ESP32forth and the RMT library

Marc PETREMANN

Version 1.0 - February 16, 2026

Table of contents

Preamble

The purpose of this manual is to give you all the information, as well as practical examples, to fully utilize the RMT 2.0 library.

The move to **RMT 2.0** (introduced with the ESP-IDF v5) marks a transition from a "static" driver to a **dynamic, object-oriented architecture** . Previously, fixed channel numbers (0-7) and manual clock divider registers were manipulated via ``` `rmt_set_clk_div` ``` . Now, everything relies on Handles **(**`rmt_channel_handle_t`) dynamically allocated by the system, ensuring better portability between chips (ESP32, S3, C3).

Signal handling has also changed radically: instead of simply filling an array of items, RMT Encoders are used to **translate** raw data into real-time waveforms. This version also introduces the **Sync Manager for perfectly synchronizing multiple channels, and native DMA** integration to free up the CPU during massive data transfers. In short, RMT 2.0 is more complex `to` initialize, but much more robust, flexible, and efficient in terms of CPU resources for complex protocols.

Note – detailed documentation:
https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/rmt.html#rmt-resource-allocation

The structures

rmt_sync_manager_config_t

rmt_sync_manager_config_t structure is a major new feature of the RMT 2.0 driver (Next Gen).

It serves one specific purpose: **the synchronization of multiple channels** .

In the old driver, it was very difficult to trigger the sending of two signals on two different pins at exactly the same time. With the **Sync Manager** , you can group several TX channel handles and tell them: "All start together on the next clock pulse".

This is essential if you are building:

- A controller for a giant LED matrix (multiple strips in parallel).
- A proprietary parallel communication protocol.
- A stepper motor control system where the pulses must be perfectly aligned.

rmt_transmit_config_t

rmt_transmit_config_t structure is a small configuration structure used only at the time of the `rmt_transmit call` . It allows you to define **how** the specific message should be sent (repetition, end behavior, etc.).
In the RMT 2.0 driver (IDF v5.x), it is very lightweight. Here is its structure:

Field C	Kind	Description
loop_count	`int`	Number of times the message has been repeated. • `0` : Simple shipment (default). • `-1` : Infinite loop. • `>0` : Specific number of repetitions.
flags	`struct`	Contains specific settings in bit form.

Within the `flags field` , you will mainly find:

- **eot_zealous** (1 bit):
    - If set to `1` (true), the channel will force the signal to the inactive state immediately after the last symbol.
    - If set to `0` , there may be a slight latency depending on the state of the hardware.

    - *Usefulness:* Crucial for protocols that are very sensitive to end timing, such as certain LED strips or infrared.

If you want to manipulate this structure directly from Forth, here's how to define it to match the processor's 32-bit memory alignment:

Code excerpt

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

In 90% of cases, you want to send a frame only once without any unusual settings. That's why the `rmt_transmit function accepts a` **NULL** pointer (or `0` in Forth) instead of this structure.

Practical tip: If you don't need to loop your signal (such as when sending a continuous infrared code), don't bother creating this structure. Simply pass `0` as the last argument to your `rmt_transmit macro` .

If you want to send a signal in a loop 5 times:

```
variable my-config
rmt_transmit_config_t allot my-config !

5 ma-config ->loop_count !
0 ma-config ->flags_packed ! \ No special flags

Switching to macro:
\ handle encoder payload len ma-config rmt_transmit
```

rmt_tx_channel_config_t

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

rmt_tx_channel_config_t structure is the "blueprint" for your transmission channel. It defines the permanent hardware characteristics of the channel at the time of its creation with `rmt_new_tx_channel` .

Here are the details of the fields for **Core 3.x (RMT 2.0)** , as they differ greatly from older versions.

Hardware Configuration (The Basics):

- **gpio_num** (`int`) : The physical GPIO number where the signal will come out.
- **clk_src** (`enum`): The clock source. Usually set to `0` (RMT_CLK_SRC_DEFAULT) to use the system clock (APB).

- **resolution_hz** (`uint32_t`): **The most important field.** This is the frequency of your clock "tick".
    - *Example:* If you set `10,000,000` (10 MHz), each unit of time of your future signals will be worth 1/107 = 100ns.

Memory Management and Queue:

- **mem_block_symbols** (`size_t`): Size of the internal memory buffer (in number of RMT symbols). The larger this number, the longer the frames you can send without interruption.
- **trans_queue_depth** (`size_t`): Number of transmissions you can "queue" (pile) before `rmt_transmit` becomes blocking.
- **intr_priority** (`int`): Hardware interrupt priority (0 to let the system choose).

The "Flags" block (Compact Structure), these fields are grouped into a single 32-bit word in memory:

- **invert_out** : Inverts the signal (High becomes Low and vice versa).
- **with_dma** : Indicates whether the channel should use the DMA (Direct Memory Access) controller. Essential for very long frames (thousands of LEDs).
- **io_loop_back** : Test mode where the output is internally fed back to the input.
- **io_od_mode** : Configures the output to "Open Drain".
- **allow_pd** : Allows the device to be powered off in deep sleep mode to save energy.
- **init_level** : Defines whether the pin is at 0 or 1 when the channel is at rest.

For your `struct` Forth to work with C++, here is the exact order you must follow in your definition:

Champ Forth	Type C	Size
`->gpio_num`	`int`	4 bytes
`->clk_src`	`int`	4 bytes
`->resolution_hz`	`uint32_t`	4 bytes
`->mem_block_symbols`	`size_t`	4 bytes
`->trans_queue_depth`	`size_t`	4 bytes
`->intr_priority`	`int`	4 bytes
`->flags_packed`	`bitfield`	4 bytes

Warning: If you forget the `intr_priority field` or if you separate the `flags` , the memory offset will be wrong and `rmt_new_tx_channel` will receive inconsistent data (such as a 0 Hz resolution or an impossible GPIO).

rmt_tx_event_callbacks_t

The words of the rmt vocabulary

List of all defined words:

```
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)

The word **rmt_apply_carrier** is the tool that allows you to modulate your RMT signal with a **carrier frequency** .

This is an essential function if you are working on **Infrared (IR) home automation** . IR receivers (like the well-known TSOP) do not react to a simple continuous light, but to a light that flashes very quickly (usually at **38 kHz**).

Instead of having a simple "square" signal (High or Low), the carrier will break down each "High" state into a multitude of ultra-fast little pulses.

- **Without carrier signal:** The GPIO remains at 3.3V for the entire duration of the "High" symbol.
- **With carrier signal:** The GPIO oscillates (0V -> 3.3V -> 0V) at the chosen frequency (e.g., 38,000 times per second) for the duration of the "High" symbol.

To use this function, you must pass it a structure that defines the "blinking" rules:

Field	Kind	Description
frequency_hz	`uint32_t`	The carrier frequency (e.g., `38000` for IR).
duty_cycle	`float`	The duty cycle (often `0.33` or `0.5`).
flags	`struct`	`polarity_active_low` : inverts the carrier.

Once the channel is created, you call this function to "arm" the modulation.

- If `config` is a valid pointer: The carrier is **enabled** .
- If `config` is **NULL** (or `0` in Forth): The carrier is **disabled** .

A typical use case, if you want to control a television:

- You are creating a standard TX channel.
- You apply a 38 kHz carrier with `rmt_apply_carrier` .
- You send your data with `rmt_transmit` . The hardware takes care of "chopping" the signal at the correct frequency without your CPU having to lift a finger.

rmt_del_channel (handle -- err)

The **rmt_del_channel command** is the final stage in the lifecycle of an RMT channel. It is responsible for "cleaning up" and releasing hardware and software resources.

When you call this function, the driver performs the following actions:

- **Frees the random access memory (RAM)** allocated for the channel control structures.
- ESP32's **internal memory block (SRAM RMT) that was reserved for your symbols.**
- **Disconnect the GPIO** : The physical pin is no longer controlled by the RMT device.
- **Disable interrupts** : The CPU will no longer monitor events on this channel.

A common cause of errors: it is **forbidden** to delete a channel that is still active. For `rmt_del_channel` to succeed, the channel must be in the **"Disabled" state** .

The correct order is always:

- `rmt_disable`
- `rmt_del_channel`

In Forth, there is a tendency to test small pieces of code, redefine variables and rerun scripts.

- If you create a new channel (`rmt_new_tx_channel`) every time you run your code without deleting the old one, you will **exhaust the resources** of the ESP32.
- The ESP32 has a limited number of RMT channels (usually 4 or 8 depending on the model). Once they are all allocated, `rmt_new_tx_channel` will always return `0` (error) until you delete `them` .

rmt_del_sync_manager (sync_handle -- err)

The word **rmt_del_sync_manager** is a cleanup function specific to the Sync **Manager** of the RMT 2.0 driver.

It only comes into play if you have used the RMT multi-channel synchronization feature.

In some complex projects, you need multiple RMT channels (for example, to control four different LED strips) that start emitting **at precisely the same time** , within the clock cycle. To achieve this, a "synchronization group" is created using `` `rmt_new_sync_manager` `` .

Just as `rmt_del_channel` releases a channel, `rmt_del_sync_manager` :

- **Releases software resources** related to the synchronization group.
- **Disconnect the channels** that were linked to this manager.
- **Returns the memory** consumed by this object to the system.

If you have finished using a group of synchronized channels and want to free up memory or rearrange your channels independently, you must delete the manager.

Note: Removing the Sync Manager does not delete the channels themselves. You will still need to call `rmt_del_channel` for each channel individually.

If you're only controlling a single device (an LED strip, a sensor, or an IR emitter), you'll probably never use the Sync Manager. It's an advanced feature for multi-channel synchronization.

rmt_disable (handle -- err)

rmt_disable() function is the switch that allows a transmission (or reception) channel to be put into "standby" or "rest" mode after use.

Here are its roles and how it works in the RMT 2.0 driver:

When you call `rmt_disable(handle)` , the driver performs the following operations behind the scenes:

- **Clock Gating:** This stops the clock source (APB, XTAL, or RC) that powers the channel. This immediately stops all unnecessary power consumption.
- **Interrupt Release:** It disables interrupt vectors associated with the channel so that the CPU is no longer called upon by end-of-transmission events.
- **State protection:** The channel is no longer "ready" to send. If you call `rmt_transmit` on a disabled channel, you will receive an `ESP_ERR_INVALID_STATE error` .

In a well-structured program, `rmt_disable` is located between active use and destruction:

1. **Enable** : The channel "wakes up" and consumes power.
2. **Transmitted** : The channel is working.
3. **Wait Done** : We are waiting for the work to be finished.
4. **Disable** : The channel "goes to sleep" (Energy saving).

Unlike older, simpler microcontrollers where a peripheral is always "on", the ESP32 manages its power very finely.

- **On battery:** Leaving an RMT channel unnecessarily activated prevents certain light sleep modes and consumes a few milliamps continuously.
- **Signal security:** Disabling the channel ensures that no spurious pulses will be sent on the GPIO if the memory is manipulated in error.

Disable and **Delete** should not be confused :

- **rmt_disable** : Turns off the engine, but the car remains parked (the configuration remains in RAM).
- **rmt_del_channel** : Sends the car to the junkyard (completely frees up memory and handle).

Note: You must call `rmt_disable` before you can do an `rmt_del_channel` .

rmt_enable(handle -- err)

The word **rmt_enable** is an essential transition step in the lifecycle of an RMT 2.0 channel. It allows the channel to move from the "Configured" state to the "Ready to transmit" state.

When you create a channel with `rmt_new_tx_channel` , the system allocates resources (memory, hardware channel), but the device is kept in a **low-power state** .

rmt_enable performs the following actions:

- **Activates the internal clock** of the RMT device.
- **Powers the hardware block** (Power Domain).
- **Allows interruptions** related to this channel.
- **Places the GPIO** in its initial state (`init_level`).

It is important to understand where it is located in your Forth code:

1. **rmt_new_tx_channel** : Creation (The channel is **IDLE** / Inactive).
2. **rmt_enable** : Activation (The channel is **READY**).
3. **rmt_transmit** : Action (The channel is **BUSY** / Sending).
4. **rmt_disable** : Sleep (The channel returns to **IDLE** to save energy).

This separation allows for precise energy management. On a battery-powered ESP32, you can create your channels at startup, but only "enable" them `just` before sending a frame, and then "disable" them `immediately` afterwards to cut the clock power consumption.

If you attempt to call `rmt_transmit on a channel that has not been "enabled", the function will return an ESP_ERR_INVALID_STATE (0x103)` error and nothing will come out on your GPIO pin.

rmt_new_sync_manager

rmt_new_rx_channel (config -- err)

This function is the counterpart to `rmt_new_tx_channel` , but for **reception** . It instructs the system to allocate the necessary hardware resources to listen to an incoming signal on a GPIO pin and convert it into RMT symbols.

In Core 3.x (RMT 2.0), reception has become much more powerful, notably thanks to better DMA management and noise filters.

The parameters:

- **config** : A pointer to the configuration structure (GPIO, clock, filters). This is where you define "how" you want to receive.
- **ret_chan** : A pointer to a variable of type `rmt_channel_handle_t` . If the allocation succeeds, the system writes the unique identifier of the receiving channel to it.

For this to work in **ESP32forth** , you must define this structure precisely. It differs slightly from the TX version:

Champ Forth	Type C	Size	Description
`->gpio_num`	`int`	4 bytes	The input GPIO.
`->clk_src`	`int`	4 bytes	Clock source (0 by default).
`->resolution_hz`	`uint32_t`	4 bytes	Accuracy of time measurement (e.g., 1 MHz = 1 µs).
`->mem_block_symbols`	`size_t`	4 bytes	Receive buffer size.
`->trans_queue_depth`	`size_t`	4 bytes	Number of receptions that may be pending.
`->intr_priority`	`int`	4 bytes	Interruption priority.
->**flags_packed**	`u32`	4 bytes	**Bit 0** : Inversion, **Bit 1** : DMA, **Bit 2** : Power Down.

The new features of RMT 2.0 for reception, unlike the old API, are that reception 2.0 is designed to be "comfortable":

1. **Glitch Filter** : Very important for ignoring glitches. (Configured via another function: `rmt_apply_rx_glitch_filter`).
2. **Idle Threshold** : You define after how long of silence (without a change of state) the frame is considered finished.
3. **Event-driven management** : You can attach "callbacks" that execute as soon as a frame is received, without the Forth needing to constantly poll the device.

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);

The word **rmt_new_tx_channel** is the essential entry point for the new RMT API (v2.0 / Next Gen) present in your Core 3.3.5. It replaces the old `rmt_config` .

Its role is simple: it asks the system to allocate a free RMT hardware resource and apply your configuration to it.

The parameters:

1. **config** : A pointer to the structure we defined in Forth. It contains the GPIO, the resolution, and the famous "flags".
2. **ret_chan** : This is where the function will write the "Handle" (the identifier) of the channel it has assigned to you. This is the handle that you will use for all other functions (sending data, stopping the channel, etc.).

Unlike the old version where you manually selected `RMT_CHANNEL_0` , this function is smarter:

- **Resource management** : It is looking for a free channel (the ESP32 has 8, but the S3 only has 4).
- **Clock arbitration** : It checks that the requested clock source is compatible with other channels already open.
- **GPIO Initialization** : This configures the internal multiplexer (GPIO Matrix) to connect the RMT device to your physical pin.
- **Memory protection** : It prepares access to the RMT RAM (symbol blocks).

The transition to this function marks the end of the "static" model.

- **Previously** : A fixed channel was configured, often with rigid global variables.
- **Now** : This is a **dynamic approach** . Channels can be created and deleted on the fly, which is much cleaner for an interactive language like Forth.

Once the channel is created with this function, it is in an "IDLE" (resting) state. To send a signal, you will no longer be able to send simple "items" directly; you will have to create an **Encoder** and attach it to this handle.

rmt_receive (rx_channel buffer buffSize config -- err)

The word **rmt_receive** is the engine of data capture in the RMT 2.0 driver. It allows a listening operation to be launched on a previously configured receive channel.

Settings:

- **rx_channel** : The handle of the RX channel (created with `rmt_new_rx_channel`).
- **buffer** : The memory address (in Forth, the address of your array) where the received data will be stored.
  - *Note:* Unlike TX, RMT writes **rmt_symbol_word_t here** . Each symbol occupies 4 bytes (2 bytes for duration, 1 bit for level).
- **buffer_size** : The maximum buffer size in bytes.
- **config : A pointer to an** `rmt_receive_config_t` structure . You can pass `NULL` (or `0` in Forth) for the default settings.

Unlike TX which uses an "Encoder", RX is more direct: it transforms the changes of state on the GPIO pin into a sequence of **durations** .

1. **Launch** : You call `rmt_receive` . The channel becomes alert.
2. **Capture** : As soon as a front (rising or falling) is detected, the internal timer measures the time until the next front.
3. **Storage** : Each segment (High or Low) is converted into a symbol and written to your buffer.
4. **End** : Reception stops either when the buffer is full, or when the signal remains stable longer than the "idle threshold" `idle_threshold_ticks` .

Key features:

- **Non-blocking** : As with transmission, this function initiates reception and immediately returns control to the Forth. It does not "block" the interpreter while waiting for the signal.
- **Synchronization** : To know when reception is complete, you must use a wait function `rmt_wait_rx_done` , otherwise you will read an empty or incomplete buffer.

It is crucial to understand what you will read in your Forth buffer after an `rmt_receive` . An RMT symbol is encoded on 32 bits:

- **15 bits** : Duration of the first part of the symbol.
- **1 bit** : Logic level of the first part (0 or 1).
- **15 bits** : Duration of the second part.
- **1 bit** : Logic level of the second part.

rmt_rx_register_event_callbacks

This is the function that makes your reception "intelligent". As we've seen, `rmt_receive` is asynchronous: it starts the capture and then stops.
rmt_rx_register_event_callbacks tells the ESP32: *"As soon as you've finished receiving a frame, execute this specific function"* .

It's the bridge between the hardware (the incoming signal) and your software logic.

Without callbacks, you would be forced to perform polling (looping endlessly to check if it's finished), which wastes CPU and lacks precision. With a callback:

1. The equipment receives the signal.
2. An interruption is triggered.
3. The driver calls your function (the callback).
4. Your function can raise a flag or release a semaphore that your Forth code is monitoring.

You are not passing a single function, but an `rmt_rx_event_callbacks_t structure` which can contain multiple events:

- **on_recv_done** : Called when a receive transaction is successfully completed.
- **on_recv_error** : Called if an error occurs (e.g., buffer too small).

The parameters:

- **rx_channel** : Your receive handle.
- **cbs** : Pointer to the structure containing the addresses of your functions
- **user_data** : An optional pointer to your own data (very useful in Forth to pass the address of a variable `or` a `value`).

Once saved, you no longer need to create a blind `while loop` . You do:

1. `rmt_receive`

2. Wait until `rx_done_flag` becomes `true` (in Forth: `BEGIN rx-flag @ UNTIL`).
3. Process the data and reset `rx-flag` to `false` .

rmt_wait_rx_done (handle timeout_ms -- err)

Unlike transmission (`TX`), where there is a dedicated wait function, **reception (RX)** in the modern Espressif driver (v5.x) relies on a **Task Notification** or **Queue system. There is no native** ``` `rmt_rx_wait_done` ``` function directly in the header.

Here the word `rmt_wait_rx_done` is a simple solution for FORTH.

When you create an RX channel, you can ask it to send you a signal (via a FreeRTOS queue) as soon as a frame is ready.

Note : this word should be used with caution.

rmt_sync_reset (sync_handle -- err)

The term **rmt_sync_reset** is a "synchronized reset" command. It is inseparable from the **Sync Manager** we discussed earlier.

Its role is to ensure that the timers of several RMT channels are perfectly aligned to the same starting value before starting a broadcast.

Normally, each RMT channel has its own internal counter. If you launch four channels one after the other in Forth mode, there will always be a few microseconds of delay between the first and last due to CPU execution time.

For precision applications (such as driving giant LED arrays or synchronized motor control signals), this offset is unacceptable.

When you call `rmt_sync_reset` , the RMT device:

1. **Blocks** the clocks of channels belonging to the group.
2. **Resets** all their counters to zero simultaneously.
3. **Release** the clocks so they start again at exactly the same "tick".

The workflow with this word looks like this:

1. Create the channels and add them to the Sync Manager.
2. Call **rmt_sync_reset** .
3. Start the transmissions (they will all start on the same clock edge).

If you use synchronization, you will need this trio:

- `rmt_new_sync_manager` : Create the group.

- **rmt_sync_reset** : Align the clocks (your question here).
- `rmt_del_sync_manager` : Destroy the group.

One important point to note: If you are only managing a single channel (a single GPIO output), this function is useless. It is only there to coordinate a "range" of channels.

rmt_transmit (handle encoder payload len config -- err)

This is the **central function** of the RMT 2.0 driver. If `rmt_new_tx_channel` prepares the motor, `rmt_transmit` is the accelerator pedal: it is the one that physically sends the data to the GPIO output.

Here are the details of its 5 parameters, essential for your Forth implementation:

- **tx_channel (The channel)**
  This is the **handle** (the identifier) that you retrieved when creating the channel. It tells the driver which RMT device to use (and therefore which GPIO pin to send the signal to).
- **Encoder (The translator)**
  This is the major new feature of RMT 2.0. The encoder is a device that can transform your raw data (your `payload`) into electrical pulses (high/low).
  Example: If you send bytes for WS2812 LEDs, the encoder transforms the bit "1" into a 0.8µs High / 0.4µs Low.
- `The payload` (data)
  is a pointer to your data in Forth memory. It can be anything: a byte array, a string, or a complex structure. The function doesn't try to understand what's inside; it simply passes this pointer to the encoder.
- `payload_bytes` (Size)
  Indicate here the total size (in bytes) of the data pointed to by the `payload` . This allows the driver to know when to stop.
- **config**
  In the transmission options; we pass a pointer to an `rmt_transmit_config_t structure` .
  - For example, it allows you to define whether the sending should be done in a loop (`loop_count`).
  - **Tip:** you can pass `NULL` (or `0` in Forth) to use the default settings (simple, one-time sending).

Unlike the previous version, `rmt_transmit` **does not wait** for the signal to finish before handing control back to the Forth code.

1. It places the request in a queue.
2. She initiates the transfer (often via DMA).
3. She immediately gives control back to the interpreter Forth.

Since the function is non-blocking, if we modify the `payload` (the data array) immediately after calling `rmt.send` in Forth, we risk corrupting the signal being sent.

You must use `rmt_tx_wait_all_done` to ensure that the sending is complete before proceeding.

rmt_tx_register_event_callbacks

rmt_tx_switch_gpio

rmt_tx_wait_all_done

The words for WS2812

These are words defined for a specific application. They are defined in the `RMT vocabulary` but are extensions of the RMT20 API.

rmt_new_ws2812_encoder (resolution_hz -- encoder|0)

This code is crucial because it transforms the RMT device (which is a simple pulse generator) into an intelligent controller capable of speaking the "language" of WS2812 LEDs.

The first parameter retrieved from the Forth stack is the resolution in Hz (e.g., **1,000,000** for 1 MHz).

- The code calculates the pulse durations based on this resolution.
- **A word of caution regarding mathematics** : In code, `(res / 1000000)` equals **1** if the resolution is 1 MHz. Multiplications by `0.4` or `0.85` may be truncated if the compiler treats these as integers. For greater accuracy, it is often preferable to multiply before dividing.

The WS2812 protocol uses the duration of a **HIGH pulse** followed by a **LOW pulse** to differentiate a `0` from a `1` .

Symbol	Total Duration	High State (Level 1)	Low State (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)

The "Bytes" encoder is a translator: it takes one byte (8 bits) and, for each bit, it asks the RMT to generate the two defined durations:

- **level0 = 1 / level1 = 0** : This means that the first part of the pulse is at 3.3V (High) and the second at 0V (Low).
- **msb_first = 1** : The WS2812 LEDs wait for data in the order of the most significant bit to the least significant bit (often Green, then Red, then Blue).

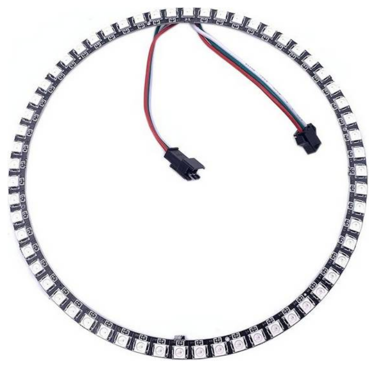
The `rmt_new_bytes_encoder function` creates the object in memory.

- If no error, push the encoder address (the **Handle**) onto the stack.
- Otherwise, stack `0` .

prepare_ws2812_config (gpio -- handle|0)

@TODO: to be written

The WS2812 project



The project involves controlling a ring of 60 LEDs. Why 60 LEDs? Simply because there are 60 minutes in an hour, or 60 seconds in a minute. The challenge, therefore, is to control these LEDs to use them as an analog clock!

Addressable LEDs

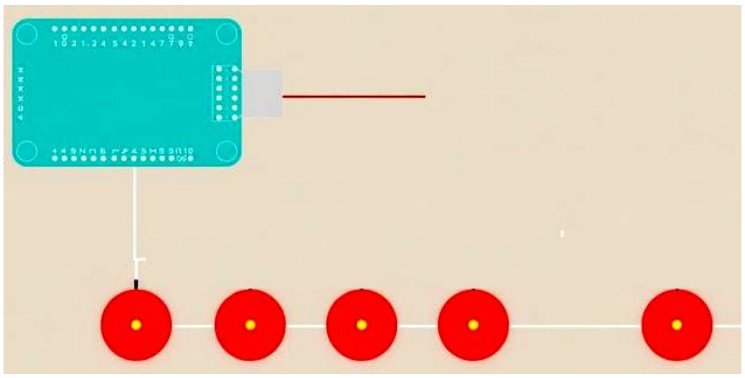
Addressable LEDs are light-emitting diodes where each unit can be controlled independently, allowing for the display of various colors and lighting effects on a single strip or panel. Here's a simple explanation of how they work:

- **Internal structure** : Each addressable LED typically includes a small integrated control chip (such as WS2812, SK6812, WS2801, etc.) inside, in addition to the light-emitting diode. This chip receives and processes data to control the color and brightness.
- **Control via a digital signal** : the LEDs are connected in series. The control signal (data) is sent via a single line from a microcontroller (Arduino, Raspberry Pi, etc.). This signal contains instructions for each LED, one after the other.
- **Cascading transmission** : The microcontroller sends a bit string representing the color (red, green, blue) to the first LED. The first LED reads its bits, saves its parameters, and then transmits the rest of the signal to the next LED, which does the same. Thus, each LED receives its own set of data and displays the specified color.
- **Update and lighting effects** : by modifying the signal sent, you can change the colors of each LED individually, create animations, gradients, twinkling effects, etc.
- **Power supply** : LEDs require a suitable power supply, often 5V, and care must be taken to manage the current when many LEDs are lit at the same time.

In summary, thanks to their integrated control chip and serial communication, addressable LEDs allow precise and flexible control of each luminous pixel in a strip or panel.

In practice, here is a description of the process:

- A control source (microcontroller) sends a data signal.
- The first LED receives the signal, reads its data, displays the color and transmits the rest of the signal to the next LED.
- Each LED has an internal control chip.
- The chain continues until the last LED.



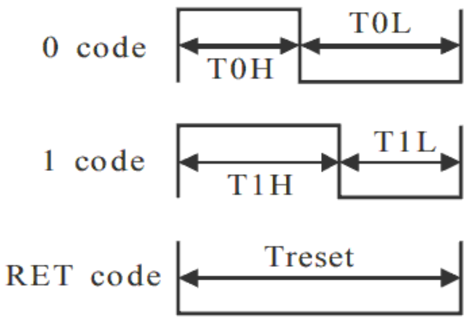
Contribution of the RMT library to the control of addressable LEDs

Addressable LEDs (like the WS2812) require signals with very precise timing to distinguish between "0" and "1" bits. The RMT library can generate these signals with microsecond precision, thus avoiding errors due to traditional software management.

With RMT (Remote Motion Transfer) words, the microcontroller can delegate signal generation to dedicated hardware, freeing up the processor for other tasks. This allows data to be sent to a large number of LEDs without CPU overload.

To drive 60 WS2812B LEDs with the RMT 2.0 driver, the "secret" lies in the clock precision. The WS2812B protocol encodes bits with very precise durations (a total cycle of approximately 1.25µs).

To achieve this precision, the **`RMT vocabulary`** incorporates the term **``` `rmt_new_ws2812_encoder` ```** . This term hardcodes all the parameters specific to controlling addressable LEDs with their WS2812 component.

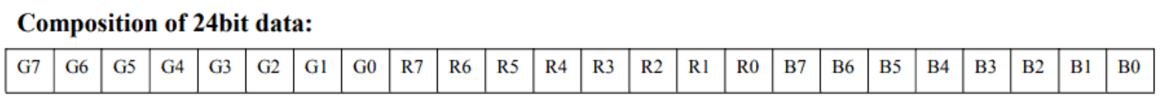


The values T0H, T0L, T1H and T1L are predefined in the word **`rmt_new_ws2812_encoder`** .

The only parameter required for **`rmt_new_ws2812_encoder`** is the resolution value in Hz needed for signal generation. In return, a handle is retrieved for future use.

Encoding for an addressable LED

To control an LED, the data to be transmitted must be configured in a 24-bit space, distributed as follows:


Composition of 24bit data:

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Detail :

- bits G7..G0 which determine the green color (G = Green)
- bits R7..R0 which determine the color red (R = Red)
- bits B7..B0 which determine the blue color (B = Blue)

The order of colors in a 24-bit sequence transmitted to the WS2812 module is **GRB** . This is very important because most color systems refer to RGB (Red Green Blue) colors. We will see later how to handle this difference in color encoding.

To control n LEDs, we will reserve a memory space of n x 3 bytes. For the LED ring, we will therefore reserve 180 bytes:

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

We use the **`RGB_struct structure`** as a multiplier to allocate our 180 bytes. Definition of this structure:

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

Note that our accessors are defined in the order of the **GRB colors** . This is the only place in our entire source code where we define colors in this order. For all other FORTH words, we will use the **RGB color order** .

Defining an LED color

To retrieve the RGB values of a color defined in memory, we define **`RGB@`** :

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

Yes, our structure accessors are indeed in **RGB order** ! That's where the strength of structures lies. Only the order in which the accessors are defined is important. After that, we can modify the order as shown above. If I specify an address, the execution of `RGB@ will retrieve the data stored in` **GRB** order and stack it in **RGB** order .

The intensity of the LEDs is controlled by these definitions;

```
\ 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 */
  ;
```

I have chosen to spread the brightness of a color over an interval of 0..100. This light intensity value is stored in the `INTENSITY value` .

The `setIntensity` keyword will assign a different value to `INTENSITY . For example, if you execute setIntensity 20 times`, all **RGB** values will be adjusted to 20% of their raw value. This is what is done in this definition:

```
\ 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!
  ;
```

This definition will only be used to store an RGB color and its intensity in the data area reserved for transmission to the WS2812 module.

Before we get to that point, here's how to define some basic colors:

```
\ 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
```

Here, the term **RGB_Red** is a kind of basic RGB color constant. To store this color with an intensity of 50%;

```
50 setIntensity
RGB_Red LEDs RGB!
```

GRB values stored in the reserved memory space of LEDS are automatically adjusted according to the required intensity.

LED management for the LED crown

Let's revisit the definition of the LEDS memory space intended to store the colors to be transmitted. For 60 LEDs, we reserve 60 x 3 bytes:

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

From there, it is easy to create the word that will store the three colors of an LED at position n:

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

In this definition, the sequence NB_LEDS 1- min is a safety feature preventing the position from overflowing, confining it to the interval 00..59 (therefore 60 positions in total).

Finally, we define the word that clears our memory space reserved for the 60 LEDs:

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

LED data transmission

The very first step is to define the pin that transmits data in output mode:

```
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
  ;
```

The word **initWS2812** defines the channel whose handle is stored in **channelHandle** , then defines the encoder whose handle is stored in **encoderHandle**.

For purely technical reasons, this structure is also defined, but it will remain empty.

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

And finally, we define the word that transmits the values of our 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
```

The core of this definition is the word `rmt_transmit`. Explanations of this word can be found earlier in this document. This definition relies heavily on values and constants. This programming method allows you to reuse this same definition almost without modification for any type of addressable LED.

Definition of an addressable LED clock

We begin by defining a new structure designed to describe the color **AND** intensity of an LED assigned to a specific use within the set of 60 LEDs in our LED ring.

```
structures also

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

only FORTH
```

This new structure uses the same one used for defining basic colors, but adds the accessor `->intensity`. The following word will retrieve the data for a color from the clock:

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

Here's how we define each point on our clock:

```
\ 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@
  ;
```

The word **CLOCKcolor:** handles four parameters and is followed by the word to be defined. The four parameters correspond to the base color of the clock dot and its intensity:

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

Let's now see how to use these points to highlight fixed minutes and hours on the background of the addressable LED clock:

```
\ A very faint blue crown to mark the minutes
```

```
: setInitMinutes ( -- )
    CLOCK_minutes setIntensity drop drop drop
    60 0 do
       CLOCK_minutes drop i nLED!
    loop
  ;
```

```
\ A least faint blue crown to mark the minutes
```

```
: setInitHours ( -- )
    CLOCK_hours setIntensity drop drop drop
    60 0 do
        CLOCK_hours drop i nLED!
    5 +loop
  ;
```

The **setInitMinutes command** will color all the LEDs in the shade defined by **CLOCK_minutes**. Similarly, the setInitHours command will color only one of the five LEDs—that is, the LEDs indicating the hours—in the shade defined by CLOCK_hours. **setInitMinutes** is executed before **setInitHours**.

Needle placement

Once the background of the clock is defined, we look at how to place the "hands" of our digital clock with analog display.

The first word **calc-led-hh** will position the hour hand on one of the 60 available positions in our clock;

```
\ 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
  ;
```

We now define the color of each needle:

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

```
RGB_red    20 CLOCKcolor: CLOCK_HH
```

The seconds hand will be cyan, the minutes green, and the hours red, all three with a luminous intensity of 20%.

The following three words position each hand on our clock:

```
: 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!
  ;
```

These three actions are grouped in **`setHMS`** :

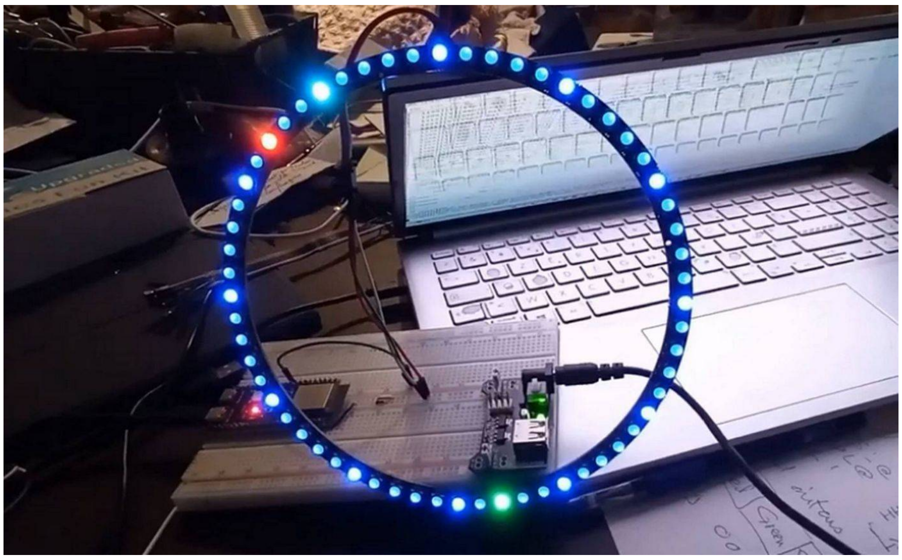
```
: setHMS ( -- )
    RTC.get-time
    setSS
    dup setMM
    setHH
  ;
```

Finally, we display the clock. The display is placed within a ` **`begin..again`` loop`** :

```
: displayClock ( -- )
    begin
        resetLEDs
        setInitMinutes
        setInitHours
        setHMS
        transmitLEDS
        1000 ms
    again
  ;
```

As it stands, we can start our clock:

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



But if you do this, the ESP32 board will no longer be available....

Let's add some multitasking to the clock.

This is where ESP32forth reveals its full power. FORTH is a multitasking language. A multitasking language can run several programs simultaneously. Here's how to do it, and it's incredibly simple:

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

We do not count the first line of code, as it simply initializes our clock.

The second line contains the execution code for **displayClock**, followed by a space for the two stacks of the task defined by **task** . The word task is followed by the task to be defined, here **my-clock**.

The last line launches the **my-clock task**.

If all goes well, the clock will display as in the picture above, but in addition, **you will still have the hand** of the FORTH interpreter!

Object 1