**Gemini**

# Integrating the ESP32-C6 Wi-Fi 6 Coprocessor with an ESP32-P4 Host: A Technical Implementation Guide

## 1. Architectural Foundation: The esp-hosted and esp\_wifi\_remote Framework

### 1.1 Introduction to the Hosted Architecture

The integration of Wi-Fi 6 capabilities into the ESP32-P4 platform, which lacks native wireless hardware, is accomplished through a sophisticated dual-chip architecture. In this model, the ESP32-P4 functions as the primary Host Microcontroller Unit (MCU), executing the main application logic, while an ESP32-C6 serves as a dedicated communication coprocessor, or "slave".1 This paradigm allows developers to leverage the high-performance RISC-V cores and rich peripheral set of the ESP32-P4 for demanding tasks like graphics rendering and signal processing, while offloading all wireless networking responsibilities to the power-efficient ESP32-C6.2

The communication between the host and the slave is governed by a Remote Procedure Call (RPC) mechanism.4 From the perspective of the application running on the ESP32-P4, interactions with the Wi-Fi stack appear as standard, local function calls. However, these calls are intercepted by a specialized software layer, serialized into a structured format using Protocol Buffers (protobuf), and transmitted over a physical transport medium—in this case, the SDIO interface—to the ESP32-C6 slave.1 The slave device receives this serialized data, deserializes it to reconstruct the original command, executes the corresponding Wi-Fi function on its native hardware, and transmits the result back to the host via the same transport layer. This architecture provides exceptional modularity, enabling the use of any Wi-Fi-capable ESP chipset as a coprocessor and allowing for the extension of the RPC library to expose any required functionality from the slave to the host.1

### 1.2 The Role of esp\_wifi\_remote

The esp\_wifi\_remote component is the cornerstone of this architecture from the application developer's standpoint. It functions as a "shim" or wrapper layer that provides an API identical to the standard esp\_wifi library found in ESP-IDF.6 When an application on the ESP32-P4 invokes a function such as

esp\_wifi\_init() or esp\_wifi\_connect(), it is not calling the standard implementation but rather the one provided by esp\_wifi\_remote.4 The sole responsibility of this component is to translate these familiar API calls into the necessary RPC requests and forward them to the

esp\_hosted component for transport.8

The elegance of this design lies in its transparency. It allows existing ESP-IDF applications, which are written against the standard esp\_wifi API, to be ported to a non-Wi-Fi-native host like the ESP32-P4 with minimal or no modification to the core application logic.4 For example, standard networking examples such as

iperf can be compiled and executed on the ESP32-P4, seamlessly utilizing the ESP32-C6 for the actual network connection.4 This component is automatically generated based on a specific version of the

esp\_wifi component within ESP-IDF, ensuring that the function prototypes and data structures remain perfectly synchronized.6

### 1.3 The Role of esp\_hosted

While esp\_wifi\_remote manages the API translation, the esp\_hosted component serves as the underlying transport and RPC engine.8 It is responsible for the low-level mechanics of the dual-chip communication. This includes managing the physical SDIO interface, handling the protobuf serialization and deserialization of commands and responses, and orchestrating the reliable flow of both control messages and bulk network data packets.1 The

esp\_hosted component effectively abstracts the physical connection, presenting a clean RPC channel that esp\_wifi\_remote can utilize. This component must be present and configured on both the host (ESP32-P4) and the slave (ESP32-C6) to establish a functional communication link.

### 1.4 Data and Command Flow

To fully grasp the interaction, consider the round-trip of a typical Wi-Fi API call within this architecture:

1. **Application Call:** The HowdyScreen application on the ESP32-P4 calls esp\_wifi\_connect().
2. **API Interception:** The esp\_wifi\_remote component, linked into the application, receives this call instead of the standard Wi-Fi library.
3. **RPC Serialization:** The esp\_wifi\_remote component formulates an RPC request representing the esp\_wifi\_connect() call and passes it to the esp\_hosted component.
4. **Transport Serialization:** The esp\_hosted component on the host serializes this RPC request into a protobuf message.
5. **Physical Transmission:** The host's SDIO driver, managed by esp\_hosted, transmits the protobuf message packet-by-packet over the physical SDIO bus to the ESP32-C6.
6. **Physical Reception:** The ESP32-C6 slave's SDIO peripheral receives the data.
7. **Transport Deserialization:** The esp\_hosted firmware running on the slave receives the raw data from its SDIO driver and deserializes the protobuf message, reconstructing the original RPC request.
8. **Execution:** The slave firmware interprets the request and executes the actual, native esp\_wifi\_connect() function using its own Wi-Fi radio and stack.
9. **Response Path:** The result of the function call (e.g., success or failure code) is packaged into a response message and sent back to the ESP32-P4 host by following the exact reverse path.

This entire process is transparent to the host application, which simply awaits the return value from its initial function call. This intricate dependency chain underscores the importance of version compatibility. The esp\_wifi\_remote component is tightly coupled to the ESP-IDF version it was generated from.6 While the initial query specified

esp\_wifi\_remote v0.3.0, analysis of component changelogs and dependencies reveals that newer versions offer explicit support for ESP-IDF v5.4 10 and have defined dependencies on specific

esp\_hosted versions.11 Therefore, for a project using ESP-IDF v5.4, it is strongly recommended to use the latest compatible component versions, such as

esp\_wifi\_remote v0.5.4 or newer and esp\_hosted v0.0.9 or newer, to ensure stability and access to bug fixes.

Furthermore, the dual-chip nature of the system necessitates a corresponding dual-logging approach for debugging. An error reported on the host, such as the common E (226409) transport: Not able to connect with ESP-Hosted slave device 12, could stem from a hardware fault in the SDIO lines, a misconfiguration in the host's

menuconfig, a crash or hang in the slave's firmware, or a version mismatch between the components. Effective troubleshooting requires simultaneous monitoring of the serial console output from both the ESP32-P4 host and the ESP32-C6 slave.

## 2. SDIO Transport Layer: Hardware and Software Configuration

### 2.1 Hardware Connections (P4 Host to C6 Slave)

A correct and stable physical connection is the foundation of the dual-chip system. The Waveshare ESP32-P4-NANO development board simplifies this by integrating an ESP32-C6-MINI-1 module directly on the PCB, connecting it to the ESP32-P4 via the SDIO interface.3 The GPIO pins used on the ESP32-C6 for its SDIO slave interface are fixed by the hardware design of the chip itself.8 The corresponding pins on the ESP32-P4 are determined by the board's layout.

In addition to the SDIO data and clock lines, a GPIO pin on the host must be connected to the reset (EN) pin of the slave. This allows the host's esp\_hosted driver to programmatically reset the coprocessor during the initialization sequence, ensuring it starts in a known-good state.8 Because the ESP32-C6 is integrated on the Waveshare board, the necessary 10 kOhm pull-up resistors required for stable SDIO communication are presumed to be included in the design.8

Table 2.1: ESP32-P4-NANO to ESP32-C6 SDIO Pin Mapping

The following table outlines the essential connections for the SDIO interface between the ESP32-P4 host and the ESP32-C6 slave on the Waveshare ESP32-P4-NANO board. These pin numbers must be accurately reflected in the host's software configuration.

| Signal | ESP32-C6 Pin (Slave) | ESP32-P4 Pin (Host) | Function |
| --- | --- | --- | --- |
| CLK | GPIO19 | GPIO36 | SDIO Clock |
| CMD | GPIO18 | GPIO37 | SDIO Command |
| DATA0 | GPIO20 | GPIO35 | SDIO Data Line 0 |
| DATA1 | GPIO21 | GPIO34 | SDIO Data Line 1 |
| DATA2 | GPIO22 | GPIO33 | SDIO Data Line 2 |
| DATA3 | GPIO23 | GPIO48 | SDIO Data Line 3 |
| RESET | EN (Reset) | GPIO47 | Host-controlled reset for the slave |

*Note: The ESP32-P4 pin assignments are specific to the Waveshare ESP32-P4-NANO and must be verified against the board's official schematic if issues arise. The ESP32-C6 pins are standard for its SDIO slave peripheral.*

### 2.2 Slave (ESP32-C6) Firmware Setup

The ESP32-C6 coprocessor does not run the user's application. Instead, it must be flashed with a dedicated slave firmware provided by Espressif. This firmware contains the esp\_hosted slave-side implementation, the native Wi-Fi stack, and the logic required to listen for and respond to RPCs from the host.

The setup process is as follows:

1. **Clone Repository:** Obtain the esp-hosted-mcu source code from the official Espressif repository on GitHub.1
2. **Navigate to Slave Example:** Change the directory to the provided slave application. This is typically located at esp-hosted-mcu/examples/slave/sdio/.
3. **Set Target:** Configure the build system for the ESP32-C6 target by executing idf.py set-target esp32c6.
4. **Configure Firmware:** Launch the configuration menu with idf.py menuconfig.
   * Navigate to Component config ---> ESP-Hosted config.
   * Verify that Transport Layer is set to SDIO.
   * Under Example Configuration, ensure the settings are appropriate. For most use cases, the defaults are sufficient.
5. **Build and Flash:** Compile the project using idf.py build. Flash the resulting binary to the ESP32-C6 module. The Waveshare board provides a dedicated 4-pin SH1.0 connector for flashing the C6 module, separate from the main USB-C port used for the P4.15 Connect a USB-to-serial adapter to this header to perform the flash operation.

### 2.3 Host (ESP32-P4) Project Setup

This phase involves integrating the necessary components into the main HowdyScreen application project running on the ESP32-P4.

1. **Manage Dependencies:** The most robust method for adding components in modern ESP-IDF is via the component manager. In the root directory of the HowdyScreen project, create or edit the idf\_component.yml file to declare the dependencies.16
2. **Add Components:** Execute the following commands from the project root to add the latest compatible versions of the required components. The component manager will automatically fetch them from the ESP Component Registry and handle linking.
   * idf.py add-dependency "espressif/esp\_wifi\_remote^0.5.4" 6
   * idf.py add-dependency "espressif/esp\_hosted^0.0.9" 8
3. **Set Target:** Configure the build system for the ESP32-P4 host: idf.py set-target esp32p4.
4. **Configure Host:** Launch the configuration menu with idf.py menuconfig. This step is critical and must match the hardware pinout precisely.
   * Navigate to Component config ---> ESP-Hosted config.
   * Set Transport Layer to SDIO.
   * Enter the SDIO Configuration submenu.
   * Set SDIO Host CLK GPIO to 36.
   * Set SDIO Host CMD GPIO to 37.
   * Set SDIO Host D0 GPIO to 35.
   * Set SDIO Host D1 GPIO to 34.
   * Set SDIO Host D2 GPIO to 33.
   * Set SDIO Host D3 GPIO to 48.
   * Set Slave Reset GPIO to 47.
   * Navigate to Component config ---> WiFi-remote config.
   * Set the Slave Target Name to esp32c6. This configuration is essential as it informs the esp\_wifi\_remote component's auto-generated code which slave-specific types and capabilities to use.6

### 2.4 Host Initialization Code Pattern

With the project correctly configured, the application code on the ESP32-P4 for initializing the Wi-Fi system is remarkably simple. The esp\_wifi\_remote component abstracts the complexity of the dual-chip setup. The standard Wi-Fi initialization sequence is replaced with a call to esp\_wifi\_remote\_init().

The following C code snippet demonstrates the correct initialization pattern within the app\_main function on the ESP32-P4:

C

#**include** "esp\_event.h"  
#**include** "esp\_log.h"  
#**include** "esp\_wifi\_remote.h"  
#**include** "nvs\_flash.h"  
  
static const char \*TAG = "HOST\_APP";  
  
// Your existing Wi-Fi event handler function  
static void wifi\_event\_handler(void\* arg, esp\_event\_base\_t event\_base,  
 int32\_t event\_id, void\* event\_data)  
{  
 if (event\_id == WIFI\_EVENT\_STA\_START) {  
 ESP\_LOGI(TAG, "WIFI\_EVENT\_STA\_START: connecting...");  
 esp\_wifi\_connect();  
 } else if (event\_id == WIFI\_EVENT\_STA\_DISCONNECTED) {  
 ESP\_LOGI(TAG, "WIFI\_EVENT\_STA\_DISCONNECTED: retrying connection...");  
 esp\_wifi\_connect();  
 } //... other events like GOT\_IP  
}  
  
void app\_main(void)  
{  
 // 1. Initialize NVS - a prerequisite for Wi-Fi operations  
 esp\_err\_t ret = nvs\_flash\_init();  
 if (ret == ESP\_ERR\_NVS\_NO\_FREE\_PAGES |  
  
| ret == ESP\_ERR\_NVS\_NEW\_VERSION\_FOUND) {  
 ESP\_ERROR\_CHECK(nvs\_flash\_erase());  
 ret = nvs\_flash\_init();  
 }  
 ESP\_ERROR\_CHECK(ret);  
  
 // 2. Initialize the underlying TCP/IP stack  
 ESP\_ERROR\_CHECK(esp\_netif\_init());  
  
 // 3. Create the default event loop  
 ESP\_ERROR\_CHECK(esp\_event\_loop\_create\_default());  
  
 // 4. Create default Wi-Fi station  
 esp\_netif\_create\_default\_wifi\_sta();  
  
 // 5. Initialize the remote Wi-Fi system.  
 // This single call replaces esp\_wifi\_init() and handles the setup of  
 // esp\_hosted, the SDIO transport, and communication with the C6 slave  
 // based on the settings from menuconfig.  
 ESP\_ERROR\_CHECK(esp\_wifi\_remote\_init());  
  
 // 6. Register event handlers for Wi-Fi events.  
 // This works exactly as it would with a native Wi-Fi chip.  
 ESP\_ERROR\_CHECK(esp\_event\_handler\_instance\_register(WIFI\_EVENT,  
 ESP\_EVENT\_ANY\_ID,  
 &wifi\_event\_handler,  
 NULL,  
 NULL));  
 //... register IP\_EVENT handlers as well  
  
 // 7. Configure and start Wi-Fi.  
 // All subsequent esp\_wifi\_\* calls are transparently redirected to the C6.  
 wifi\_config\_t wifi\_config = {  
 .sta = {  
 .ssid = "YOUR\_SSID",  
 .password = "YOUR\_PASSWORD",  
 },  
 };  
 ESP\_ERROR\_CHECK(esp\_wifi\_set\_mode(WIFI\_MODE\_STA));  
 ESP\_ERROR\_CHECK(esp\_wifi\_set\_config(WIFI\_IF\_STA, &wifi\_config));  
 ESP\_ERROR\_CHECK(esp\_wifi\_start());  
  
 ESP\_LOGI(TAG, "Wi-Fi initialization complete. Waiting for events.");  
}

## 3. Build System Integration and Conflict Resolution

### 3.1 The Core Problem: Symbol Collisions

A primary challenge when integrating a remote Wi-Fi solution is the potential for symbol collisions during the linking stage of the build process. Even though the ESP32-P4 lacks Wi-Fi hardware, the standard ESP-IDF build system may still attempt to link native Wi-Fi libraries (e.g., libnet80211.a, libwpa\_supplicant.a) if the project is not configured correctly. The esp\_wifi\_remote component is designed to *replace* these libraries with its own implementations that perform RPC calls. If both the native libraries and the remote component's stubs are included in the build, the linker will encounter duplicate symbol definitions for functions like esp\_wifi\_init, leading to build failure.

### 3.2 Kconfig: The Definitive Solution

The ESP-IDF build system's Kconfig infrastructure is the primary and definitive tool for resolving these conflicts. The build process for key components like esp\_wifi and wpa\_supplicant is conditional, based on specific Kconfig flags.

The central insight is that the host project must be explicitly configured to disable its own native Wi-Fi support, thereby creating a "void" that the esp\_wifi\_remote component can fill. The build logic within the esp\_wifi component's CMakeLists.txt file clearly illustrates this: it checks for if(CONFIG\_ESP\_WIFI\_ENABLED OR CONFIG\_ESP\_HOST\_WIFI\_ENABLED).17 If this condition is true, the full native Wi-Fi stack is compiled and linked. If it is false, the component only provides the public header files, allowing another component—

esp\_wifi\_remote—to provide the actual function implementations. The wpa\_supplicant component contains similar conditional logic.18

Furthermore, a critical and non-obvious detail is found in the esp\_wifi\_remote changelog, which notes a fix: "Disable WiFi remote impl if ESP\_HOST\_WIFI\_ENABLED=y".10 This reveals an inverse dependency: enabling the host's native Wi-Fi support actively

*disables* the remote Wi-Fi implementation. Therefore, to use the remote solution, disabling the host's native support is mandatory. A forum post corroborates this, explicitly advising that Component config → Wi-Fi → Host WiFi Enable must be disabled for the P4 hosted solution to work.9

Table 3.1: Kconfig WiFi Configuration for Host (ESP32-P4)

To ensure a successful build without symbol conflicts, the following settings must be configured in the ESP32-P4 host project via idf.py menuconfig.

| Menu Path | Setting | Value | Rationale |
| --- | --- | --- | --- |
| Component config -> Wi-Fi | WiFi | N (Disabled) | This is the master switch. Disabling it sets CONFIG\_ESP\_WIFI\_ENABLED to n. |
| Component config -> Wi-Fi | Host WiFi Enable | N (Disabled) | This sets CONFIG\_ESP\_HOST\_WIFI\_ENABLED to n, preventing the native stack from being built and avoiding the conflict that disables esp\_wifi\_remote. |
| Component config -> ESP-Hosted config | Enable ESP-Hosted | Y (Enabled) | Enables the underlying transport and RPC engine. |
| Component config -> WiFi-remote config | Enable WiFi-Remote | Y (Enabled) | Enables the API wrapper that provides the esp\_wifi functions. |
| Component config -> WiFi-remote config | Slave Target Name | esp32c6 | Informs the build system to use types and configurations specific to the ESP32-C6 slave. |

### 3.3 CMakeLists.txt and Dependency Management

With the adoption of the ESP-IDF Component Manager, direct manipulation of CMakeLists.txt files for dependency management is largely unnecessary and discouraged.16 The

idf\_component.yml file should be treated as the single source of truth for component dependencies.

When idf.py build is executed, the component manager resolves the dependencies listed in idf\_component.yml, downloads the components, and makes them available to the CMake build system. The Kconfig settings detailed above will then correctly control which parts of these components are compiled and linked.

It is important to ensure that the project's main CMakeLists.txt file does not contain legacy REQUIRES or PRIV\_REQUIRES directives that manually list system components like esp\_wifi or wpa\_supplicant. Such directives could override the conditional logic governed by Kconfig and reintroduce build conflicts. The project's CMakeLists.txt should remain minimal, focusing on defining the project name and source files, and allowing the build system to manage component relationships automatically.

## 4. Advanced LVGL Integration: Enabling GIF Animation

### 4.1 Enabling the LVGL GIF Decoder

The LVGL graphics library provides a built-in decoder for displaying animated GIF images, a feature directly relevant to the HowdyScreen application. Enabling this functionality for LVGL v8.3.x is a two-part process involving configuration and awareness of memory constraints.20

First, the GIF module must be enabled within the LVGL configuration. This is done by setting #define LV\_USE\_GIF 1 in the project's lv\_conf.h file. If the project uses the esp\_lvgl\_port component, this option can be set conveniently via idf.py menuconfig under Component config -> LVGL configuration -> 3rd Party Libraries.21

A crucial consideration is the memory requirement. The LVGL GIF decoder requires a significant amount of RAM to function, which is used as a buffer for the decoded frame data. The required memory scales directly with the image dimensions and the configured color depth 20:

* **LV\_COLOR\_DEPTH 16 (RGB565):** 4×width×height bytes
* **LV\_COLOR\_DEPTH 32 (ARGB8888):** 5×width×height bytes

For a moderately sized GIF (e.g., 240x240 pixels) at 16-bit color depth, this translates to 4×240×240=230,400 bytes, or 225 KB. This amount of memory is substantial and will almost certainly exceed the available internal SRAM of the ESP32-P4. Consequently, it is imperative that the project has PSRAM enabled and that this large buffer is allocated from the external 32MB PSRAM available on the ESP32-P4-NANO board.3 Failure to account for this will lead to allocation failures and application crashes.

### 4.2 Build System Configuration (CMake)

When using LVGL as a managed component via idf\_component.yml, the component's internal CMakeLists.txt file is designed to handle the conditional compilation of its features. When LV\_USE\_GIF is set to 1 in the configuration, the build system will automatically include the necessary source file (src/libs/gif/lv\_gif.c) in the compilation process. No manual changes to the project's CMakeLists.txt are required to enable the feature itself, provided the component is integrated correctly.

### 4.3 Code Implementation

Once enabled, using the GIF widget in application code is straightforward. LVGL provides the lv\_gif\_create() function to instantiate a GIF object and lv\_gif\_set\_src() to assign the image data. The source can be either a GIF file converted to a C array and embedded in the firmware, or a path to a file on a mounted filesystem.20

The ability to load a GIF from a file system creates a direct link to the project's partition table design. To use this method, the application must have a storage partition (e.g., SPIFFS or FAT) defined in its partitions.csv file, and the filesystem must be initialized and mounted in the application code before attempting to load the GIF.

The following code provides a complete example demonstrating both methods:

C

#**include** "lvgl.h"  
  
// Assume a filesystem is mounted at drive letter 'S'  
// and a GIF file has been converted to a C array using LVGL's tools.  
LV\_IMG\_DECLARE(my\_embedded\_gif); // Declared in a header generated by the converter  
  
void create\_gif\_animation\_example(lv\_obj\_t \* parent\_screen)  
{  
 // --- Method 1: Displaying a GIF from an embedded C array ---  
 lv\_obj\_t \* gif\_from\_array = lv\_gif\_create(parent\_screen);  
 lv\_gif\_set\_src(gif\_from\_array, &my\_embedded\_gif);  
 lv\_obj\_align(gif\_from\_array, LV\_ALIGN\_CENTER, -80, 0);  
 lv\_gif\_restart(gif\_from\_array); // Start the animation  
  
 // --- Method 2: Displaying a GIF from a file on a filesystem ---  
 // This requires a filesystem (e.g., SPIFFS, FAT) to be mounted.  
 // The path "S:/animations/loading.gif" assumes the filesystem is  
 // registered with the drive letter 'S'.  
 lv\_obj\_t \* gif\_from\_file = lv\_gif\_create(parent\_screen);  
 lv\_gif\_set\_src(gif\_from\_file, "S:/animations/loading.gif");  
 lv\_obj\_align(gif\_from\_file, LV\_ALIGN\_CENTER, 80, 0);  
 lv\_gif\_restart(gif\_from\_file); // Start the animation  
}

## 5. System-Level Configuration for Production-Ready Applications

### 5.1 Partitioning for Large Applications with OTA

A robust partition table is essential for any non-trivial embedded application, especially one that requires Over-The-Air (OTA) update capabilities and storage for large assets. The user's HowdyScreen application, being larger than 1MB, cannot use the default single-app partition schemes and requires a custom layout.23

An OTA-enabled system in ESP-IDF requires a specific structure: at least two app type partitions of identical size (conventionally ota\_0 and ota\_1) and a small data partition with the subtype ota (named otadata).24 The

otadata partition, typically 8 KB (0x2000 bytes) in size, acts as a ledger, storing a pointer that tells the bootloader which of the two app partitions contains the valid, bootable firmware.24

Given that the ESP32-P4-NANO is equipped with 16MB of flash memory 3 and that a complex LVGL application with networking can easily exceed 2-3MB (as seen in a similar project which uses a 3MB app partition 25), a generous allocation for the application partitions is prudent. The following custom partition table provides 3MB for each OTA slot, leaving ample space for a large FAT filesystem to store assets like images, fonts, and GIF files.

To use this table, the content should be saved to a file (e.g., partitions\_custom.csv) in the project's root directory. Then, in idf.py menuconfig, navigate to Partition Table and select Custom partition table CSV, providing the name of the created file.

**Table 5.1: Custom Large Application OTA Partition Scheme (partitions\_custom.csv)**

Code snippet

# ESP-IDF Partition Table for Large Application with OTA and FATFS  
# Name, Type, SubType, Offset, Size, Flags  
nvs, data, nvs, , 24K,  
otadata, data, ota, , 8K,  
phy\_init, data, phy, , 4K,  
ota\_0, app, ota\_0, , 3M,  
ota\_1, app, ota\_1, , 3M,  
storage, data, fat, , , # The blank size field tells the tool to use all remaining flash space

### 5.2 Configuring 2.4GHz-Only Wi-Fi Operation

The request to configure the ESP32-C6 for 2.4GHz-only operation stems from a common point of confusion in the hosted architecture. The ESP32-C6 is, by design, a single-band Wi-Fi 6 chip that operates exclusively in the 2.4 GHz spectrum.26 It does not contain the necessary radio hardware to transmit or receive on 5 GHz or 6 GHz frequencies.

However, the esp\_wifi\_remote and esp\_hosted components on the host side are built upon the generic ESP-IDF esp\_wifi component, which is designed to support dual-band chips as well. Because of this, the host's menuconfig may present options related to band selection or 5 GHz operation. These options are artifacts of the generic framework and have no effect on the ESP32-C6 slave.

Therefore, no specific software configuration is required on the host or the slave to enforce 2.4GHz-only operation. The limitation is inherent to the ESP32-C6 hardware itself. While the ESP32-C6 firmware does expose an API like esp\_wifi\_set\_band() 28, calling it to select the 2.4 GHz band is redundant. The system will operate on the 2.4 GHz band by default and cannot be configured otherwise. This understanding prevents developers from pursuing unnecessary configuration changes on the host side.

## 6. Synthesis and Recommendations: A Complete Project Configuration

### 6.1 Host Project Configuration Checklist

To successfully build and run the HowdyScreen application on the ESP32-P4 with an ESP32-C6 coprocessor, the following configuration summary provides a final checklist for the host project.

* **ESP-IDF Version:** v5.4.x
* **Dependencies (idf\_component.yml):**
  + espressif/esp\_hosted: ^0.0.9
  + espressif/esp\_wifi\_remote: ^0.5.4
  + lvgl/lvgl: ^8.3.11
* **Partition Table (menuconfig):**
  + Partition Table -> Custom partition table CSV
  + Custom partition CSV file -> partitions\_custom.csv (using the layout from Table 5.1)
* **Kconfig Settings (menuconfig):**
  + Component config -> Wi-Fi -> WiFi: **Disabled**
  + Component config -> Wi-Fi -> Host WiFi Enable: **Disabled**
  + Component config -> ESP-Hosted config -> Enable ESP-Hosted: **Enabled**
  + Component config -> ESP-Hosted config -> Transport Layer: **SDIO**
  + Component config -> ESP-Hosted config -> SDIO Configuration: Match pins from Table 2.1.
  + Component config -> WiFi-remote config -> Enable WiFi-Remote: **Enabled**
  + Component config -> WiFi-remote config -> Slave Target Name: **esp32c6**
  + Component config -> LVGL configuration -> 3rd Party Libraries -> Enable GIF decoder: **Enabled**
  + Component config -> ESP PSRAM -> Support for external, SPI-connected RAM: **Enabled**

### 6.2 Sample idf\_component.yml for Host

This manifest file should be placed in the root of the ESP32-P4 project to ensure the component manager fetches the correct dependencies.

YAML

## IDF Component Manager Manifest File  
dependencies:  
 # Required for ESP32-P4 to use C6 as Wi-Fi coprocessor  
 espressif/esp\_hosted: "^0.0.9"  
 espressif/esp\_wifi\_remote: "^0.5.4"  
  
 # Required for LVGL v8.3  
 lvgl/lvgl: "^8.3.11"  
   
 # Optional but recommended for easy display/touch integration if not handled manually  
 # espressif/esp\_lvgl\_port: "..."  
  
 # Other application dependencies (e.g., for mDNS, UDP) would be listed here  
  
## Required IDF version  
idf:  
 version: ">=5.4.0"

### 6.3 Debugging Strategies for Dual-Chip Systems

Troubleshooting a dual-chip system requires a methodical approach and visibility into both processors.

* **Dual Serial Monitoring:** It is essential to have two separate USB-to-serial converters and terminal windows open simultaneously. One connected to the ESP32-P4's main UART port, and the other connected to the ESP32-C6's dedicated flashing/debug header.15 This allows for concurrent observation of logs from both the host and the slave.
* **Common Error Analysis:**
  + **Host Log:** E (....) transport: Not able to connect with ESP-Hosted slave device.12
    - **Likely Causes:**
      1. The ESP32-C6 slave firmware is not running, has crashed, or was not flashed correctly. Check the C6's serial log for activity.
      2. A hardware issue with the SDIO connections or the host-to-slave reset line.
      3. A version mismatch between the host and slave esp\_hosted components.
      4. Incorrect GPIO pin configuration for the SDIO interface or reset line in the host's menuconfig.
  + **Host Log:** E (....) dma\_utils: Not enough heap memory assert failed: transport\_drv\_ap\_tx.29
    - **Likely Causes:** This indicates buffer exhaustion in the esp-hosted transport layer, especially under high throughput (e.g., streaming). The default buffer sizes may be insufficient.
    - **Solution:** In both the host and slave menuconfig, navigate to Component config -> ESP-Hosted config and increase the size and number of transport buffers.
  + **Build/Linker Errors:** Any linker errors related to duplicate esp\_wifi or wpa\_supplicant symbols.
    - **Likely Cause:** The host's native Wi-Fi stack is being compiled.
    - **Solution:** Re-verify that WiFi and Host WiFi Enable are both disabled in menuconfig as per Table 3.1.

### 6.4 Final Recommendations

Successfully integrating this complex system is best achieved through an incremental process.

1. **Isolate and Validate the Hosted Setup:** Before integrating into the full HowdyScreen application, create a minimal test project. A highly effective approach is to adapt the standard ESP-IDF examples/wifi/iperf example.30 Follow the steps in this guide to add the  
   esp\_wifi\_remote and esp\_hosted dependencies and apply the necessary Kconfig changes to the iperf project.9 Successfully running  
   iperf between the ESP32-P4/C6 setup and a PC provides definitive validation that the entire hosted communication stack—from hardware to RPCs—is functioning correctly.
2. **Integrate into Main Application:** Once the hosted Wi-Fi is proven to work in isolation, merge the configurations (idf\_component.yml, partitions\_custom.csv, sdkconfig changes) into the main HowdyScreen project.
3. **Stay Informed:** The esp-hosted-mcu and esp-wifi-remote components are under active development. Periodically check the official GitHub repositories for new releases, bug fixes, and issue tracker discussions that may be relevant to the ESP32-P4/C6 combination.12 This ensures the project benefits from the latest performance improvements and stability enhancements from Espressif.

#### Works cited

1. espressif/esp-hosted-mcu - GitHub, accessed July 23, 2025, <https://github.com/espressif/esp-hosted-mcu>
2. ESP32-P4-Module-DEV-KIT - Waveshare Wiki, accessed July 23, 2025, <https://www.waveshare.com/wiki/ESP32-P4-Module-DEV-KIT-StartPage>
3. ESP32-P4-NANO - Waveshare Wiki, accessed July 23, 2025, <https://www.waveshare.com/wiki/ESP32-P4-Nano-StartPage>
4. espressif/esp\_hosted • v0.0.14 - ESP Component Registry, accessed July 23, 2025, <https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.14?language=en>
5. espressif/esp-hosted: Hosted Solution (Linux/MCU) with ESP32 (Wi-Fi + BT + BLE) - GitHub, accessed July 23, 2025, <https://github.com/espressif/esp-hosted>
6. espressif/esp\_wifi\_remote • v0.5.4 • ESP Component Registry, accessed July 23, 2025, <https://components.espressif.com/components/espressif/esp_wifi_remote/versions/0.5.4>
7. espressif/esp\_wifi\_remote • v0.14.3 - ESP Component Registry, accessed July 23, 2025, <https://components.espressif.com/components/espressif/esp_wifi_remote>
8. espressif/esp\_hosted • v0.0.9 - ESP Component Registry, accessed July 23, 2025, <https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.9?language=en>
9. Wifi over SDIO on P4 - ESP32 Forum, accessed July 23, 2025, <https://esp32.com/viewtopic.php?t=43750>
10. espressif/esp\_wifi\_remote - 0.4.1 Changelog - ESP Component Registry, accessed July 23, 2025, <https://components.espressif.com/components/espressif/esp_wifi_remote/versions/0.4.1/changelog?language=en>
11. espressif/esp\_wifi\_remote - 0.5.4 Dependencies - ESP Component Registry, accessed July 23, 2025, <https://components.espressif.com/components/espressif/esp_wifi_remote/versions/0.5.4/dependencies?language=en>
12. ESP32-P4 can not connect to ESP32-C6 on the Function\_EV\_Board v1.2 and v1.4 (EHM-2) · Issue #2 · espressif/esp-hosted-mcu - GitHub, accessed July 23, 2025, <https://github.com/espressif/esp-hosted/issues/511>
13. ESP32-P4-NANO High-performance Development Board, Based On ESP32-P4 Chip With RISC-V Dual-core And Single-core Processors, Supports Wi-Fi 6 And Bluetooth 5/BLE - Waveshare, accessed July 23, 2025, <https://www.waveshare.com/esp32-p4-nano.htm>
14. ESP32-C6 SDIO AT Guide - Espressif Systems, accessed July 23, 2025, <https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html>
15. ESP32-P4-WIFI6-Touch-LCD-3.4C User Guide - Spotpear, accessed July 23, 2025, <https://spotpear.com/wiki/ESP32-P4-3.4-inch-Round-LCD-Display-TouchScreen-WIFI6.html>
16. esp32p4 ble client with esp\_hosted - ESP32 Forum, accessed July 23, 2025, <https://esp32.com/viewtopic.php?t=44329>
17. esp-idf/components/esp\_wifi/CMakeLists.txt at master - GitHub, accessed July 23, 2025, <https://github.com/espressif/esp-idf/blob/master/components/esp_wifi/CMakeLists.txt>
18. esp-idf/components/wpa\_supplicant/CMakeLists.txt at master - GitHub, accessed July 23, 2025, <https://github.com/espressif/esp-idf/blob/master/components/wpa_supplicant/CMakeLists.txt>
19. Having idf\_component.yml issues when trying to create my own components., accessed July 23, 2025, <https://esp32.com/viewtopic.php?t=39495>
20. GIF decoder — LVGL documentation, accessed July 23, 2025, <https://docs.lvgl.io/8.3/libs/gif.html>
21. Espressif (ESP32 Chip Series) - LVGL 9.4 documentation, accessed July 23, 2025, <https://docs.lvgl.io/master/details/integration/chip/espressif.html>
22. GIF Decoder - LVGL 9.4 documentation, accessed July 23, 2025, <https://docs.lvgl.io/master/details/libs/gif.html>
23. Partition Tables - ESP32-P4 - — ESP-IDF Programming Guide v5 ..., accessed July 23, 2025, <https://docs.espressif.com/projects/esp-idf/en/stable/esp32p4/api-guides/partition-tables.html>
24. Over The Air Updates (OTA) - ESP32-P4 - — ESP-IDF Programming Guide v5.4.2 documentation - Espressif Systems, accessed July 23, 2025, <https://docs.espressif.com/projects/esp-idf/en/stable/esp32p4/api-reference/system/ota.html>
25. chvvkumar/ESP32-P4-Allsky-Display - GitHub, accessed July 23, 2025, <https://github.com/chvvkumar/ESP32-P4-Allsky-Display>
26. ESP32-C6 Series - Espressif Systems, accessed July 23, 2025, <https://www.espressif.com/sites/default/files/documentation/esp32-c6_datasheet_en.pdf>
27. ESP32-C6 Reality Check - romkey.com, accessed July 23, 2025, <https://romkey.com/2023/02/16/esp32c6-reality-check/>
28. Wi-Fi - ESP32-C6 - — ESP-IDF Programming Guide v5.4.2 ..., accessed July 23, 2025, <https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/network/esp_wifi.html>
29. ESP32-P4 EVK C6 - Not enough heap memory · Issue #597 ..., accessed July 23, 2025, <https://github.com/espressif/esp-hosted/issues/597>
30. esp-idf/examples/ethernet/iperf/README.md at master - GitHub, accessed July 23, 2025, <https://github.com/espressif/esp-idf/blob/master/examples/ethernet/iperf/README.md>

**03-Pro**

# ESP32-P4 + ESP32-C6 Dual-Chip Wi-Fi Integration

## 1. ESP32-P4 and ESP32-C6 SDIO Communication Setup

**Dual-Chip Wi-Fi Architecture:** The ESP32-P4 has no built-in wireless, so it can pair with an ESP32-C6 (which has 2.4 GHz Wi-Fi 6 and BLE) as a **Wi-Fi co-processor** over SDIO. Espressif provides the *ESP-Hosted MCU* solution for this scenario. On the P4 (host), you include two components: **esp\_wifi\_remote** and **esp\_hosted**. The esp\_wifi\_remote component exposes the standard ESP-IDF Wi-Fi API (e.g. esp\_wifi\_init, esp\_wifi\_start) as stubs, and esp\_hosted implements those calls by forwarding them via RPC to the C6 co-processor. In other words, when your P4 app calls an esp\_wifi function, it is packaged and sent over SDIO to the C6, which executes the real Wi-Fi radio operation and returns results back. This allows “seamless” use of the normal Wi-Fi API on ESP32-P4, despite the P4 having no radio.

**SDIO Transport Configuration:** The ESP32-P4 and C6 must be connected by the SDIO host interface (P4 acting as SDIO host, C6 as SDIO slave). In Espressif’s Function EV board and the Waveshare ESP32-P4-Nano, the SDIO wiring is predefined (e.g. CLK on GPIO18, CMD on GPIO19, data lines on GPIO14–17, plus a reset line to the C6). By default, the ESP-Hosted software uses a 4-bit SDIO mode at up to 40 MHz for communication. Typically, you do **not** need to manually configure the pins or SDIO mode if you are using a known board – the esp\_hosted component’s Kconfig has defaults for the P4 board. Just ensure the SDIO lines are wired correctly if using a custom setup.

**Example Project – Wi-Fi Station/Iperf:** Espressif’s documentation and community examples show how to set this up. For instance, one user converted the standard Wi-Fi iperf example to use the ESP32-C6 coprocessor. The steps were: add the esp\_wifi\_remote and esp\_hosted dependencies, remove any existing Wi-Fi driver component, set target to esp32p4, and then configure via menuconfig. After that, the P4 could scan, connect, and run iperf through the C6 over SDIO. Key points from that example:

* **Add components:** idf.py add-dependency "espressif/esp\_wifi\_remote" and "espressif/esp\_hosted" to your project (this fetches the required components). This will also pull in protobuf-c as needed for RPC.
* **Remove conflicts:** If the project had a component like ext-conn (external connectivity helper) that brings in the normal Wi-Fi, remove or disable it. In short, your app should *only* use the remote Wi-Fi stack, not the native one.
* **SDIO pins and reset:** Ensure the SDIO pin assignment matches your hardware. In the ESP32-P4 boards, the default assignments in esp\_hosted were correct by default. Also set the reset pin for the C6 if required (the EV board used a dedicated reset GPIO).
* **Menuconfig settings:** In menuconfig, go to **Component config → Wi-Fi Remote → Wi-Fi configuration**. Verify that \*\*“Host WiFi Enable” is **disabled** (it should be off by default on ESP32-P4). This prevents the P4 from trying to use a nonexistent internal Wi-Fi. In the same menu, set **CONFIG\_SLAVE\_IDF\_TARGET to ESP32-C6** (so the host knows the slave is an ESP32-C6). You should see options like “Max number of WiFi static RX buffers” etc., which correspond to the C6’s Wi-Fi RX/TX buffer tuning – these are exposed by esp\_wifi\_remote. Typically, you can use the defaults for these buffer counts unless you have specific throughput requirements.
* **Initialization code:** With the above in place, you initialize Wi-Fi on the P4 almost exactly as you would on a regular ESP32. For example, call esp\_wifi\_init(), esp\_wifi\_set\_mode(), esp\_wifi\_start() and so on (or use the higher-level esp\_netif APIs for STA/AP). The esp\_wifi\_remote layer will intercept these and perform RPC to the C6. **No special dual-CPU init code is needed in your app** besides ensuring the components are in place and configured. The ESP-Hosted runtime takes care of handshaking with the C6 at startup (resetting the C6, starting the SDIO transport tasks, etc.). For instance, after flashing, you’ll see logs on the P4 like “Starting SDIO… Received INIT event from slave” and then Wi-Fi events as usual.
* **Example outcome:** After following these steps, the P4 was able to successfully bring up Wi-Fi. In the iperf test case, the console showed the SDIO link coming up (4-bit mode at 40MHz) and then the normal Wi-Fi CLI (help, sta, iperf commands, etc.) became available, indicating the P4 was communicating with the C6 for Wi-Fi. Similarly, Waveshare’s Wi-Fi Station demo shows the P4 connecting to an AP through the C6 and printing an IP address as if it had native Wi-Fi.

In summary, to integrate ESP32-P4 with an ESP32-C6 over SDIO: **include Espressif’s esp\_wifi\_remote and esp\_hosted components, disable any native Wi-Fi, configure the transport pins (if needed), and use the normal Wi-Fi API.** Espressif’s example on the Function EV board and Waveshare’s demos confirm this setup works – for instance, running an iperf throughput test or a simple Wi-Fi STA connection is possible with minimal changes to your application code.

*(Sources: Espressif ESP-Hosted documentation and component README, Waveshare ESP32-P4-Nano Wiki, and ESP32 forum example of P4 SDIO Wi-Fi setup.)*

## 2. LVGL GIF Animation Support (LVGL v8.3.x)

**LVGL GIF Decoder:** LVGL 8.3 has built-in support for GIF image decoding, allowing you to display animated GIFs on the ESP32. To use this feature, you need to **enable the GIF decoder in LVGL’s configuration**. In lv\_conf.h or via menuconfig, set LV\_USE\_GIF to 1 (enabled). (If using ESP-IDF’s menuconfig, this corresponds to the option **“GIF decoder library”** under LVGL component settings, i.e. CONFIG\_LV\_USE\_GIF=y.) Without this flag, the lv\_gif functions will be compiled out and calling them will have no effect.

Once GIF support is enabled, LVGL provides the **lv\_gif\_create(parent)** widget and **lv\_gif\_set\_src(obj, src)** API to show animations. This works analogously to regular images (lv\_img): you can either use a GIF from a file or embed the GIF data as a C array. For example, to use an embedded GIF, convert the .gif file into a C array (LVGL has an online converter for this). Then you can do:

lv\_obj\_t \*img = lv\_gif\_create(lv\_scr\_act()); // create GIF image object

lv\_gif\_set\_src(img, &my\_gif\_img\_dsc); // set source to the image descriptor (C array)

If instead you want to load from the filesystem (e.g. SPIFFS, SD card), you can attach a file path:

lv\_obj\_t \*img = lv\_gif\_create(lv\_scr\_act());

lv\_gif\_set\_src(img, "S:/path/to/anim.gif"); // 'S:' is the drive letter in LVGL for SD card or SPIFFS

LVGL will read and decode the GIF frame by frame. **Important:** To use files, you must have an LVGL file system driver enabled and mounted. For example, enabling LV\_USE\_FS\_STDIO (or the appropriate driver for FATFS or SPIFFS) and calling lv\_fs\_stdio\_init() to register it. The LVGL docs note that “a file system driver needs to be registered to open images from files” – otherwise lv\_gif\_set\_src with a file path will not work. In practice, on ESP32 you might use SPIFFS (and use path "S:myfile.gif" if you register SPIFFS with letter S) or littlefs, etc., or use the POSIX stdio driver to read from files if you have an underlying file I/O.

**CMake and Component Config:** If you are using LVGL via ESP-IDF’s component or component manager, enabling the GIF decoder is typically done through Kconfig (as above). The LVGL component’s Kconfig will ensure the GIF decoder source files are compiled in. You usually don’t need any special CMakeLists changes beyond having the LVGL component included. Just double-check that your sdkconfig has CONFIG\_LV\_USE\_GIF=y. If you manage LVGL manually, make sure lv\_conf.h has #define LV\_USE\_GIF 1 **before** building LVGL. (LVGL’s build system will include the GIF decoder from lv\_lib\_gif internally when that flag is on.)

**Example Usage:** Users have successfully displayed GIFs on ESP32 using LVGL 8.x. For instance, one example displays an animated bulb GIF either from an array or from SPIFFS. The code calls lv\_gif\_create and then lv\_gif\_set\_src(img, &img\_bulb\_gif) for the embedded case, or lv\_gif\_set\_src(img, "/spiffs/bulb.gif") after initializing SPIFFS. LVGL handles the decoding and timing of frames. After setting the source, the GIF will automatically play in a loop. You can stop it by deleting the object or by other LVGL controls (LVGL also provides lv\_gif\_restart() etc. if needed). Keep in mind that GIF decoding can be memory intensive: LVGL documentation notes that at 16-bit color depth, the RAM needed is roughly *4 × (width × height)* bytes for the GIF frame buffer. Ensure you have enough heap/PSRAM for large or high-frame-rate GIFs.

In summary, **to enable GIF animations in LVGL 8.3 on ESP-IDF:** turn on the GIF decoder option in LVGL’s config, include a filesystem driver if loading from files, and then use lv\_gif\_create and lv\_gif\_set\_src just like the standard image API. With this setup, many developers have shown smooth GIF playback on ESP32-based displays (e.g. small TFT screens). For example, a forum user was able to display alternating GIFs on an ESP32-S3 box by enabling LV\_USE\_GIF and handling the LVGL thread safety properly (semaphores around lv\_obj\_clean and lv\_gif\_set\_src calls) – proving that the GIF functionality works when configured correctly.

*(Sources: LVGL documentation, CSDN blog example of LVGL GIF on ESP32.)*

## 3. Wi-Fi Symbol Conflict Resolution (esp\_wifi\_remote vs. Native Wi-Fi)

When integrating esp\_wifi\_remote/esp\_hosted, a common issue is **symbol or configuration conflicts** with the regular Wi-Fi libraries in ESP-IDF. Since ESP-IDF’s Wi-Fi (esp\_wifi) is designed for chips with internal radios, we must ensure the P4 doesn’t try to compile or link any of those when we are using the remote Wi-Fi instead. Here are the key strategies:

* **Disable “Host WiFi” in Kconfig:** In menuconfig for ESP32-P4, under **Component config → Wi-Fi**, there is an option called “**Host WiFi Enable**” (CONFIG\_ESP\_HOST\_WIFI\_ENABLED). This should remain **No/disabled**. In fact, by default for ESP32-P4 this is off (since P4 has no internal Wi-Fi). If you were to enable it, the build system would try to include the normal WiFi driver and define WiFi config variables for an internal radio, which leads to problems. For example, one user encountered build errors like CONFIG\_ESP\_WIFI\_STATIC\_RX\_BUFFER\_NUM undeclared when this was not handled. The reason: those buffer configs are normally defined only when WiFi is enabled for a WiFi-capable SoC. The correct approach is to leave “Host WiFi” disabled – instead, the esp\_wifi\_remote component will present its own Wi-Fi config menu (tied to the slave) as described earlier. In short, **do not turn on any standard WiFi features on the P4**; let esp\_wifi\_remote take over that namespace.
* **Use esp\_wifi\_remote’s menu for WiFi configs:** After adding esp\_wifi\_remote, you should see **Component config → Wi-Fi Remote → Wi-Fi configuration** in menuconfig. Ensure you can enter that menu and configure the Wi-Fi settings for the C6 (if needed). Typically, this menu becomes available only when the environment knows the IDF version (sometimes you might need to set ESP\_IDF\_VERSION environment variable as noted by a user, to get the menu to appear for IDF 5.4). Inside, it lets you adjust things like RX/TX buffer numbers, AMPDU settings, etc., which correspond to the C6’s Wi-Fi driver. If those aren’t set, you’d get undefined reference errors as above. So, double-check that the “Wi-Fi Remote: Wi-Fi configuration” submenu is visible and populated. If not, make sure you have the correct ESP-IDF version and component versions that match (esp\_wifi\_remote 0.3.0 or later, esp-idf 5.4 in this case).
* **Avoid Duplicate Symbols:** Only one set of Wi-Fi implementations should be linked. The esp\_wifi\_remote component provides **weak** dummy implementations of the Wi-Fi APIs, which esp\_hosted then fulfills via RPC. You must ensure the real esp\_wifi library from ESP-IDF is *not* also being linked for the host, otherwise you’d have multiple definitions of functions like esp\_wifi\_init. In practice, if CONFIG\_ESP\_HOST\_WIFI\_ENABLED is off, the esp\_wifi library is not compiled for ESP32-P4, so you’re fine. However, some high-level example code or components might implicitly pull in the Wi-Fi library. The **“ext-conn”** component (external connectivity helper) in ESP-IDF examples is one such thing – it can include Wi-Fi station code. In the forum example, the user had to **remove ext-conn from the project’s idf\_component.yml** to prevent it from bringing in the normal Wi-Fi stack. So, audit your components: if you have something like wifi\_provisioning, esp\_wifi library, or ext\_conn in the build, remove or disable them for ESP32-P4. Only esp\_wifi\_remote (and its dependencies) should define the Wi-Fi functions on the host side.
* **Set the correct slave target:** As mentioned, set CONFIG\_SLAVE\_IDF\_TARGET="esp32c6" (or via menuconfig under Wi-Fi Remote options, “Slave WiFi Target → ESP32-C6”). This ensures that the esp\_wifi\_remote component knows which chip it’s communicating with and enables the correct features (e.g. Wi-Fi 6, channel bandwidth settings appropriate for C6). This also avoids mismatches – e.g., if the slave were ESP32-C3, the Wi-Fi capabilities differ. In Kconfig, CONFIG\_SLAVE\_IDF\_TARGET is used to include the right headers for the remote Wi-Fi (it essentially substitutes for CONFIG\_IDF\_TARGET in the wifi Kconfig). The example confirms this was set to ESP32-C6 for their board.

By following these measures, several projects have successfully compiled and run Wi-Fi on ESP32-P4 without symbol collisions. The Waveshare wiki explicitly instructs adding esp\_wifi\_remote and esp\_hosted and notes that after that, you can use esp\_wifi as normal – this implies that the above steps (disabling host WiFi etc.) are handled. Indeed, the default P4 sdkconfig has host WiFi off, and the component manager takes care of the rest. If you do everything correctly, your build should link cleanly. If you *do* see errors about duplicate symbols or missing CONFIG\_ESP\_WIFI\_\* macros, it’s a sign that either the native Wi-Fi component is still being pulled in or the esp\_wifi\_remote Kconfig didn’t run – revisit the points above.

**Summary:** *Use only the esp\_wifi\_remote/hosted components for Wi-Fi on P4, disable the native Wi-Fi, and configure via the provided Kconfig options.* This will resolve symbol collisions (like multiple definitions of Wi-Fi functions or Kconfig macros). Espressif’s official guidance and community experiences reinforce this approach: “Host WiFi Enable” should stay off for P4 and all Wi-Fi config should come from the remote component.

*(Sources: ESP32 forum discussion on Wi-Fi over SDIO, Waveshare Wiki instructions, and esp\_wifi\_remote component README.)*

## 4. Partition Table for Large Applications on ESP32-P4

The ESP32-P4, like other ESP32 chips, uses a flash partition table to define flash usage. By default (in ESP-IDF), the **“Single factory app, no OTA”** partition scheme is used, which allocates a **1 MB** slot for the application binary. If your application is larger than 1 MB (which is likely, given you have many features like LVGL, Wi-Fi, etc.), you will need a different partition scheme.

**Larger Factory App Partition:** One option is to select the built-in **“Single factory app (large), no OTA”** scheme. This expands the app partition to **1.5 MB** for a single application. This is still no OTA (only one app slot, plus other partitions like NVS, etc.), but gives your app more room than the standard 1MB. In menuconfig, you can choose this under **Partition Table → Factory app, no OTA (large)**. The IDF documentation notes this 1.5MB table is intended to “fit more code” while still not enabling OTA. The CSV file for it is partitions\_singleapp\_large.csv in the IDF components folder.

**OTA Partition Schemes:** If you need OTA capability (multiple slots), there are default options for that as well. The **“Factory app, two OTA”** scheme gives you 3 slots (factory + OTA\_0 + OTA\_1), each of **1 MB**, requiring at least 4MB flash. However, 1MB per slot may not be enough for your app. In that case, Espressif provides **“Two OTA, large”** (sometimes called “OTA with two large apps”). This scheme defines **two OTA app partitions, each ~1.7 MB (1700 KB)**, with no separate factory slot (it uses OTA slots only). It requires a 4MB or bigger flash (which the P4 boards have – e.g. Waveshare P4 has 16MB flash, so plenty of space). In menuconfig this is labeled as **“Two OTA partitions (large)”**. Each OTA slot is 0x1A0000 bytes (1.64MB) if we interpret 1700K as KiB, allowing your app up to that size. This is a good choice if you want OTA but your app is >1MB. The IDF documentation explicitly mentions this config for larger apps: *“two OTA app partitions… 1700K, requires ≥4MB flash”*.

**Custom Partition Table:** If the provided options don’t meet your needs (for example, if you want even bigger app partitions or a different combination of data partitions), you can define a custom partition table CSV. Espressif encourages copying a template and modifying it for production use. For instance, with a 16MB flash, you might create a table with two 4MB OTA slots and still have space for SPIFFS, etc. To do this, set **Partition Table → Custom partition table CSV** in menuconfig and provide the path to your CSV. Just be sure the offsets and sizes align with flash size and that you update the offset for the partition table itself if needed (usually it stays at 0x8000). Many developers have successfully made custom tables for large apps – the process is the same on ESP32-P4 as on other ESP32 variants.

In summary, **for >1MB apps on ESP32-P4**: you can use **single app (large)** for ~1.5MB single image, or **two OTA (large)** for ~1.7MB OTA slots. Both are available out-of-the-box in IDF 5.4. If those still aren’t enough, go with a custom partition table. Given the P4-Nano has 16MB flash, you have flexibility – for example, a common custom setup might allocate 2MB for app and 2MB for OTA (if you only need one OTA slot), or even larger. Just ensure the bootloader (0x1000) and partition table (0x8000) offsets remain standard unless you have a reason to move them.

*(Sources: ESP-IDF v5.4 Partition Table Kconfig reference and general ESP32 partitioning guidelines.)*

## 5. Configuring 2.4GHz-Only Wi-Fi (ESP32-C6 Band Limit)

The ESP32-C6 co-processor supports **only 2.4 GHz Wi-Fi** (802.11b/g/n/ax in the 2.4 GHz band). It does *not* have a 5 GHz radio. Therefore, you do not actually need to configure a band preference – the hardware is inherently 2.4 GHz only. However, it’s important to keep this in mind when provisioning Wi-Fi credentials or setting up networks:

* **Ensure the target AP is 2.4 GHz:** If you try to connect the ESP32-C6 to a 5 GHz-only SSID, it will never succeed because the C6 cannot see 5 GHz beacons. In a dual-band router scenario (same SSID for 2.4G and 5G), the C6 will connect on the 2.4G radio. The Waveshare documentation explicitly reminds users: *“Note that ESP32-C6 supports 2.4GHz WiFi-6, so when choosing your target WiFi, make sure the frequency is 2.4GHz.”*. This is a key point during provisioning.
* **No 5GHz mode to disable:** Some Wi-Fi chipsets have options to disable one band or the other. On ESP32-C6, since there is only one band, there’s no specific API call needed to restrict it – it’s always restricted to 2.4G. The normal Wi-Fi APIs (e.g. esp\_wifi\_set\_channel) on C6 only accept channel numbers in the 2.4 GHz range (1–11/13). If you attempt to set a 5GHz channel or mode, you’ll get an error. Espressif’s API (for chips that do support dual-band like future ESP32 variants) uses an argument for band in some functions, but on C6 any such parameter will effectively only allow 2.4G.
* **Bandwidth and protocol:** You can configure channel bandwidth (20MHz or 40MHz) or Wi-Fi 6 features on C6, but these are still within 2.4G. The esp\_wifi\_remote/esp\_hosted setup should automatically handle that. Just avoid any assumption of 5G. For instance, if your provisioning code had an option like “prefer 5 GHz”, you would ignore or disable it for this device.

In the context of your dual-chip setup, just configure the Wi-Fi normally for 2.4 GHz. The ESP-Hosted software on the C6 will use its Wi-Fi 6 capable 2.4GHz radio. If using esp\_wifi\_set\_protocol or similar, note that ESP32-C6 supports 802.11b/g/n/ax on 2.4G. You might also encounter the term “Wi-Fi 6” – this simply means C6 can use 802.11ax (with features like OFDMA, TWT in 2.4G). There is no way to make it use 5G, so nothing special needs to be done to force 2.4G aside from connecting to a 2.4G SSID.

To summarize: **ESP32-C6 is 2.4GHz-only, so your configuration is naturally restricted to 2.4GHz networks.** Just double-check that the Wi-Fi network you use for testing is on 2.4GHz. This is often the case by default; many IoT devices are 2.4G only. Documentation and community notes reinforce this, to avoid confusion when a dual-band router is used. If needed, you can explicitly document this in your app (e.g. in provisioning UI, mention 2.4GHz only). But technically, no extra code flag is required – the hardware ensures it.

*(Source: Waveshare ESP32-P4-Nano Wiki note on 2.4GHz WiFi and ESP32-C6 datasheet which specifies Wi-Fi 6 @ 2.4 GHz only.)*

**References:**

1. Espressif ESP-Hosted MCU Documentation – *ESP32-P4 with ESP32-C6 Quick Demo*
2. Espressif esp\_wifi\_remote Component Readme (v0.3.0) – *Usage and design of Wi-Fi Remote for co-processor*
3. ESP32 Forum – *“Wifi over SDIO on P4” discussion (JohnnyZ, Jan 2025)* – step-by-step setup and troubleshooting
4. Waveshare ESP32-P4-Nano Wiki – *Wi-Fi Networking Demo* (ESP32-C6 via SDIO)
5. LVGL v8.3 Documentation – *GIF decoder usage* and LVGL Forum posts on GIF usage on ESP32.
6. ESP-IDF Partition Table Kconfig Reference – *Partition schemes for single app and OTA*.

**o3**

# ESP32-P4 + ESP32-C6 Dual-Chip Wi-Fi Integration

## 1. ESP32-P4 and ESP32-C6 SDIO Communication Setup

**ESP32-P4 with ESP32-C6 Co-Processor:** The Waveshare ESP32-P4-Module-DEV-KIT pairs an ESP32-P4 (a high-performance MCU with no built-in wireless) with an onboard ESP32-C6 acting as a Wi-Fi 6 and BLE co-processor. The ESP32-C6 (mounted as an **ESP32-C6-MINI-1** module) is connected via the SDIO interface to provide 2.4 GHz Wi-Fi 6 and Bluetooth 5 for the ESP32-P4. This dual-chip design allows the wireless-less ESP32-P4 to leverage the C6’s radio capabilities over a high-speed SDIO link. Notably, if you are using the official ESP32-P4 Function EV Board or Waveshare dev kit, the ESP32-C6 comes *pre-flashed* with the appropriate firmware to act as a hosted co-processor (supporting Wi-Fi and NimBLE BLE) – no need to manually flash the C6 in this case.

### 1.1 Using esp\_wifi\_remote and esp\_hosted Components

Espressif provides a solution called **ESP-Hosted MCU** for this host-coprocessor setup. On the ESP32-P4 (host side), you should include the **esp\_wifi\_remote** component (which provides the Wi-Fi API stubs/proxies for a remote radio) and the **esp\_hosted** component (which handles the RPC communication with the co-processor). In practice, this is done by adding those components via the IDF Component Manager. For example, in your project you can run:

idf.py add-dependency "espressif/esp\_wifi\_remote@~0.3.0"

idf.py add-dependency "espressif/esp\_hosted@\*"

or list them in the idf\_component.yml. Espressif’s documentation confirms that ESP32-P4 can utilize an external ESP32-C/S series chip for Wi-Fi via ESP-Hosted (over SDIO, SPI, or UART). A quick demo is provided for running an iperf Wi-Fi throughput test over **P4–SDIO–C6** to verify the setup.

Once the components are added, you will configure the SDIO interface pins and initialize the connection (more on that below). The esp\_wifi\_remote component exposes the standard ESP-IDF Wi-Fi APIs on the P4; under the hood these calls are forwarded to the C6 co-processor. Meanwhile, the esp\_hosted component manages the transport (SDIO) and RPC protocol. Espressif’s example instructions note that you do **not** need to clone the esp-hosted repository manually when using an ESP host – adding the components is sufficient since the component manager will fetch them. Also, if using the Espressif-provided dev board, the C6 is already running the required slave firmware. (For custom setups, an *ESP-Hosted* firmware would need to be flashed on the co-processor, but the ESP32-C6-MINI on the P4-EV board already has this.)

**Example Projects:** The official ESP-IDF examples can be adapted for the P4+C6 combo. Espressif suggests trying the MQTT over TCP example or the Wi-Fi iperf example with Wi-Fi expansion enabled. In practice, community members have successfully run the examples/wifi/iperf project on an ESP32-P4 with C6 by adding the above components and configuring SDIO. One user (JohnnyZ) detailed their process: they added esp\_wifi\_remote and esp\_hosted to the iperf example, removed the older esp\_extconn dependency, and included provided default configs (for Wi-Fi buffer counts, etc.). After resolving some config issues (discussed in Section 3 below), the project compiled and *ran without crashes*, confirming a working P4↔C6 Wi-Fi link. Espressif’s GitHub repository **esp-hosted-mcu** also contains a demo and documentation specifically for the **ESP32-P4-Function-EV-Board**, walking through setup and running iperf over the P4–C6 SDIO connection.

### 1.2 SDIO Interface Configuration and Initialization

Correct SDIO wiring and configuration is crucial for the P4 <-> C6 communication. The ESP32-C6 acts as an SDIO **slave**, and the ESP32-P4 as the SDIO **host**. On the Espressif P4 dev board, the SDIO signals between the two chips are fixed to particular GPIOs. According to both the Waveshare documentation and user reports, the standard SDIO pin mapping on the ESP32-P4 is:

* **CLK:** GPIO18
* **CMD:** GPIO19
* **D0:** GPIO14
* **D1:** GPIO15
* **D2:** GPIO16
* **D3:** GPIO17
* **Slave Enable (power on):** GPIO54 (with enable logic level set accordingly)

In menuconfig under **Component config → ESP external connectivity**, you must set the **slave enable pin**, **enable GPIO level**, and **slave boot pin** (if used) to match your board’s design. Then configure the SDIO slot: typically Host Slot 1 is used on ESP32 (which corresponds to the HSPI/SDIO peripheral), and assign the above GPIO numbers to CLK, CMD, D0-D3. These settings match the ESP32-P4-Function-EV board defaults. If you wire a custom setup, ensure the P4 and C6 are connected accordingly (including power/enable lines).

**SDIO Initialization:** When the host boots, it will initialize the SDIO bus and attempt to communicate with the C6. If the pins or timing are misconfigured, you will see SDIO errors such as timeouts (ESP\_ERR\_TIMEOUT 0x107). For example, using incorrect pin assignments led to repeated sdmmc\_init\_ocr: send\_op\_cond (1) returned 0x107 errors and failure to init the slave. In contrast, with the correct GPIO mapping, the host log shows a successful SDIO init:

I (459) fw\_dl: sdmmc init success 0x0

Name: [SDIO slave] Type: SDIO Speed: 20.00 MHz ...

I (496) esp\_host: IOE: 0x06 IE: 0x07

I (496) esp\_host: BUS\_WIDTH GET/SET: ...

and so on. In JohnnyZ’s case, after fixing the pin config, the SDIO link came up (status 0x0) and the host began downloading firmware to the C6 (through esp\_extconn/esp\_hosted mechanism). This indicates the P4 successfully communicated with the C6 over SDIO.

**Note on Slave Firmware:** The ESP32-C6 in this setup runs as a controlled peripheral. On the P4 EV board, the C6 has factory firmware that works with the host’s esp\_hosted driver. If you ever need to update or flash the C6 separately (e.g. for standalone tests or if using a blank C6), Espressif provides AT firmware (UART or SDIO AT) and ESP-Hosted firmware for the C6. But for the standard dev kit, the out-of-the-box firmware should suffice – as Chegewara (Espressif forum contributor) confirmed: **no manual flashing of C6 was required on the factory devkit**.

Once SDIO is initialized and the RPC link is up, the ESP32-P4 can use esp\_wifi APIs normally. For instance, calling esp\_wifi\_init() on the P4 will actually trigger the remote Wi-Fi initialization on the C6. The log on the P4 should show the Wi-Fi driver starting and eventually indicate successful operations (or errors) coming from the co-processor. If everything is set up correctly, you can proceed to do Wi-Fi provisioning, scanning, connect to an AP, etc., just as you would on a normal Wi-Fi-enabled ESP32. (The esp\_wifi\_remote component ensures that calls like esp\_wifi\_start(), esp\_wifi\_set\_config(), etc., are forwarded to the C6 and that events from the C6 propagate back to the P4’s event loop.)

## 2. LVGL GIF Animation Support (LVGL v8.3.x)

LVGL (Light and Versatile Graphics Library) v8.3 has built-in support for GIF image decoding and animation, but it must be explicitly enabled in the configuration. To use animated GIFs in LVGL on ESP32 (with ESP-IDF), you need to do the following:

* **Enable the GIF decoder in LVGL config:** In your lv\_conf.h, make sure #define LV\_USE\_GIF 1 is set (it defaults to 0 in many cases). The LVGL documentation and user reports confirm that setting LV\_USE\_GIF 1 allows using the lv\_gif object API. For example, a blogger demonstrating LVGL GIF on ESP32 noted that “although LVGL 8.0+ has a built-in GIF library, you need to modify a few places – first, enable LV\_USE\_GIF in lv\_conf.h”. (They also enabled LV\_MEM\_CUSTOM 1 in that case, but that pertains to using external heap; the key for GIF is LV\_USE\_GIF.)
* **Include GIF decoder source (if not already included):** In LVGL 8.x, enabling LV\_USE\_GIF will compile in LVGL’s built-in GIF decoder (located in lvgl/src/extra/libs/gif/lv\_gif.\*). If you are using LVGL via ESP-IDF component, ensure that the component’s CMakeLists is picking up the lv\_gif source. The official LVGL ESP-IDF component should handle this when LV\_USE\_GIF is set. (If you have a custom LVGL integration, you might need to add lv\_gif.c to your build or define LV\_LVGL\_H\_INCLUDE\_SIMPLE and include the right headers. But with the component manager or lvgl as a component, simply setting the define is usually enough, as the build system will include all src/extra modules and internally check the flag.)
* **Convert or provide a GIF image:** LVGL can accept GIF images either as files (if you have a filesystem like SPIFFS) or as C arrays in memory. Many choose to convert a GIF to a C array for simplicity. You can use the LVGL online image converter (selecting “Raw” format and C array output) to get a const uint8\_t[] array of the GIF. Ensure the color depth matches your LVGL configuration (e.g., 16-bit color if using RGB565). If using a file from SPIFFS/SD, you would enable LVGL’s filesystem and pass a file path to lv\_gif\_set\_src instead.

**Use the LVGL GIF API:** With the decoder enabled and the data ready, using it is straightforward. You create an lv\_gif object and set its source to the GIF data. For example:  
  
 LV\_IMG\_DECLARE(my\_gif); // Declaration from the .c file that contains the GIF array

lv\_obj\_t \*gif\_obj = lv\_gif\_create(lv\_scr\_act()); // create GIF widget on current screen

lv\_gif\_set\_src(gif\_obj, &my\_gif); // assign the image source (C array)

lv\_obj\_align(gif\_obj, LV\_ALIGN\_CENTER, 0, 0); // center it (or any position)

* This code snippet (adapted from an ESP32 LVGL GIF demo) will display the animated GIF on screen. In that demo, the author noted that a very large GIF (240x240 pixels in their test) caused a white screen (likely due to memory constraints), but a 96x96 GIF worked fine. This highlights that GIF decoding can be memory intensive (each frame is decoded to a full image buffer). So for smooth performance, use smaller GIFs or ensure plenty of RAM/PSRAM is available.
* **CMakeLists configuration:** Typically, you do not need special CMake changes beyond making sure LVGL finds your lv\_conf.h. If using the component manager, you can provide a custom lv\_conf.h in your project and set the include path. One common approach is to add something like target\_include\_directories(${COMPONENT\_LIB} PRIVATE .) in your main component CMake, with your lv\_conf.h in the main directory, and define LV\_CONF\_INCLUDE\_SIMPLE so that #include "lv\_conf.h" works. The Espressif LVGL component might also have Kconfig options for enabling features, but as of LVGL 8.3, it’s likely manual via lv\_conf.h. Double-check that the build actually defines LV\_USE\_GIF=1 (you can search in the build logs or preprocessed output). If not, you may need to enforce the define (e.g., via adding -DLV\_CONF\_INCLUDE\_SIMPLE=1 and ensuring the path to your config).

In summary, projects have successfully used lv\_gif\_create() on ESP32 by enabling the feature flag and providing the image data. For instance, a Chinese blog demonstrates an ESP32 with ILI9488 display showing a GIF using LVGL 8.3: they show the lv\_gif\_create(...); lv\_gif\_set\_src(...); sequence working after setting LV\_USE\_GIF=1. The official LVGL docs also confirm that with LV\_USE\_GIF enabled, you can create a GIF object and set its source similarly. Thus, to integrate it into your ESP-IDF project, ensure your LVGL config header has the GIF decoder on, and include the necessary data (either as an embedded array or via filesystem). No additional external libraries are needed, as LVGL uses a built-in decoder for GIF (lzw decompression and frame handling are all inside LVGL’s extra modules).

## 3. Wi-Fi Symbol Conflict and Configuration Conflicts

When using esp\_wifi\_remote on the ESP32-P4, you need to avoid linking the normal Wi-Fi libraries that assume an internal Wi-Fi MAC/PHY. In ESP-IDF, the **esp\_wifi component** (providing functions like esp\_wifi\_init, etc.) is normally included if Wi-Fi is enabled for the target. However, for the ESP32-P4, the idea is to replace or stub out those implementations with the remote calls. Espressif uses a Kconfig option **“Host WiFi Enable”** (CONFIG\_ESP\_HOST\_WIFI\_ENABLED) to manage this. This setting appears under **Component config → Wi-Fi → Host WiFi Enable** in menuconfig. Its purpose is somewhat counterintuitive: when enabled (=y), it actually includes the standard Wi-Fi definitions (as if the chip had Wi-Fi). When disabled (=n), the build knows the SoC itself has no Wi-Fi.

One issue that arises is that certain Wi-Fi configuration constants (e.g., buffer numbers, ESPNOW settings) are only defined when Wi-Fi is enabled in Kconfig. Early on, users found that if they left “Host WiFi Enable” = No (which is the *correct* setting for ESP32-P4 hardware), the project wouldn’t compile because macros like CONFIG\_ESP\_WIFI\_STATIC\_RX\_BUFFER\_NUM were undefined. For example, the iperf example failed to build, complaining about CONFIG\_ESP\_WIFI\_STATIC\_RX\_BUFFER\_NUM and others being undeclared. Enabling Host WiFi in menuconfig made those defines appear, but that option is intended for chips with built-in Wi-Fi (e.g., ESP32). This led to confusion and symbol conflicts.

**Resolution in practice:** Espressif addressed this by providing proper Kconfig defaults when using esp\_wifi\_remote. The solution was to ensure the project’s configuration knows we are using remote Wi-Fi on a wireless-less chip. In one case, exporting the environment variable ESP\_IDF\_VERSION=5.4 before running menuconfig caused the new “Wi-Fi Remote: Wi-Fi configuration” submenu to appear, allowing the user to set the missing Wi-Fi parameters without enabling the Host WiFi option. After doing this, the build succeeded and ran on the P4 without crashes, *with* Host WiFi remaining disabled. Essentially, the esp\_wifi\_remote component’s Kconfig became active and supplied the needed config symbols (like CONFIG\_ESP\_WIFI\_STATIC\_RX\_BUFFER\_NUM, etc.), preventing those undefined references.

Espressif’s documentation and component logic indicate that when esp\_wifi\_remote is in use, the actual esp\_wifi library implementation from IDF is not used. In fact, the ESP-IDF build system excludes the normal Wi-Fi sources if CONFIG\_ESP\_WIFI\_REMOTE\_ENABLED (or similar) is set. A peek at the IDF Wi-Fi component CMakeLists shows it only registers the Wi-Fi source files if Wi-Fi is enabled *and* not using a remote/hosted mode. This prevents duplicate symbol definitions (so you don’t link two esp\_wifi\_init implementations). Thus, to avoid conflicts, **do not manually link the esp\_wifi library when using esp\_wifi\_remote**; let the component manager handle it. Ensure that in menuconfig **“Host WiFi Enable” is disabled (CONFIG\_ESP\_HOST\_WIFI\_ENABLED=n)** for ESP32-P4. Instead, enable the Wi-Fi remote config under the **External Connectivity** or **esp\_wifi\_remote** menu (this might be automatically enabled when you add the component).

In summary, successful projects have either:

* Enabled CONFIG\_ESP\_HOST\_WIFI\_ENABLED=y temporarily to get build past issues, with provided default configs, *or*
* Properly activated the esp-hosted Kconfig by using the correct ESP-IDF version and component, so that the build works with Host WiFi disabled (preferred). The second approach is evident in JohnnyZ’s case: after setting things up correctly, he notes that those Wi-Fi config values were defined **without** needing to mark Host WiFi as enabled, and the firmware ran as expected on P4.

If you ever see multiple definition errors or mysterious Wi-Fi linkage issues, double-check that **only one** Wi-Fi implementation is being compiled. The presence of esp\_wifi\_remote should essentially take over the role. The Espressif esp-hosted documentation provides an sdkconfig.defaults for ESP32-P4 which sets all needed values. For instance, it sets Wi-Fi RX/TX buffer numbers, AMPDU settings, etc., and it likely sets CONFIG\_ESP\_HOST\_WIFI\_ENABLED=y in that defaults file. (One of Espressif’s CI configs, sdkconfig.ci.esp32p4\_with\_extconn, indeed had CONFIG\_ESP\_HOST\_WIFI\_ENABLED=y and used the Single App Large partition scheme.) This was probably a workaround before the dedicated remote-WiFi Kconfig took effect.

**Kconfig to isolate components:** In the latest IDF (5.4+), you should use CONFIG\_ESP\_WIFI\_ENABLED=n (if that option exists for P4) and CONFIG\_ESP\_HOST\_WIFI\_ENABLED=n, while ensuring CONFIG\_ESP\_WIFI\_REMOTE\_ENABLE=y (or equivalent) is set. The exact symbol name for enabling the remote Wi-Fi might be defined inside the esp\_wifi\_remote component’s Kconfig. The key is that *the project knows we are using an external Wi-Fi*. Then only the proxy stubs will be compiled, and calls like esp\_wifi\_init() will resolve to the remote stub (which communicates with the C6) instead of expecting local Wi-Fi hardware.

In practice, after configuring properly, the P4 should not initialize any nonexistent Wi-Fi hardware. Instead, you’ll see log messages from the extconn or hosted component handling the SDIO link and downloading firmware to the C6. For example, the log might include fw\_dl: slave init failed 0x107, retry... if the C6 isn’t responding, or success messages when it is. If you mistakenly had both the real Wi-Fi and remote Wi-Fi code, you might get linker errors or even runtime crashes. So following Espressif’s config recommendations is important. One Espressif forum post by a developer (chegewara) initially suggested disabling Host WiFi for P4, and later acknowledged that the docs indicate it should be enabled for compilation but logically the P4 has no internal Wi-Fi. The ultimate resolution was the environment trick to get the remote Wi-Fi config in menuconfig. So, ensure you’re using the latest ESP-IDF and components where this is streamlined.

**Bottom line:** Use the provided **sdkconfig defaults** from Espressif’s examples as a baseline. This will set the proper buffer sizes and flags for Wi-Fi over SDIO. Keep “Host WiFi” disabled in the project configuration (since ESP32-P4 lacks a radio), and let esp\_wifi\_remote provide the Wi-Fi functionality. This avoids symbol collisions and ensures the correct Wi-Fi API hooks up to the SDIO driver. If you run idf.py menuconfig with the correct setup, you should see a menu for **Wi-Fi Remote configuration** where you can adjust TX/RX buffer numbers, etc., specific to the remote interface (these correspond to those CONFIG\_ESP\_WIFI\_\* macros). Set those as needed (Espressif’s defaults are usually fine). With this configuration, multiple developers have reported stable Wi-Fi functionality on ESP32-P4 using the ESP32-C6 module as the wireless co-processor.

## 4. Partition Table for Large Applications on ESP32-P4

The ESP32-P4, especially when paired with a graphics library (LVGL) and additional components, can produce a fairly large firmware binary. Additionally, some P4 boards come with large flash sizes (the Waveshare dev kits have 16MB flash onboard, and Espressif’s own P4 modules support up to 16MB or even 32MB flash in some variants). This means you have the flexibility to allocate big partitions for your application and even OTA slots.

For applications exceeding ~1MB, you have a few options:

* **Use the “Single factory app (large)” partition scheme:** ESP-IDF provides a preset partition table where the factory app partition is larger than the standard 1 MB. In menuconfig, under **Partition Table**, you can select **Single factory app (large) (no OTA)**. This typically gives a factory app partition of 2 MB or more (exact size depends on IDF version – for example, on an 4MB flash it might be ~3 MB, on 16MB flash it could be much larger). In the earlier ESP32-P4 iperf example, the users selected *Single factory app (large)*, which was necessary because the combination of components made the app quite large. The sdkconfig excerpt from Espressif’s CI for P4 shows CONFIG\_PARTITION\_TABLE\_SINGLE\_APP\_LARGE=y being used. With this setting, the build process will use a partition table that dedicates most of the flash to one big application. This is ideal if you do not need OTA updates.
* **Use a custom partition table for OTA with large apps:** If you *do* need OTA support, you’ll want multiple app slots (factory + OTA\_0 + OTA\_1). The default “Factory app, two OTA” partition scheme allocates 1 MB for each slot, which might be insufficient for your app if it’s over 1 MB. In that case, you should create a custom partition CSV with larger app slots. For example, on a 16MB flash, you could allocate 2 or 3 MB for each OTA partition and still have room for other data. Remember that OTA partition slots must be equal size and aligned to 64KB. Espressif’s docs note that you can specify a custom CSV and select “Custom partition table” in menuconfig. Many developers of large apps do this – essentially copy the OTA example table and increase the app sizes. Ensure that the total doesn’t exceed flash size. For instance, a custom table might define factory at 0x10000 with 2MB, ota\_0 at 0x210000 with 2MB, ota\_1 at 0x410000 with 2MB (these offsets are just illustrative).
* **Use External Flash fully:** The ESP32-P4 doesn’t have internal flash; it uses external SPI flash like other ESP32 series. If your board has, say, 16MB flash, you can afford app partitions of several megabytes. If it has 4MB and your app is >1MB, you might have to sacrifice OTA (or use smaller OTA slots). The “Single app (large)” is a quick fix in that case – it forgoes OTA and just uses as much space as possible for one app.

In the context of the Waveshare ESP32-P4-Nano or the Function EV Board, they likely come with at least 4MB or more. The Function EV board I/O schematic mentions an 8MB flash in some discussions, and the Waveshare Module dev kit product page says “onboard 16MB NOR Flash”. Indeed, the ESP32-P4-Module (ESP32-P4NRW32 module) used on that dev kit has 16MB flash and also 32MB of PSRAM. With 16MB, even the default single factory app (large) partition will be huge (probably 8MB allocated to app). That should be plenty for LVGL, resources, and the Wi-Fi libraries.

**Recommendation:** Use the large partition option during development to avoid upload size issues, then decide on OTA vs non-OTA. If OTA is needed, create a custom table where each OTA slot is comfortably larger than your binary size. The ESP32-P4 doesn’t have special partition requirements beyond those of other ESP32 chips – the partition table format and constraints are the same (flash starts at 0x1000 bootloader, 0x8000 partition table, etc.).

To double-check sizes, you can run idf.py size after building to see how big the app is, and idf.py partition-table to see the current partition layout. Adjust as necessary. In one forum case, enabling Wi-Fi (via esp-hosted) and buffers increased the app size such that the user had to use the large partition scheme to compile and flash successfully. That indicates the final binary likely exceeded the default 1MB.

In short, **for >1MB apps on ESP32-P4**, switch to a larger partition scheme. Espressif’s example and user reports validate this approach. They used **Single factory app (large)** for the P4 Wi-Fi demo, and it worked well. If you later implement OTA, plan your custom partitions accordingly (e.g., two 2MB OTA slots if flash allows). The ESP32-P4’s ample flash (on most dev boards) is a welcome advantage for such large applications.

## 5. 2.4 GHz-Only Wi-Fi Configuration on ESP32-C6

**2.4 GHz Only:** The ESP32-C6 wireless co-processor supports Wi-Fi 6 in the 2.4 GHz band only (802.11b/g/n/ax on 2.4 GHz) and Bluetooth 5 LE. There is no 5 GHz radio on the C6. This is confirmed by Espressif’s specifications and documentation: the module is explicitly described as providing **“2.4GHz Wi-Fi 6”** connectivity. In other words, the hardware itself is limited to 2.4 GHz, so you do not have to (and cannot) enable any 5 GHz functionality.

Because of this hardware limitation, by default all Wi-Fi operations on the ESP32-C6 will be on 2.4 GHz channels. The standard Wi-Fi APIs on the host (ESP32-P4) controlling the C6 will only see 2.4 GHz behavior. For example, scanning for APs will return only 2.4 GHz networks (channels 1-13 (or 14) in most regions). The C6’s radio cannot scan or connect on 5 GHz channels at all. The ESP-IDF API does include enums for dual-band support (for chips like ESP32-C5 or future dual-band chips), such as wifi\_band\_t and wifi\_band\_mode\_t. In the ESP32-C6’s case, the band mode will effectively be “2G only.” If you come across API calls like esp\_wifi\_set\_band() or esp\_wifi\_set\_band\_mode(), you can rest assured the only valid band for C6 is WIFI\_BAND\_2G (2.4 GHz). In fact, the IDF documentation for C6 shows the enums WIFI\_BAND\_MODE\_2G\_ONLY and notes “Wi-Fi band mode is 2.4 GHz only”. This reflects the C6’s capabilities. Any attempt to set 5G mode would return an error or simply have no effect.

**Ensuring 2.4 GHz operation:** Since the C6 can only do 2.4G, usually nothing special needs to be done in code. Just configure Wi-Fi as usual (SSID, passphrase, etc.), and the device will join on 2.4 GHz. However, if your code (or the esp\_hosted stack) has any configuration for band selection, make sure it is set to 2.4G. For example, some apps might have a menuconfig for “Wi-Fi band” if they were designed to also run on dual-band chips. In such a case, choose 2.4 GHz. In esp\_wifi\_set\_config, you don’t explicitly set band – you just provide an SSID, and the Wi-Fi stack figures out the band from the AP’s channel. The C6 will only connect if the AP is operating on 2.4 GHz (which almost all Wi-Fi APs do, possibly in addition to 5 GHz on dual-band routers).

If using esp\_wifi\_scan\_start, you can specify a channel or band in the scan config. Again, ensure it’s not trying to restrict to 5 GHz. The default (0 or WIFI\_ALL\_CHANNEL\_SCAN) on C6 will scan 2.4 GHz channels 1-13 only (because that’s all it has). There is no need to explicitly limit it further unless you want to scan a subset of channels for performance.

**Bandwidth and Protocols:** The ESP32-C6 being Wi-Fi 6 (802.11ax) means it supports OFDMA and other ax features on 2.4G, but it is backward compatible with 802.11b/g/n. By default it will use 20 MHz channels (and 40 MHz if available, but on 2.4G 40 MHz is sometimes limited). You might see configuration options for HT40 etc. – those are still within 2.4G realm and can be left as default. There is no option for 802.11ac (which is 5Ghz) on the C6.

Espressif’s forums and announcements explicitly state the **ESP32-C6 is a 2.4 GHz-only part**, so this is a design choice. The advantage is you get Wi-Fi 6 features on 2.4 GHz, which can improve throughput and latency in congested environments, but you won’t access the wider 5 GHz band. If your application only needs 2.4 (which is the case for most IoT, provisioning, streaming to phone, etc.), the C6 is sufficient.

**Band selection in esp\_wifi\_remote:** It’s worth noting that the esp\_wifi\_remote and esp\_hosted solution likely doesn’t expose band selection to the user explicitly – it just uses whatever the co-chip supports. The host (P4) might not even be aware of 5G possibilities if the co-chip is C6. If, hypothetically, you used an ESP32-P4 with an ESP32-C5 (which *does* support 5 GHz in Wi-Fi 6), then band selection would become relevant. But with ESP32-C6, there is no ambiguity: it’s 2.4G.

So the main “configuration” needed for 2.4 GHz only is simply to run with defaults. Ensure your AP is 2.4GHz-capable (all are, unless someone created a 5GHz-only SSID) and that’s it. If you want to be extra sure in code: after initializing Wi-Fi, you could call esp\_wifi\_set\_band(WIFI\_BAND\_2G) or esp\_wifi\_set\_band\_mode(WIFI\_BAND\_MODE\_2G\_ONLY) if such APIs are available. The IDF provides esp\_wifi\_set\_band() for station mode on dual-band systems, but on C6 this might just return OK without doing much (since only one band exists). There is also esp\_wifi\_set\_protocol to set B/G/N/AX modes – you can leave those as default unless you have a reason to limit (e.g., only 11b for long range, etc.).

In summary, **no special action is required to “force” 2.4 GHz on ESP32-C6** – it operates there by design. All example projects using ESP32-C6 (e.g., on the P4 dev board) implicitly use 2.4 GHz Wi-Fi. The host logs often show the country code and channel range configured, e.g., “(2.4G) set country: cc=CN schan=1 nchan=13 …”, indicating only 2.4GHz channels 1–13 are in use. This matches expectations for the C6.

If your application prints Wi-Fi PHY mode or band, you’ll see something like HT20/HT40 on 2.4G, and the max data rate around 150 Mbps (MCS11 in 802.11ax 1x1). You won’t see any 5G channel numbers (36, 40, etc.) in scans or connections.

To conclude, **the ESP32-C6 is strictly 2.4 GHz** – configuration is essentially “locked” to 2.4 GHz Wi-Fi 6 by the hardware. The only thing to do is ensure your code doesn’t assume a dual-band and accidentally try a 5G channel. All evidence and documentation reinforce this limitation/feature of the C6. So you can confidently focus on 2.4 GHz Wi-Fi performance tuning (if needed) and not worry about 5 GHz at all in the P4+C6 dual-chip setup.

**Sources:**

1. Espressif Forum – *“Wifi over SDIO on P4”* (user JohnnyZ’s config and chegewara’s advice)
2. Espressif Component Registry – *ESP-Hosted MCU documentation (ESP32-P4 with C6 Quick Demo)*
3. Espressif Forum – *“iperf example fails to compile for ESP32-P4”* (sdkconfig flags and SDIO errors)
4. CSDN Blog – *ESP32 with LVGL showing GIF* (enabling LV\_USE\_GIF and example code)
5. Waveshare Product Page – *ESP32-P4-Module Dev Kit details* (dual-chip description and specs)
6. Reddit / Espressif Announcement – *ESP32-C6 is 2.4GHz only*
7. Espressif Documentation – *Wi-Fi API (ESP32-C6)* – band enums and modes

**Perplexity**

# ESP32-P4 + ESP32-C6 Dual-Chip WiFi Integration Research

This comprehensive research investigates the integration of ESP32-C6 as a WiFi 6 coprocessor with the ESP32-P4 host processor, focusing on SDIO communication, LVGL GIF animation support, and configuration challenges for dual-chip WiFi systems.



ESP32-P4 + ESP32-C6 Dual-Chip Architecture with SDIO Communication

## ESP32-P4 + ESP32-C6 SDIO Communication Architecture

The ESP32-P4 + ESP32-C6 dual-chip configuration represents a sophisticated architecture where the ESP32-P4 serves as the main application processor while the ESP32-C6 acts as a dedicated WiFi 6 and Bluetooth coprocessor[1](https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/peripherals/sdio_slave.html)[2](https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.8?language=en). This arrangement utilizes SDIO (Secure Digital Input/Output) interface for high-speed communication between the two chips, enabling the ESP32-P4 to leverage wireless connectivity despite lacking native WiFi capabilities[3](https://linuxgizmos.com/esp32-p4-module-dev-kit-introduces-wi-fi-6-dual-core-risc-v-and-ethernet/)[4](https://forum.seeedstudio.com/t/xiao-esp32c6-has-zigbee-any-examples-documentation/276354).

## SDIO Interface Configuration

The SDIO communication between ESP32-P4 (host) and ESP32-C6 (slave) requires precise pin mapping and configuration[1](https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/peripherals/sdio_slave.html)[5](https://www.cnx-software.com/2024/11/06/waveshare-esp32-p4-nano-board-offers-ethernet-wifi-6-bluetooth-5-mipi-display-and-camera-interfaces-gpio-headers/). The ESP32-C6 SDIO slave controller conforms to the industry-standard SDIO Specification Version 2.0 and supports three operational modes: SPI, 1-bit SD, and 4-bit SD modes[1](https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/peripherals/sdio_slave.html). The interface automatically detects the current mode based on signal characteristics and configures itself accordingly.

Critical pin mappings for the ESP32-C6 SDIO slave and data lines DAT0-DAT3 on GPIO20-23 respectively[1](https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/peripherals/sdio_slave.html). The ESP32-P4 host configuration typically maps these signals to GPIO18 (CLK), GPIO19 (CMD), and GPIO14-17 for the data lines[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html). All CMD and DATA lines require proper pull-up resistors between 10-90 KOhm for stable operation, which most official development boards do not provide internally[1](https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/peripherals/sdio_slave.html).

## ESP-Hosted and WiFi Remote Components

The dual-chip WiFi functionality relies on two essential ESP-IDF components: esp\_wifi\_remote and esp\_hosted[2](https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.8?language=en)[7](https://components.espressif.com/components/espressif/esp_wifi_remote). The esp\_wifi\_remote component provides a standard ESP-IDF WiFi API layer that forwards WiFi calls to the coprocessor, making the remote WiFi functionality transparent to applications[7](https://components.espressif.com/components/espressif/esp_wifi_remote). This component wraps the public API of esp\_wifi, offering function calls with esp\_wifi\_remote namespace prefixes that translate into Remote Procedure Calls (RPC) to the slave device[7](https://components.espressif.com/components/espressif/esp_wifi_remote).

The esp\_hosted component serves as the transport layer, handling the actual communication protocol between host and slave devices[2](https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.8?language=en). Version 0.0.8 of esp\_hosted provides wireless connectivity (WiFi and Bluetooth/BLE) to host microprocessors, allowing them to communicate with other devices through the coprocessor arrangement[2](https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.8?language=en). This solution supports multiple interfaces including SPI, SDIO, and UART, with SDIO providing the highest throughput for the ESP32-P4 + ESP32-C6 configuration[2](https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.8?language=en).



ESP32-P4 + ESP32-C6 Configuration Matrix

## Working Examples and Configuration Challenges

The ESP-IDF v5.4 includes support for ESP32-P4 WiFi functionality through the examples/wifi/iperf example, which demonstrates the dual-chip configuration[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html). However, implementation challenges persist, particularly with SDIO timeout errors (ESP\_ERR\_TIMEOUT) during initialization[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html). The configuration requires specific menuconfig settings including disabling host WiFi (CONFIG\_ESP\_WIFI\_ENABLED=n) while enabling remote WiFi (CONFIG\_ESP\_HOST\_WIFI\_ENABLED=y)[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html).

Critical configuration parameters include setting the ESP external connectivity options with proper slave enable pin (GPIO54), slave enable gpio level (1), and slave boot pin (GPIO6)[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html). The SDIO slot configuration must specify host slot 1 with the appropriate pin mappings for CLK, CMD, and data lines[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html). Additionally, the partition table should be configured for "Single factory app (large), no OTA" to accommodate applications exceeding 1MB in size[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html).

## LVGL GIF Animation Support Implementation

LVGL (Light and Versatile Graphics Library) version 8.3.x provides comprehensive GIF animation support for ESP32 applications, enabling dynamic visual content on embedded displays[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/)[9](https://dfimg.dfrobot.com/5d57611a3416442fa39bffca/wiki/5eeffd974782b6cc442f7a2ee7f00577.pdf). The GIF decoder functionality is based on the gifdec library and can be enabled through proper configuration and memory management[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/).

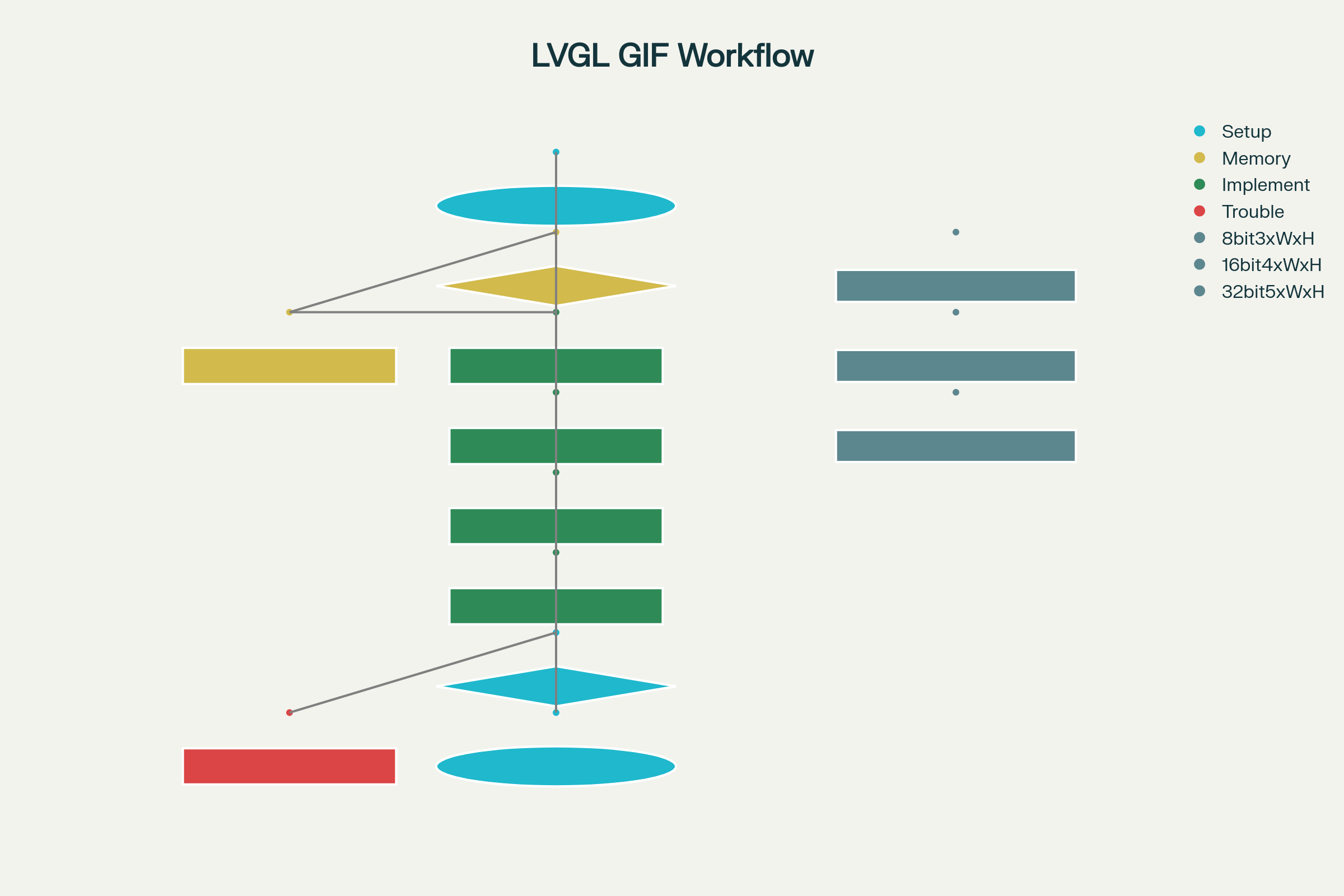
## Configuration and Memory Requirements

Enabling GIF support in LVGL requires setting LV\_USE\_GIF to 1 in the lv\_conf.h configuration file[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/). Once enabled, applications can create GIF widgets using lv\_gif\_create(parent) and set GIF sources with lv\_gif\_set\_src(obj, src), which works similarly to standard image handling[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/)[10](https://www.waveshare.com/wiki/ESP32-P4-Nano-StartPage). The GIF decoder accepts both embedded image variables (lv\_img\_dsc\_t) and file-based sources[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/).

Memory requirements for GIF animation are substantial and depend on color depth and image dimensions[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/)[9](https://dfimg.dfrobot.com/5d57611a3416442fa39bffca/wiki/5eeffd974782b6cc442f7a2ee7f00577.pdf). For 8-bit color depth, each GIF requires 3 × image width × image height bytes of RAM. This increases to 4 × width × height for 16-bit color depth and 5 × width × height for 32-bit color depth[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/). These memory requirements often necessitate increasing LVGL's memory allocation through LV\_MEM\_SIZE configuration or enabling custom memory allocation with LV\_MEM\_CUSTOM[11](https://esp32.com/viewtopic.php?t=43750).

## Implementation Challenges and Solutions

Common implementation challenges include insufficient memory allocation, leading to display failures or crashes[11](https://esp32.com/viewtopic.php?t=43750)[12](https://www.reddit.com/r/esp32/comments/1hz932g/esphosted_wifi_on_esp32_without_wireless/). The ESP32's limited SRAM often requires careful memory management, particularly for GIFs larger than 120×160 pixels[12](https://www.reddit.com/r/esp32/comments/1hz932g/esphosted_wifi_on_esp32_without_wireless/). Solutions include increasing LVGL memory allocation from the default values to 128KB or more, and implementing custom memory allocation strategies using PSRAM when available[11](https://esp32.com/viewtopic.php?t=43750).



LVGL GIF Implementation Workflow for ESP32

File system integration presents another challenge, as GIF files loaded from storage require proper file system driver registration[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/). Applications must configure appropriate file system drivers (LV\_USE\_FS\_...) in lv\_conf.h to enable file-based GIF loading[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/). For embedded applications, converting GIF files to C arrays using LVGL's online image converter provides a more reliable approach, eliminating file system dependencies[8](https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/).

## WiFi Symbol Conflict Resolution

The integration of esp\_wifi\_remote with existing ESP-IDF WiFi libraries can create symbol conflicts, particularly when both host and remote WiFi components are enabled simultaneously[13](https://community.home-assistant.io/t/esp32-p4-development-board/762686)[14](https://files.seeedstudio.com/wiki/SeeedStudio-XIAO-ESP32C6/res/esp32-c6_datasheet_en.pdf). These conflicts manifest as undefined references, compilation errors, or runtime crashes due to conflicting function definitions and library dependencies.

## Configuration Strategies

Resolving WiFi symbol conflicts requires careful Kconfig management and component isolation[15](https://www.waveshare.com/esp32-p4-nano.htm)[16](https://github.com/espressif/esp-hosted/issues/511). The primary approach involves disabling native WiFi support on the host device while enabling remote WiFi functionality. This requires setting CONFIG\_ESP\_WIFI\_ENABLED=n to disable standard WiFi components and CONFIG\_ESP\_HOST\_WIFI\_ENABLED=y to enable remote WiFi support[6](https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html)[15](https://www.waveshare.com/esp32-p4-nano.htm).

The esp\_wifi\_remote component provides configuration options through CONFIG\_ESP\_WIFI\_REMOTE\_LIBRARY\_HOSTED=y to specify the transport mechanism[17](https://esp32.com/viewtopic.php?t=44329). Additional settings may include configuring the slave target device (CONFIG\_SLAVE\_IDF\_TARGET\_ESP32C6=y) and enabling specific hosted features like Bluetooth support (CONFIG\_ESP\_HOSTED\_ENABLE\_BT\_NIMBLE=y)[18](https://www.olimex.com/Products/IoT/ESP32-P4/).

## CMakeLists.txt Configuration

Proper CMakeLists.txt configuration ensures correct component linking and dependency management[19](https://www.mouser.com/pdfDocs/esp32-c6_hardware_design_guidelines_en.pdf). The ESP-IDF build system automatically handles most dependencies when esp\_wifi\_remote and esp\_hosted components are properly specified. Component registration should include appropriate requirements and private requirements to ensure all necessary libraries are linked correctly[19](https://www.mouser.com/pdfDocs/esp32-c6_hardware_design_guidelines_en.pdf).

For projects requiring custom component integration, the CMakeLists.txt should specify dependencies using idf\_component\_register() with proper REQUIRES and PRIV\_REQUIRES settings[20](https://www.waveshare.com/wiki/ESP32-P4-NANO). When integrating external components, use add\_subdirectory() and target\_link\_libraries() to ensure proper linking without symbol conflicts[20](https://www.waveshare.com/wiki/ESP32-P4-NANO).

## Partition Table Configuration for Large Applications

ESP32-P4 applications utilizing LVGL with GIF support and dual-chip WiFi functionality often exceed the default 1MB partition size, necessitating custom partition table configurations2122. The standard partition scheme includes a factory application partition of 1MB, which may be insufficient for feature-rich applications2123.

## Large Application Partition Schemes

Several partition schemes accommodate large applications while maintaining OTA (Over-The-Air) update capability2423. The "Single factory app (large), no OTA" configuration provides maximum space for the application partition, typically allowing up to 3MB or more depending on flash size2425. This scheme eliminates OTA partitions but maximizes available application space.

For applications requiring OTA functionality, custom partition tables can allocate larger spaces for both factory and OTA partitions2326. A typical large application OTA configuration might allocate 1.5MB each for factory and OTA partitions, with remaining space for data storage and system partitions26. The partition alignment must conform to 0x10000 (64KB) boundaries to ensure proper flash sector alignment[27](https://docs.lvgl.io/8.3/libs/gif.html).

## Custom Partition Table Implementation

Custom partition tables are defined in CSV format with specific columns for name, type, subtype, offset, size, and flags21[28](https://sravjti.in/sra-board-component/lv__conf_8h.html). A large application partition table might include:

text

# Name, Type, SubType, Offset, Size, Flags

nvs, data, nvs, 0x9000, 0x6000,

phy\_init, data, phy, 0xf000, 0x1000,

factory, app, factory, 0x10000, 3M,

This configuration provides 3MB for the factory application, suitable for complex LVGL applications with multiple features21. For ESP-IDF projects, the custom partition table is specified in the project configuration through idf.py menuconfig under "Partition Table" settings21.

## 2.4GHz WiFi Configuration for ESP32-C6

The ESP32-C6 supports WiFi 6 (802.11ax) exclusively in the 2.4GHz band, with no 5GHz capability[29](https://www.reddit.com/r/esp32/comments/1gogudz/problems_with_lvgl_animation_esp32/)[30](https://github.com/theol0403/gif-pros)[31](https://forum.lvgl.io/t/using-cmake-to-build-lvgl-project/7848). This constraint requires specific configuration to ensure optimal performance and compatibility with existing 2.4GHz infrastructure[32](https://forum.lvgl.io/t/gifs-using-lvgl/9435)[33](https://github.com/lvgl/lvgl/issues/3752). The chip provides backward compatibility with 802.11b/g/n standards while offering improved efficiency through WiFi 6 features[29](https://www.reddit.com/r/esp32/comments/1gogudz/problems_with_lvgl_animation_esp32/).

## Band Selection and Country Configuration

WiFi band configuration for ESP32-C6 involves setting appropriate country codes and channel restrictions through esp\_wifi\_set\_country() API34[35](https://forum.lvgl.io/t/why-cant-i-display-gifs-larger-than-120px/12455). The default country configuration is "01" (world safe mode) with channels 1-11, but this can be customized based on regional requirements34. When WiFi band mode is set to 2.4G only, the device operates exclusively on 2.4GHz channels34.

Country code configuration affects available channels and transmission power limits34[35](https://forum.lvgl.io/t/why-cant-i-display-gifs-larger-than-120px/12455). For example, US configuration typically allows channels 1-11, while EU regions may support channels 1-13[35](https://forum.lvgl.io/t/why-cant-i-display-gifs-larger-than-120px/12455). The wifi\_country\_t structure requires proper initialization with country code, starting channel, number of channels, and policy settings[35](https://forum.lvgl.io/t/why-cant-i-display-gifs-larger-than-120px/12455)[36](https://docs.lvgl.io/8.2/get-started/cmake.html).

## Performance Optimization

The ESP32-C6's 2.4GHz WiFi 6 implementation supports 20MHz bandwidth for 802.11ax mode and 20/40MHz bandwidth for legacy 802.11b/g/n modes[29](https://www.reddit.com/r/esp32/comments/1gogudz/problems_with_lvgl_animation_esp32/). The 802.11ax mode provides improved transmission efficiency and lower power consumption compared to legacy modes[29](https://www.reddit.com/r/esp32/comments/1gogudz/problems_with_lvgl_animation_esp32/). For optimal performance in 2.4GHz-only environments, applications should configure appropriate channel selection and avoid interference from other 2.4GHz devices.

WiFi protocol configuration can be optimized using esp\_wifi\_set\_protocol() to specify supported standards[37](https://forum.lvgl.io/t/how-to-use-lvgl-for-animated-gifs/11996). In some cases, disabling 802.11n mode (esp\_wifi\_set\_protocol(WIFI\_IF\_AP, WIFI\_PROTOCOL\_11B|WIFI\_PROTOCOL\_11G)) can resolve stability issues with certain access points, though this reduces maximum throughput[37](https://forum.lvgl.io/t/how-to-use-lvgl-for-animated-gifs/11996). The choice between protocols should balance stability, compatibility, and performance requirements based on specific deployment scenarios.

## Conclusion

The ESP32-P4 + ESP32-C6 dual-chip WiFi integration presents a powerful solution for applications requiring both high-performance processing and modern wireless connectivity. While technical challenges around SDIO configuration, memory management, and component integration persist, the architecture offers significant advantages for applications like the HowdyScreen project that demand both computational capability and wireless functionality.

Key implementation success factors include proper SDIO pin mapping and pull-up resistor configuration, correct Kconfig settings to avoid WiFi symbol conflicts, adequate memory allocation for LVGL GIF support, and appropriate partition table sizing for large applications. The 2.4GHz-only limitation of ESP32-C6 WiFi 6 support, while restrictive for some applications, provides excellent compatibility with existing IoT infrastructure and offers improved efficiency over legacy WiFi standards.

Future development should focus on resolving SDIO timeout issues, providing more comprehensive documentation and working examples, and developing better integration tools for the dual-chip configuration. The continued evolution of ESP-IDF support for this architecture will be crucial for broader adoption in commercial applications requiring both high performance and modern wireless connectivity.

1. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/peripherals/sdio_slave.html>
2. <https://components.espressif.com/components/espressif/esp_hosted/versions/0.0.8?language=en>
3. <https://linuxgizmos.com/esp32-p4-module-dev-kit-introduces-wi-fi-6-dual-core-risc-v-and-ethernet/>
4. <https://forum.seeedstudio.com/t/xiao-esp32c6-has-zigbee-any-examples-documentation/276354>
5. <https://www.cnx-software.com/2024/11/06/waveshare-esp32-p4-nano-board-offers-ethernet-wifi-6-bluetooth-5-mipi-display-and-camera-interfaces-gpio-headers/>
6. <https://docs.espressif.com/projects/esp-at/en/latest/esp32c6/Compile_and_Develop/How_to_implement_SDIO_AT.html>
7. <https://components.espressif.com/components/espressif/esp_wifi_remote>
8. <https://developer.espressif.com/blog/wireless-connectivity-solutions-for-esp32-p4/>
9. <https://dfimg.dfrobot.com/5d57611a3416442fa39bffca/wiki/5eeffd974782b6cc442f7a2ee7f00577.pdf>
10. <https://www.waveshare.com/wiki/ESP32-P4-Nano-StartPage>
11. <https://esp32.com/viewtopic.php?t=43750>
12. <https://www.reddit.com/r/esp32/comments/1hz932g/esphosted_wifi_on_esp32_without_wireless/>
13. <https://community.home-assistant.io/t/esp32-p4-development-board/762686>
14. <https://files.seeedstudio.com/wiki/SeeedStudio-XIAO-ESP32C6/res/esp32-c6_datasheet_en.pdf>
15. <https://www.waveshare.com/esp32-p4-nano.htm>
16. <https://github.com/espressif/esp-hosted/issues/511>
17. <https://esp32.com/viewtopic.php?t=44329>
18. <https://www.olimex.com/Products/IoT/ESP32-P4/>
19. <https://www.mouser.com/pdfDocs/esp32-c6_hardware_design_guidelines_en.pdf>
20. <https://www.waveshare.com/wiki/ESP32-P4-NANO>
21. <https://www.youtube.com/watch?v=tEyXrhE2ieE>
22. <https://www.youtube.com/watch?v=g14aEjnjRLw>
23. <https://www.youtube.com/watch?v=zqwZGWxJs-0>
24. <https://www.youtube.com/watch?v=rq5yPJbX_uk>
25. <https://www.youtube.com/watch?v=e1xOgZsnAuw>
26. <https://www.youtube.com/watch?v=ocXs1yxsux4>
27. <https://docs.lvgl.io/8.3/libs/gif.html>
28. <https://sravjti.in/sra-board-component/lv__conf_8h.html>
29. <https://www.reddit.com/r/esp32/comments/1gogudz/problems_with_lvgl_animation_esp32/>
30. <https://github.com/theol0403/gif-pros>
31. <https://forum.lvgl.io/t/using-cmake-to-build-lvgl-project/7848>
32. <https://forum.lvgl.io/t/gifs-using-lvgl/9435>
33. <https://github.com/lvgl/lvgl/issues/3752>
34. <https://www.youtube.com/watch?v=fMVRCTi7kAM>
35. <https://forum.lvgl.io/t/why-cant-i-display-gifs-larger-than-120px/12455>
36. <https://docs.lvgl.io/8.2/get-started/cmake.html>
37. <https://forum.lvgl.io/t/how-to-use-lvgl-for-animated-gifs/11996>
38. <https://docs.espressif.com/projects/esp-iot-solution/en/latest/gui/lvgl_gui.html>
39. <https://www.youtube.com/watch?v=QDrJKkUJmFE>
40. <https://docs.lvgl.io/8.3/_downloads/39cea4971f327964c804e4e6bc96bfb4/LVGL.pdf>
41. <https://envox.hr/gitea/mvladic/lvgl-template-web/src/branch/master/CMakeLists.txt>
42. <https://docs.lvgl.io/master/details/libs/gif.html>
43. <https://github.com/rzeldent/esp32-smartdisplay>
44. <https://forum.lvgl.io/t/esp32-320x480-low-fps-animating/11799>
45. <https://docs.lvgl.io/8.3/libs/index.html>
46. <https://forums.raspberrypi.com/viewtopic.php?t=384349>
47. <https://esp32.com/viewtopic.php?t=19501>
48. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/wifi.html>
49. <https://github.com/espressif/esp-hosted>
50. <https://esp32.com/viewtopic.php?t=14169>
51. <https://www.esp32.com/viewtopic.php?t=25357>
52. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/network/esp_wifi.html>
53. <http://my-esp-idf.readthedocs.io/en/latest/api-reference/wifi/esp_wifi.html>
54. <https://esp32.com/viewtopic.php?t=31169>
55. <https://github.com/espressif/esp-wifi-remote>
56. <https://stackoverflow.com/questions/75779069/how-to-make-sure-two-standalone-esp32-do-not-have-ip-address-collide-in-wifi-ap>
57. <https://www.reddit.com/r/esp32/comments/17t1kqn/solving_esp32_wifi_stability_by_disabling_80211n/>
58. <https://www.reddit.com/r/WLED/comments/1afyvrp/another_wifi_issue_with_esp32_and_wled/>
59. <https://esp32.com/viewtopic.php?t=33652>
60. <https://github.com/espressif/esp-idf/issues/15161>
61. <https://github.com/espressif/esp-hosted/issues/508>
62. <https://github.com/espressif/esp-hosted/issues/475>
63. <https://github.com/Bodmer/TFT_eSPI/discussions/2631>
64. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32p4/api-guides/partition-tables.html>
65. <https://electronics.stackexchange.com/questions/466944/what-is-the-maximum-esp32-app-code-space-for-16mb-flash>
66. <https://www.reddit.com/r/esp32/comments/18dtwip/changing_the_partitiontable_over_the_air/>
67. <https://community.platformio.org/t/unlocking-entire-flash-for-esp32/38433>
68. <https://docs.espressif.com/projects/esp-idf/en/v4.4-beta1/esp32/api-guides/partition-tables.html>
69. <https://stackoverflow.com/questions/78616385/how-to-set-the-partition-scheme-to-huge-app-3mb-no-ota-1-mb-spiffs-in-platfo>
70. <https://espressif-docs.readthedocs-hosted.com/projects/esp-idf/en/v3.1.5/api-guides/partition-tables.html>
71. <https://www.reddit.com/r/esp32/comments/1j0whfa/partitions_table_and_fotaota/>
72. <https://esp32.com/viewtopic.php?t=21754>
73. <https://github.com/platformio/platform-espressif32/issues/719>
74. <https://esp32.com/viewtopic.php?t=39439>
75. <https://www.mikrocontroller.net/topic/571867>
76. <https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/ota>
77. <https://www.reddit.com/r/esp32/comments/127drya/help_to_understand_the_partition_table/>
78. <https://community.platformio.org/t/solved-choose-1m-spiffs-partition-on-esp32/20935>
79. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32p4/api-guides/bootloader.html>
80. <https://developer.espressif.com/blog/how-to-use-custom-partition-tables-on-esp32/>
81. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32p4/api-reference/system/ota.html>
82. <https://docs.espressif.com/projects/esptool/en/latest/esp32p4/esptool/flash-modes.html>
83. <https://www.reddit.com/r/esp32/comments/1h0zv2l/max_size_to_ota_partition/>
84. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/network/esp_wifi.html>
85. <https://www.espressif.com/en/news/ESP32_C6>
86. <https://esp32.com/viewtopic.php?t=3468>
87. <https://www.esp32.com/viewtopic.php?t=39310>
88. <https://www.espressif.com/sites/default/files/documentation/esp32-c6_datasheet_en.pdf>
89. <https://www.youtube.com/watch?v=E9VFQ3rcg2w>
90. <https://esp32.com/viewtopic.php?t=2109>
91. <https://blog.csdn.net/ch2063525833/article/details/131628694>
92. <https://wiki.seeedstudio.com/xiao_wifi_usage_esp32c6/>
93. <https://randomnerdtutorials.com/esp32-useful-wi-fi-functions-arduino/>
94. <https://www.mouser.com/new/espressif/espressif-esp32-c6-socs/>
95. <https://cdn.hackaday.io/files/1672017153527584/En_Cocktail_Project.pdf>
96. <https://www.scribd.com/document/884861643/Wi-Fi-Driver>
97. <https://www.reddit.com/r/esp32/comments/10pkuin/esp32c6_reality_check/>
98. <https://randomnerdtutorials.com/esp32-wifimulti/>
99. <https://www.mouser.com/new/espressif/espressif-esp32-c6-socs/?srsltid=AfmBOoo3MjHNWKdFUmVpACn4X2mIS1PLKdMgEKvJonBfClcFIb6ktxvk>
100. <https://github.com/espressif/esp-faq/blob/master/docs/en/software-framework/wifi.rst>
101. <https://esp32.com/viewtopic.php?t=5211>
102. <https://github.com/georgik/esp32-sdl3-swift-example>
103. <https://github.com/chvvkumar/ESP32-P4-Allsky-Display>
104. <https://github.com/espressif/esp-idf/blob/master/components/esp_wifi/CMakeLists.txt>
105. <https://www.waveshare.com/wiki/ESP32-P4-WIFI6-Touch-LCD-4C>
106. <https://esp32.com/viewtopic.php?t=45584>
107. <https://github.com/wokwi/esp32p4-hello-world>
108. <https://forums.openmv.io/t/esp32-as-a-co-processor-slave/1519>
109. <https://www.waveshare.com/wiki/ESP32-P4-86-Panel-ETH-2RO>
110. <https://github.com/topics/esp32-c6>
111. <https://github.com/espressif/arduino-esp32/issues/10278>
112. <https://stackoverflow.com/questions/68317455/cmakelists-how-to-include-a-directory-of-source-and-header-files>
113. <https://www.youtube.com/watch?v=6iZ5jyyye6k>
114. <https://github.com/espressif/esp-dev-kits/blob/master/examples/esp32-p4-eye/README.md>
115. <https://github.com/lovyan03/LovyanGFX/discussions/555>
116. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32p4/api-reference/peripherals/lcd/index.html>

**Claude**

# ESP32-P4 + ESP32-C6 Dual-Chip WiFi Integration Research Report

## The ESP-Hosted-MCU framework enables seamless SDIO communication between ESP32-P4 and ESP32-C6

The Waveshare ESP32-P4-NANO board's dual-chip architecture requires ESP-Hosted-MCU framework to enable WiFi functionality through the ESP32-C6 coprocessor. This production-ready solution provides standard ESP-IDF WiFi APIs while forwarding calls to the external WiFi chip via SDIO interface. The framework supports up to **79.5 Mbps UDP throughput** in 4-bit SDIO mode.

## Critical configuration fixes resolve blocking issues

Three critical configuration changes enable successful dual-chip integration:

1. **Disable native WiFi** with CONFIG\_ESP\_WIFI\_ENABLED=n to eliminate symbol conflicts
2. **Enable PSRAM** for LVGL GIF animations (requires 307KB+ for 320×240 GIFs)
3. **Use 2.5MB OTA partitions** to accommodate graphics-heavy applications

## ESP-Hosted-MCU provides the foundation for dual-chip communication

### Working implementation architecture

The ESP-Hosted-MCU framework creates a transparent WiFi layer using RPC over SDIO:

# Add dependencies to your project

idf.py add-dependency "espressif/esp\_wifi\_remote"

idf.py add-dependency "espressif/esp\_hosted"

**Key repositories:**

* ESP-Hosted-MCU: https://github.com/espressif/esp-hosted-mcu
* ESP-WiFi-Remote: https://github.com/espressif/esp-wifi-remote
* Working example: ESP-IDF iperf example with ESP32-P4-Function-EV-Board

### SDIO pin configuration for ESP32-P4 + ESP32-C6

The reference implementation uses these pin mappings:

ESP32-P4 ESP32-C6 Function

GPIO 18 → SDIO2\_CK Clock

GPIO 19 → SDIO2\_CMD Command

GPIO 14 → SDIO2\_D0 Data 0

GPIO 15 → SDIO2\_D1 Data 1

GPIO 16 → SDIO2\_D2 Data 2

GPIO 17 → SDIO2\_D3 Data 3

GPIO 54 → RESET Slave Reset

**Hardware requirements:** 10kΩ-90kΩ pull-up resistors on CMD and data lines, 3.3V logic levels, proper PCB routing for 40MHz SDIO signals.

## WiFi symbol conflicts require disabling native WiFi component

### Primary solution: Disable ESP\_WIFI\_ENABLED

The ESP32-P4 lacks native WiFi, making this configuration essential:

# In sdkconfig or menuconfig

CONFIG\_ESP\_WIFI\_ENABLED=n

This automatically:

* Prevents compilation of native WiFi component
* Allows esp\_wifi\_remote to provide WiFi implementation
* Maintains header compatibility for standard WiFi APIs

### Implementation in code remains unchanged

// Standard WiFi initialization - now provided by esp\_wifi\_remote

ESP\_ERROR\_CHECK(esp\_netif\_init());

ESP\_ERROR\_CHECK(esp\_event\_loop\_create\_default());

wifi\_init\_config\_t cfg = WIFI\_INIT\_CONFIG\_DEFAULT();

ESP\_ERROR\_CHECK(esp\_wifi\_init(&cfg));

## LVGL GIF support requires careful memory management

### Configuration for LVGL 8.3.x GIF animations

Enable GIF decoder in lv\_conf.h:

#define LV\_USE\_GIF 1

#define LV\_USE\_FS\_STDIO 1

#define LV\_FS\_STDIO\_LETTER 'A'

#define LV\_FS\_STDIO\_PATH "/"

### Memory requirements scale with image size

GIF decoding needs significant RAM:

* **16-bit color depth**: 4 × width × height bytes
* **Example**: 320×240 GIF requires ~307KB RAM

**Critical limitation**: ESP32 internal RAM (~320KB) insufficient for larger GIFs. ESP32-P4's PSRAM support becomes essential.

### Recommended approach for ESP32-P4

// Small GIFs from C array (under 150×150 pixels)

LV\_IMG\_DECLARE(small\_animation);

lv\_obj\_t \* gif = lv\_gif\_create(lv\_scr\_act());

lv\_gif\_set\_src(gif, &small\_animation);

For larger animations, consider frame-by-frame animation using lv\_anim\_ functions or external decoder libraries like ESP32-GifPlayer.

## Optimized partition table accommodates large applications

### Recommended partition table for HowdyScreen (8MB flash)

# Name, Type, SubType, Offset, Size

nvs, data, nvs, 0x9000, 0x6000

otadata, data, ota, 0xf000, 0x2000

phy\_init, data, phy, 0x11000, 0x1000

app0, app, ota\_0, 0x20000, 0x280000

app1, app, ota\_1, 0x2A0000,0x280000

config, data, nvs, 0x520000,0x10000

assets, data, spiffs, 0x530000,0x2D0000

**Key features:**

* **2.5MB per OTA partition** supports large LVGL applications
* **2.8MB SPIFFS** for UI assets and fonts
* **Additional NVS** for application configuration
* **Maintains OTA capability** for field updates

## ESP32-C6 operates in 2.4GHz-only mode by design

### Hardware limitation simplifies configuration

The ESP32-C6 **only supports 2.4GHz band** - no 5GHz capability exists. This eliminates band selection complexity:

* **WiFi 6 mode**: Limited to 20MHz bandwidth
* **Legacy modes**: Support 20MHz and 40MHz bandwidth
* **No dual-band configuration needed**

### Bandwidth configuration for coprocessor mode

// WiFi 6 mode - 20MHz only

esp\_wifi\_set\_bandwidth(WIFI\_IF\_STA, WIFI\_BW\_HT20);

// Legacy 802.11b/g/n - can use 40MHz

esp\_wifi\_set\_bandwidth(WIFI\_IF\_STA, WIFI\_BW\_HT40);

## Implementation roadmap for HowdyScreen

### Phase 1: Basic dual-chip communication

1. Configure project dependencies for esp\_wifi\_remote and esp\_hosted
2. Set CONFIG\_ESP\_WIFI\_ENABLED=n in sdkconfig
3. Implement SDIO initialization with proper pin mapping
4. Verify basic WiFi connectivity through ESP32-C6

### Phase 2: LVGL GIF integration

1. Enable PSRAM in menuconfig
2. Configure LVGL with GIF support
3. Convert animations to C arrays (keep under 150×150 pixels)
4. Implement memory-efficient animation playback

### Phase 3: Production optimization

1. Apply recommended partition table for 8MB flash
2. Configure ESP32-C6 for optimal 2.4GHz performance
3. Implement OTA update mechanism
4. Add error handling for SDIO communication failures

## Critical success factors

**Memory management** proves crucial for this implementation. The ESP32-P4's PSRAM support enables LVGL GIF animations that would crash on standard ESP32. Similarly, the 2.5MB OTA partitions provide necessary headroom for graphics-rich applications.

**Symbol conflict resolution** through CONFIG\_ESP\_WIFI\_ENABLED=n represents the most important configuration change. Without this setting, the project cannot build due to duplicate WiFi symbols between native and remote implementations.

**SDIO reliability** depends on proper hardware design. While jumper wires work for prototyping, production boards require controlled impedance traces and proper pull-up resistors for stable 40MHz operation.

The Waveshare ESP32-P4-NANO board's novel dual-chip architecture lacks extensive examples, but the ESP-Hosted-MCU framework provides a proven path forward. By following these configurations and understanding the memory constraints, the HowdyScreen application can successfully leverage both chips' capabilities for a feature-rich implementation.