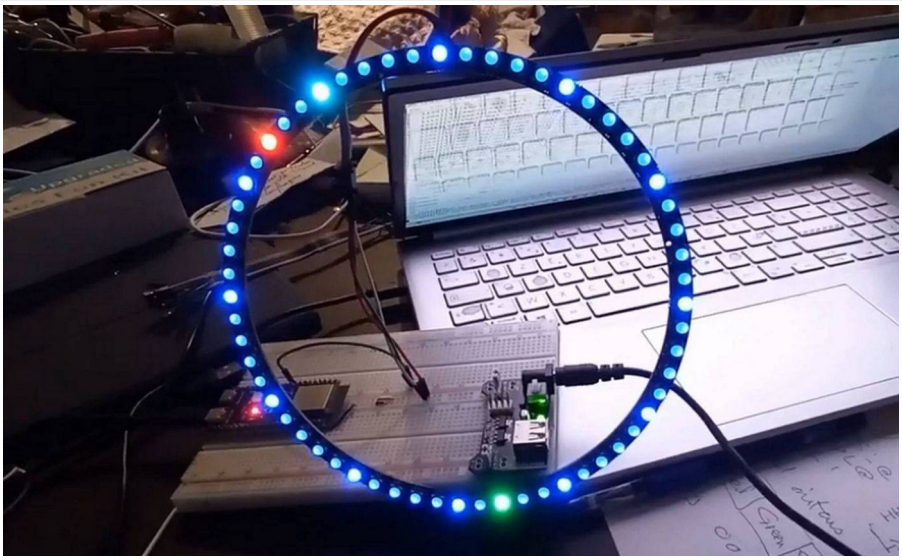
Et on termine par l'affichage de l'horloge. L'affichage est placé dans une boucle **begin..again** :

```
: displayClock ( -- )
    begin
        resetLEDs
```

```
        setInitMinutes
        setInitHours
        setHMS
        transmitLEDS
        1000 ms
    again
  ;
```

En l'état, on peut lancer notre horloge :

```
16 42 00 RTC.set-time
displayClock
```



Mais si vous faites ceci, la carte ESP32 ne sera plus disponible….

Mettons un peu de multi-tâches dans l'horloge

C'est à cette étape que ESP32forth révèle toute sa puissance. FORTH est un langage multi-tâches. Un langage multi-tâches peut exécuter plusieurs programmes simultanément. Voici comment procéder, et c'est vraiment d'une simplicité extraordinaire :

```
15 57 00 RTC.set-time
' displayClock 100 100 task my-clock
my-clock start-task
```

On ne compte pas la première ligne de code, celle-ci initialisant simplement notre horloge.

La seconde ligne prend le code d'exécution de **`displayClock`**, suivi d'un espace dédié aux deux piles de la tâche définie par **`task`**. Le mot task est suivi de la tâche à définir, ici **`my-clock`**.

La dernière ligne lance la tâche **`my-clock`**.

Si tout se passe bien, l'horloge s'affiche comme sur la photo ci-dessus, mais en plus, **vous avez toujours la main** de l'interpréteur FORTH !