ESP32forth
et la librairie RMT

Marc PETREMANN

Version 1.0 - 07/02/26

Table des matières

Les mots du vocabulaire rmt

Liste de tous les mots définis :

```
rmt_set_clk_div rmt_get_clk_div rmt_set_rx_idle_thresh rmt_get_rx_idle_thresh
rmt_set_mem_block_num rmt_get_mem_block_num rmt_set_tx_carrier rmt_set_mem_pd
rmt_get_mem_pd rmt_tx_start rmt_tx_stop rmt_rx_start rmt_rx_stop
rmt_tx_memory_reset rmt_rx_memory_reset rmt_set_memory_owner
rmt_get_memory_owner rmt_set_tx_loop_mode rmt_get_tx_loop_mode rmt_set_rx_filter
rmt_set_source_clk rmt_get_source_clk rmt_set_idle_level rmt_get_idle_level
rmt_get_status rmt_set_rx_intr_en rmt_set_err_intr_en rmt_set_tx_intr_en
rmt_set_tx_thr_intr_en rmt_set_gpio rmt_config rmt_isr_register
rmt_isr_deregister rmt_fill_tx_items rmt_driver_install rmt_driver_uinstall
rmt_get_channel_status rmt_get_counter_clock rmt_write_items rmt_wait_tx_done
rmt_get_ringbuf_handle rmt_translator_init rmt_translator_set_context
rmt_translator_get_context rmt_write_sample
```

rmt_config (&config -- fl)

Le mot **`rmt_config`** est le chef d'orchestre. C'est la fonction qui prend toute la structure de configuration et l'applique d'un coup au matériel pour réveiller le périphérique.

En Forth, c'est le moment où les données en mémoire deviennent une réalité physique sur les broches de la carte ESP32.

Cette fonction ne se contente pas de copier des chiffres. Elle effectue plusieurs opérations critiques :

1. **Activation de l'horloge :** Elle allume le moteur du périphérique RMT (qui est éteint par défaut pour économiser l'énergie).
2. **Routage Matrix GPIO :** Elle connecte le signal interne du RMT à la broche physique (`rmt_gpio_num`) que tu as choisie.
3. **Configuration du canal :** Elle définit si le canal est en mode **TX** (émission) ou **RX** (réception).
4. **Allocation mémoire :** Elle réserve des blocs de la RAM interne du RMT (le `rmt_mem_block_num`).
5. **Initialisation du Divider :** Elle applique le fameux `rmt_clk_div` pour caler le timing.

Le mot `rmt_config` attend un pointeur vers une structure `rmt_config_t`. Voici à quoi cela ressemble conceptuellement :

Champ	Rôle
rmt_mode	Décide si le canal parle (TX) ou écoute (RX).
rmt_channel	Lequel des 8 canaux (0-7) tu configures.
rmt_gpio_num	La broche physique (ex: GPIO 18).
clk_div	La vitesse des "ticks" (1-255).
mem_block_num	Combien de "blocs" de 64 mots de 32 bits on alloue (souvent 1).

Exemple d'utilisation, pour appeler cette fonction, ondoit d'abord remplir le tampon mémoire (la structure) puis passer son adresse à la fonction :

```
\ 1. Créer un espace mémoire pour la config
create ma-config rmt_config_t allot

\ 2. Remplir les champs
RMT_MODE_TX      ma-config rmt_mode !
0                ma-config rmt_channel !
18               ma-config rmt_gpio_num !
80               ma-config rmt_clk_div c!   \ 1 tick = 1 microseconde
1                ma-config rmt_mem_block_num c!

\ 3. Appeler la fonction C (via l'interface FFI)
ma-config rmt_config ( renvoie un code d'erreur esp_err_t )
```

Les points de vigilance :

- **Les erreurs de retour :** `rmt_config` renvoie toujours un `esp_err_t`. En Forth, il est vital de vérifier si la pile renvoie `0` (ESP_OK). Si elle renvoie autre chose, c'est probablement que le GPIO est invalide ou que le canal est déjà utilisé.
- **Réinitialisation :** Appeler `rmt_config` remet le canal à zéro. Si on avait des données en cours d'envoi, elles seront interrompues.
- **Alignement :** Si le `rmt_clk_div` (1 octet) est mal placé à cause d'un oubli de padding, `rmt_config` lira les mauvaises valeurs et le signal sera illisible.

rmt_driver_install (channel RxBufSize flag -- n0)

Avant même de dire à quel GPIO auquel il faut se connecter, `rmt_driver_install` prépare les ressources logicielles nécessaires au sein d'ESP-IDF pour gérer un canal spécifique.

Contrairement à `rmt_config` qui règle le matériel (le silicium), `rmt_driver_install` s'occupe de la partie logicielle (la mémoire système et les interruptions).

Les trois paramètres clés :

1. **`channel` (0-7)** : Le numéro du canal à activer.
2. **`RxBufSize`** : C'est la taille (en octets) du **Ring Buffer** (tampon circulaire).
    - **En mode TX :** On met généralement **0**, car les données sont envoyées directement depuis ta RAM vers la RAM du RMT.
    - **En mode RX :** C'est indispensable. C'est la "salle d'attente" où le driver stocke les impulsions reçues avant que le code Forth ne vienne les lire.
3. **`flags`** : Définit la priorité de l'interruption (souvent mis à `0` pour laisser l'ESP32 choisir la valeur par défaut, ou `ESP_INTR_FLAG_IRAM` pour plus de performance).

Si on essaie d'utiliser `rmt_write_items` ou `rmt_config` sans avoir fait un `rmt_driver_install` au préalable, le système risque de planter ou de renvoyer une erreur `ESP_ERR_INVALID_STATE`.

C'est cette fonction qui :

- Alloue la mémoire pour gérer les files d'attente (Queues).
- Installe le "Handler" d'interruption qui va surveiller la fin des transmissions.

- Protège le canal pour qu'une autre partie du programme ne tente pas de l'utiliser en même temps (Mutex).

Exemple d'utilisation en Forth, pour configurer le **canal 0** pour de l'émission seule (TX) :

```
\ Paramètres : canal=0, rx_buf=0, flags=0
0 0 0 rmt_driver_install ( -- esp_err_t )

\ Vérification du résultat
dup 0 <> if
    ." Échec de l'installation du driver: " . cr
else
    drop ." Driver RMT installé avec succès !" cr
then
```

L'ordre logique pour ne pas avoir de surprises est le suivant :

1. **`rmt_driver_install`** : On réserve les ressources système.
2. **`rmt_config`** : On lie le canal au GPIO et on règle le `clk_div`.
3. **`rmt_write_items`** : On envoie les données.
4. *(Optionnel)* **`rmt_driver_uninstall`** : Si tu as fini et que tu veux libérer la RAM.

Si on compte utiliser le RMT pour lire des signaux (comme une télécommande infrarouge), la valeur de `rx_buf_size` est critique. Si elle est trop petite, on perdra des morceaux du message si le signal est long. Pour de l'infrarouge, on utilise souvent une valeur autour de 1000 octets.

rmt_driver_uninstall (channel -- fl)

C'est l'étape du "**grand ménage**". Si `rmt_driver_install` est l'ouvrier qui installe son chantier, `rmt_driver_uninstall` est celui qui range ses outils, nettoie la pièce et rend les clés. Son rôle est crucial pour la stabilité de ton système sur le long terme.

Lors de son appel, `rmt_driver_uninstall(channel)`, l'ESP32 effectue les actions suivantes :

- **Libération de la RAM :** Elle supprime le **Ring Buffer** (le tampon circulaire) que tu avais alloué lors de l'installation. Si tu avais réservé 1000 octets pour la réception, ils retournent dans la mémoire libre du système.
- **Désactivation des interruptions :** Elle retire le "handler" d'interruption lié à ce canal. Cela évite que le processeur ne tente de sauter vers un code qui n'existe plus si un signal parasite arrive sur la broche.
- **Libération du GPIO :** La broche (GPIO) qui était verrouillée par le périphérique RMT redevient disponible pour d'autres usages (comme du GPIO standard ou un autre protocole).
- **Extinction partielle :** Si c'était le dernier canal utilisé, le driver peut éteindre l'horloge du périphérique RMT pour économiser de l'énergie.

En Forth, on a souvent tendance à "allumer" les choses et à les laisser tourner. Cependant, `rmt_driver_uninstall` est indispensable si :

1. **On changes de mode :** Par exemple, pour passer d'un canal configuré en RX (Réception) à un canal en TX (Émission) sur la même broche.
2. **Gestion de l'énergie :** quand on a fini d'envoyer les données et qu'on veut que l'ESP32 consomme le moins possible avant de passer en mode sommeil.
3. **Éviter les fuites de mémoire (Memory Leaks) :** Si le programme Forth installe le driver en boucle sans jamais le désinstaller, ça finiras par épuiser la RAM de l'ESP32 (erreur `ESP_ERR_NO_MEM`).

La fonction est très simple car elle ne prend qu'un seul argument : le numéro du canal. Exemple :

```
\ Désinstaller le driver du canal 0
0 rmt_driver_uninstall ( -- esp_err_t )

\ Toujours vérifier si tout s'est bien passé
dup 0 <> if
    ." Erreur lors de la désinstallation : " . cr
else
    drop ." Canal 0 libéré." cr
then
```

N'appellez jamais `rmt_driver_uninstall` pendant qu'une transmission est en cours ! Si on lance un envoi avec `rmt_write_items`, s'assurer d'appeler `rmt_wait_tx_done` avant de désinstaller le driver. Sinon, on risque de provoquer un "crash" car le matériel essaiera d'accéder à une mémoire qui viens de libérer.

rmt_fill_tx_items

rmt_get_channel_status

rmt_get_clk_div (channel divAddr -- fl)

Si `rmt_set_clk_div` est l'action d'écrire une consigne sur le tableau de bord, **`rmt_get_clk_div`** est l'action de regarder le compteur pour vérifier ce qui est réellement en train de se passer.

C'est la fonction de lecture (le "Getter") qui permet d'interroger le matériel (le périphérique RMT) pour savoir quelle division d'horloge est actuellement appliquée à un canal donné.

On pourrait se dire : "Pourquoi demander à la puce ce que je viens de lui dire ?". En programmation système/Forth, c'est utile pour trois raisons :

- **Vérification (Sanity Check) :** S'assurer que la configuration a bien été prise en compte par le matériel.
- **Calculs dynamiques :** Si une autre partie de ton programme a modifié le timing, tu peux lire la valeur actuelle pour recalculer la durée réelle de tes impulsions sans avoir à stocker une variable redondante en RAM.
- **Débogage :** Inspecter l'état du périphérique après un crash ou une mise en veille.

La fonction va lire directement dans les registres de configuration de l'ESP32. Pour le RMT, cela correspond généralement au champ `div_cnt` du registre `RMT_CHnCONF0_REG`.

Paramètres :

- **`channel`** : Le canal (0 à 7).
- **`*div`** : Un pointeur vers un octet où la fonction va écrire la valeur trouvée.

Exemple :

```
variable mon_diviseur
0 mon_diviseur rmt_get_clk_div  \ Lit le diviseur du canal 0
mon_diviseur @  .               \ Affiche la valeur (1-255)
```

N'oubliez pas que `rmt_get_clk_div` renvoie la **valeur brute du diviseur** (le chiffre entre 1 et 255). Pour connaître la fréquence réelle des ticks, il faut quand même connaître la fréquence de la source (généralement 80 MHz).

rmt_get_counter_clock

rmt_get_idle_level

rmt_get_mem_block_num

rmt_get_mem_pd

rmt_get_memory_owner

rmt_get_ringbuf_handle

rmt_get_rx_idle_thresh

rmt_get_source_clk

rmt_get_status

rmt_get_tx_loop_mode

rmt_isr_deregister

rmt_isr_register

rmt_rx_memory_reset

rmt_rx_start

rmt_rx_stop

rmt_set_clk_div (channel div -- fl)

Le mot `rmt_set_clk_div` est essentiellement le **métronome** de votre signal. Pour que l'ESP32 sache combien de temps doit durer un "1" ou un "0" sur le fil, il a besoin d'une unité de temps de base : c'est le **tick**.

Voici comment cela fonctionne techniquement. Le périphérique RMT est cadencé par une horloge source, généralement l'**APB Clock** (qui tourne à 80 MHz sur la plupart des ESP32). 80 MHz, c'est beaucoup trop rapide pour la plupart des protocoles (un tick durerait 12,5 nanosecondes). Le `clk_div` permet de ralentir cette horloge pour obtenir une résolution qui nous arrange.

Le diviseur est un entier codé sur **8 bits** (valeur de 1 à 255).

En utilisant l'horloge APB (80 MHz) :

- Avec un `div` de **80**, le tick dure exactement 1µs.
- Avec un `div` de **1**, le tick dure 12,5ns.
- Avec un `div` de **255** (maximum), le tick dure environ 3,18µs.

Chaque élément de donnée envoyé par le RMT est composé de deux parties : une durée de niveau haut et une durée de niveau bas. Ces durées ne sont pas exprimées en secondes, mais en **nombre de ticks**.

- **Exemple pour des LEDs WS2812 (NeoPixels) :** Elles demandent souvent des impulsions d'environ 0.4µs et 0.8µs. En réglant le `clk_div` à **8** (sur une base 80 MHz), le tick vaut 0.1µs.
    - Pour 0.4µs, ça indique au RMT de compter **4 ticks**.
    - Pour 0.8µs, ça indiquera de compter **8 ticks**.
- **Résolution vs Durée totale :** Plus le diviseur est petit, plus tu es précis (résolution fine), mais plus la durée maximale de ton signal est courte (car le compteur de ticks dans le matériel est lui aussi limité, souvent à 15 bits).
- **Zéro :** Dans la structure de l'ESP32, une valeur de `0` pour le diviseur est généralement traitée comme un `1` (pas de division).

Dans la structure, le champ `rmt_clk_div` attend cette valeur de 1 à 255. Pour piloter un protocole spécifique, c'est la première valeur à calculer pour s'assurer que tes "ticks" tombent juste par rapport aux durées cibles du protocole.

@TODO : exemple à écrire

rmt_set_err_intr_en

rmt_set_gpio

rmt_set_idle_level

rmt_set_mem_block_num

rmt_set_mem_pd

rmt_set_memory_owner

rmt_set_rx_filter

rmt_set_rx_idle_thresh

rmt_set_rx_intr_en

rmt_set_source_clk

rmt_set_tx_carrier

rmt_set_tx_intr_en

rmt_set_tx_loop

rmt_set_tx_thr_intr_en

rmt_translator_get_context

rmt_translator_init

rmt_translator_se

rmt_tx_memory_reset (channel -- fl)

C'est un mot qui fait exactement ce qu'il dit, mais il est souvent mal compris. Si `rmt_driver_install` est le chantier et `rmt_config` les plans, **`rmt_tx_memory_reset`** est le bouton "Rembobiner" du magnétophone interne.

Pour comprendre son utilité, il faut regarder comment l'ESP32 gère sa mémoire interne dédiée au RMT. Le périphérique RMT possède sa propre petite zone de RAM (512 mots de 32 bits). Lors de l’envoi des données, le matériel utilise un **index interne** pour savoir où il en est dans cette mémoire.

Si on envoie un message, puis un autre, sans "réinitialiser", le pointeur pourrait ne pas revenir automatiquement au début de la zone mémoire allouée au canal. `rmt_tx_memory_reset` force ce pointeur à revenir à l'adresse **0** du bloc mémoire de ton canal.

Dans la plupart des cas, les fonctions de haut niveau comme `rmt_write_items` gèrent cela pour vous. Mais ce mot devient indispensable si :

- **On utilise `rmt_fill_tx_items` :** Si on remplit manuellement la mémoire du RMT par petits morceaux, on a besoin de réinitialiser le pointeur avant de commencer un nouveau "paquet" pour être sûr de ne pas écrire à la suite des anciennes données.
- **Après une erreur ou une interruption :** Si une transmission a planté ou a été stoppée brutalement, la mémoire peut être dans un état incohérent. Un petit coup de "reset" permet de repartir sur une base saine.
- **Optimisation manuelle :** Si on fait du pilotage de précision (genre du bit-banging ultra-rapide via RMT), pour contrôler exactement où tes données sont stockées dans la RAM du périphérique.

C'est une fonction très simple qui ne prend que le numéro du canal. Exemple :

```
\ Remettre à zéro le pointeur mémoire TX du canal 0
0 rmt_tx_memory_reset ( -- esp_err_t )

dup 0 <> if
    ." Erreur Reset Mémoire ! " . cr
then
```

Une petite précision technique : ce mot ne "vide" pas physiquement la RAM (il ne met pas des zéros partout). Il se contente de **déplacer le curseur d'écriture**. Les anciennes données sont toujours là, mais elles seront écrasées dès l’envoi de nouveaux items.

Il existe son jumeau, `rmt_rx_memory_reset`, qui fait exactement la même chose pour le mode réception (utile pour vider le tampon avant d'écouter un nouveau signal).

rmt_tx_start

rmt_tx_stop

rmt_wait_tx_done

rmt_write_it

rmt_write_sample

Code source rmt.h

```
  YV(rmt, rmt_set_rx_idle_thresh, n0 = rmt_set_rx_idle_thresh((rmt_channel_t)
n1, n0); NIP) \
  YV(rmt, rmt_get_rx_idle_thresh, \
    n0 = rmt_get_rx_idle_thresh((rmt_channel_t) n1, (uint16_t *) a0); NIP) \
  YV(rmt, rmt_set_mem_block_num, n0 = rmt_set_mem_block_num((rmt_channel_t) n1,
n0); NIP) \
  YV(rmt, rmt_get_mem_block_num, n0 = rmt_get_mem_block_num((rmt_channel_t) n1,
b0); NIP) \
  YV(rmt, rmt_set_tx_carrier, n0 = rmt_set_tx_carrier((rmt_channel_t) n4, n3,
n2, n1, (rmt_carrier_level_t) n0); NIPn(4)) \
  YV(rmt, rmt_set_mem_pd, n0 = rmt_set_mem_pd((rmt_channel_t) n1, n0); NIP) \
  YV(rmt, rmt_get_mem_pd, n0 = rmt_get_mem_pd((rmt_channel_t) n1, (bool *) a0);
NIP) \
  YV(rmt, rmt_tx_start, n0 = rmt_tx_start((rmt_channel_t) n1, n0); NIP) \
  YV(rmt, rmt_tx_stop, n0 = rmt_tx_stop((rmt_channel_t) n0)) \
  YV(rmt, rmt_rx_start, n0 = rmt_rx_start((rmt_channel_t) n1, n0); NIP) \
  YV(rmt, rmt_rx_stop, n0 = rmt_rx_stop((rmt_channel_t) n0)) \
  YV(rmt, rmt_rx_memory_reset, n0 = rmt_rx_memory_reset((rmt_channel_t) n0)) \
  YV(rmt, rmt_set_memory_owner, n0 = rmt_set_memory_owner((rmt_channel_t) n1,
(rmt_mem_owner_t) n0); NIP) \
  YV(rmt, rmt_get_memory_owner, n0 = rmt_get_memory_owner((rmt_channel_t) n1,
(rmt_mem_owner_t *) a0); NIP) \
  YV(rmt, rmt_set_tx_loop_mode, n0 = rmt_set_tx_loop_mode((rmt_channel_t) n1,
n0); NIP) \
  YV(rmt, rmt_get_tx_loop_mode, n0 = rmt_get_tx_loop_mode((rmt_channel_t) n1,
(bool *) a0); NIP) \
  YV(rmt, rmt_set_rx_filter, n0 = rmt_set_rx_filter((rmt_channel_t) n2, n1, n0);
NIPn(2)) \
  YV(rmt, rmt_set_source_clk, n0 = rmt_set_source_clk((rmt_channel_t) n1,
(rmt_source_clk_t) n0); NIP) \
  YV(rmt, rmt_get_source_clk, n0 = rmt_get_source_clk((rmt_channel_t) n1,
(rmt_source_clk_t * ) a0); NIP) \
  YV(rmt, rmt_set_idle_level, n0 = rmt_set_idle_level((rmt_channel_t) n2, n1, \
        (rmt_idle_level_t) n0); NIPn(2)) \
  YV(rmt, rmt_get_idle_level, n0 = rmt_get_idle_level((rmt_channel_t) n2, \
        (bool *) a1, (rmt_idle_level_t *) a0); NIPn(2)) \
  YV(rmt, rmt_get_status, n0 = rmt_get_status((rmt_channel_t) n1, (uint32_t *)
a0); NIP) \
  YV(rmt, rmt_set_rx_intr_en, n0 = rmt_set_rx_intr_en((rmt_channel_t) n1, n0);
NIP) \
  YV(rmt, rmt_set_err_intr_en, n0 = rmt_set_err_intr_en((rmt_channel_t) n1,
(rmt_mode_t) n0); NIP) \
  YV(rmt, rmt_set_tx_intr_en, n0 = rmt_set_tx_intr_en((rmt_channel_t) n1, n0);
NIP) \
  YV(rmt, rmt_set_tx_thr_intr_en, n0 = rmt_set_tx_thr_intr_en((rmt_channel_t)
n2, n1, n0); NIPn(2)) \
  YV(rmt, rmt_set_gpio, n0 = rmt_set_gpio((rmt_channel_t) n3, (rmt_mode_t) n2,
(gpio_num_t) n1, n0); NIPn(3)) \
  YV(rmt, rmt_config, n0 = rmt_config((const rmt_config_t *) a0)) \
  YV(rmt, rmt_isr_register, n0 = rmt_isr_register((void (*)(void*)) a3, a2, n1,
(rmt_isr_handle_t *) a0); NIPn(3)) \
  YV(rmt, rmt_isr_deregister, n0 = rmt_isr_deregister((rmt_isr_handle_t) n0)) \
  YV(rmt, rmt_fill_tx_items, n0 = rmt_fill_tx_items((rmt_channel_t) n3, \
        (rmt_item32_t *) a2, n1, n0); NIPn(3)) \
  YV(rmt, rmt_driver_install, n0 = rmt_driver_install((rmt_channel_t) n2, n1,
n0); NIPn(2)) \
  YV(rmt, rmt_driver_uinstall, n0 = rmt_driver_uninstall((rmt_channel_t) n0)) \
  YV(rmt, rmt_get_channel_status, n0 =
rmt_get_channel_status((rmt_channel_status_result_t *) a0)) \
  YV(rmt, rmt_get_counter_clock, n0 = rmt_get_counter_clock((rmt_channel_t) n1,
(uint32_t *) a0); NIP) \
```

```
  YV(rmt, rmt_write_items, n0 = rmt_write_items((rmt_channel_t) n3,
(rmt_item32_t *) a2, n1, n0); NIPn(3)) \
  YV(rmt, rmt_wait_tx_done, n0 = rmt_wait_tx_done((rmt_channel_t) n1, n0); NIP)
\
  YV(rmt, rmt_get_ringbuf_handle, n0 = rmt_get_ringbuf_handle((rmt_channel_t)
n1, (RingbufHandle_t *) a0); NIP) \
  YV(rmt, rmt_translator_init, n0 = rmt_translator_init((rmt_channel_t) n1,
(sample_to_rmt_t) n0); NIP) \
  YV(rmt, rmt_translator_set_context, n0 =
rmt_translator_set_context((rmt_channel_t) n1, a0); NIP) \
  YV(rmt, rmt_translator_get_context, n0 = rmt_translator_get_context((const
size_t *) a1, (void **) a0); NIP) \
  YV(rmt, rmt_write_sample, n0 = rmt_write_sample((rmt_channel_t) n3, b2, n1,
n0); NIPn(3))
```

Fastled

```
/*
 * FastLED_min - Minimal WS2812B Library for ESP32
 * Memory efficient alternative to FastLED
 * Uses ESP32 RMT peripheral for precise timing
 *
 * Based on FastLED's proven timing values
 * Supports: WS2812B, WS2812, WS2811 compatible LEDs
 */

#ifndef FASTLED_MIN_H
#define FASTLED_MIN_H

#include <Arduino.h>
#include <driver/rmt.h>

// WS2812B timing constants (FastLED proven values)
// 80MHz RMT clock = 12.5ns per tick
#define T0H_TICKS 32   // 0 code high time (400ns)
#define T1H_TICKS 64   // 1 code high time (800ns)
#define TL_TICKS  52   // Both low times (650ns average)

class CRGB {
public:
  union {
    struct {
      uint8_t r;
      uint8_t g;
      uint8_t b;
    };
    uint8_t raw[3];
  };

  // Constructors
  CRGB() : r(0), g(0), b(0) {}
  CRGB(uint8_t red, uint8_t green, uint8_t blue) : r(red), g(green), b(blue) {}

  // Named colors
  static const CRGB Black;
  static const CRGB Red;
  static const CRGB Green;
  static const CRGB Blue;
  static const CRGB Yellow;
  static const CRGB Cyan;
  static const CRGB Magenta;
```

```
  static const CRGB White;
  static const CRGB Orange;
  static const CRGB Purple;
  static const CRGB Pink;

  // Operators
  CRGB& operator=(const CRGB& rhs) {
    r = rhs.r;
    g = rhs.g;
    b = rhs.b;
    return *this;
  }

  // Array access
  uint8_t& operator[](uint8_t index) {
    return raw[index];
  }
};

// Define named colors
inline const CRGB CRGB::Black = CRGB(0, 0, 0);
inline const CRGB CRGB::Red = CRGB(255, 0, 0);
inline const CRGB CRGB::Green = CRGB(0, 255, 0);
inline const CRGB CRGB::Blue = CRGB(0, 0, 255);
inline const CRGB CRGB::Yellow = CRGB(255, 255, 0);
inline const CRGB CRGB::Cyan = CRGB(0, 255, 255);
inline const CRGB CRGB::Magenta = CRGB(255, 0, 255);
inline const CRGB CRGB::White = CRGB(255, 255, 255);
inline const CRGB CRGB::Orange = CRGB(255, 165, 0);
inline const CRGB CRGB::Purple = CRGB(128, 0, 128);
inline const CRGB CRGB::Pink = CRGB(255, 192, 203);

template<uint8_t DATA_PIN>
class CFastLED_min {
private:
  CRGB* leds;
  uint16_t numLeds;
  uint8_t brightness;
  rmt_channel_t rmtChannel;
  bool initialized;

  void sendPixel(uint8_t r, uint8_t g, uint8_t b) {
    rmt_item32_t items[24];

    // Apply brightness
    r = (r * brightness) >> 8;
    g = (g * brightness) >> 8;
    b = (b * brightness) >> 8;

    // Pack GRB order (WS2812B format)
    uint32_t color = (g << 16) | (r << 8) | b;

    // Convert to RMT format
    for (int i = 0; i < 24; i++) {
      bool bit = color & (1 << (23 - i));
      items[i].level0 = 1;
      items[i].duration0 = bit ? T1H_TICKS : T0H_TICKS;
      items[i].level1 = 0;
      items[i].duration1 = TL_TICKS;
    }

    // Send
    rmt_write_items(rmtChannel, items, 24, true);
    rmt_wait_tx_done(rmtChannel, pdMS_TO_TICKS(100));
```

```
  }

public:
  CFastLED_min() : leds(nullptr), numLeds(0), brightness(255),
                   rmtChannel(RMT_CHANNEL_0), initialized(false) {}

  void addLeds(CRGB* ledArray, uint16_t count) {
    leds = ledArray;
    numLeds = count;

    // RMT configuration
    rmt_config_t config = {};
    config.rmt_mode = RMT_MODE_TX;
    config.channel = rmtChannel;
    config.gpio_num = (gpio_num_t)DATA_PIN;
    config.clk_div = 1;  // 80MHz
    config.mem_block_num = 1;
    config.tx_config.loop_en = false;
    config.tx_config.carrier_en = false;
    config.tx_config.idle_output_en = true;
    config.tx_config.idle_level = RMT_IDLE_LEVEL_LOW;

    rmt_config(&config);
    rmt_driver_install(config.channel, 0, 0);

    initialized = true;
  }

  void setBrightness(uint8_t b) {
    brightness = b;
  }

  void show() {
    if (!initialized || leds == nullptr) return;

    for (uint16_t i = 0; i < numLeds; i++) {
      sendPixel(leds[i].r, leds[i].g, leds[i].b);
    }
  }

  void clear() {
    if (leds == nullptr) return;
    for (uint16_t i = 0; i < numLeds; i++) {
      leds[i] = CRGB::Black;
    }
  }

  CRGB& operator[](uint16_t index) {
    return leds[index];
  }
};

// Global instance (FastLED-style API)
template<uint8_t DATA_PIN>
CFastLED_min<DATA_PIN> FastLED_min;

// Macro for easy setup (FastLED-style)
#define FASTLED_MIN_SETUP(PIN, LEDS, COUNT) \
  FastLED_min<PIN>.addLeds(LEDS, COUNT)

#endif // FASTLED_MIN_H
```

Phase d’installation du driver rmt

Avant de configurer un canal, il faut préparer le terrain pour le pilote global.

- **`rmt_driver_install`** : C'est la toute première étape. Elle alloue les ressources mémoires (Ring Buffer) et installe les gestionnaires d'interruptions système.
- **`rmt_config`** : Comme on l'a vu, elle applique tes paramètres (`&config`) au canal choisi.

Une fois configuré, on lance ou arrête le signal.

- **`rmt_tx_start`** / **`rmt_tx_stop`** : Déclenche ou fige l'envoi de données sur le fil.
- **`rmt_write_items`** : C'est le mot "star" pour l'émission. Tu lui donnes une adresse de données (souvent des impulsions au format `rmt_item32_t`) et il s'occupe de les envoyer.
- **`rmt_wait_tx_done`** : Crucial en Forth ! Comme le RMT travaille en arrière-plan (non-bloquant), ce mot permet d'attendre que la transmission soit finie avant de faire autre chose.

Le réglage fin (Tuning)

Ces mots servent à modifier les paramètres "à la volée" sans refaire toute la config :

- **`rmt_set_clk_div`** : Pour changer la vitesse du métronome.
- **`rmt_set_idle_level`** : Pour décider si la broche doit être à 0 ou à 1 quand on n'envoie rien (très important pour l'infra-rouge).
- **`rmt_set_rx_idle_thresh`** : Pour la réception, définit après combien de temps de silence on considère que le message est terminé.

La gestion de la mémoire

Le RMT possède sa propre petite RAM interne (512 mots de 32 bits partagés entre les canaux).

- **`rmt_set_mem_block_num`** : Si tu veux envoyer de très longs messages, tu peux dire à un canal d'utiliser plusieurs blocs de mémoire.
- **`rmt_tx_memory_reset`** : Nettoie la RAM du canal pour repartir sur une base saine.

Exemple de "Pipeline" en Forth

Voici l'ordre dans lequel tu devrais normalement appeler ces mots :

1. `... rmt_driver_install` (Une seule fois au démarrage)
2. `&config rmt_config` (Pour préparer ton canal)
3. `mon_buffer nb_items wait_flag rmt_write_items` (Pour envoyer)
4. `canal timeout rmt_wait_tx_done` (Pour attendre la fin)

Un petit conseil sur `rmt_item32_t`. La plupart des mots (comme `rmt_write_items` ou `rmt_fill_tx_items`) vont manipuler des **items**. Un "item" RMT en mémoire, c'est un mot de 32 bits qui contient :

- La durée du niveau haut (15 bits) + son état.
- La durée du niveau bas (15 bits) + son état.

Le projet WS2812

….@TODO : à rédiger