partition table 使用说明

简介

分区表（partition table ）是 ESP-IDF 框架中实现的一种分配 flash 的方式 ，对 `spi_flash_{read|write|erase}` 等接口进行了封装，增加了越界检查，同时尽量避免了直接操作 flash 地址，所以更加简单安全，但分区表擦写操作本身不具备擦写均衡，如果希望使用分区表存储用户数据，只建议存储不会频繁更改的数据。

分区表将一块完整的 flash 根据模块进行划分，规定了每个模块使用的 flash 区域和大小。分区表本身也占用存储在 flash 中（默认为地址 0x8000 ）。系统初始化时，首先读取分区表，拿到其他模块对应的地址，然后根据这些地址初始化不同的模块。

下图是一张工厂程序分区表，

```
# Espressif ESP32 Partition Table
# Name,   Type, SubType, Offset,  Size
nvs,      data, nvs,     0x9000,  0x6000
phy_init, data, phy,     0xf000,  0x1000
factory,  app,  factory, 0x10000, 1M
```

分区表中的每一列都有名字，类型（app，data或者其他），子类型，在 flash 中的偏移量（分区的加载地址）和对应的大小。通过名字（Name），类型（Type）和子类型（SubType）就可以读写相应的模块。

- **Name**

  Name 字段可以是任何有意义的名称，但不能超过 16 个字符（之后的内容将被截断）。该字段对 ESP32 并不是特别重要。

- **Type**

  Type 字段可以指定为 app (0) 或者 data (1)，也可以直接使用数字 0-254（或者十六进制 0x00-0xFE）。注意，0x00-0x3F 是预留给 esp-idf 的核心功能，如果应用程序需要保存数据，需要在 0x40-0xFE 内添加一个自定义分区类型。

- **SubType**

  SubType 字段长度为 8 bit，内容与具体 Type 有关，具体可以参考 ESP-IDF SubType。

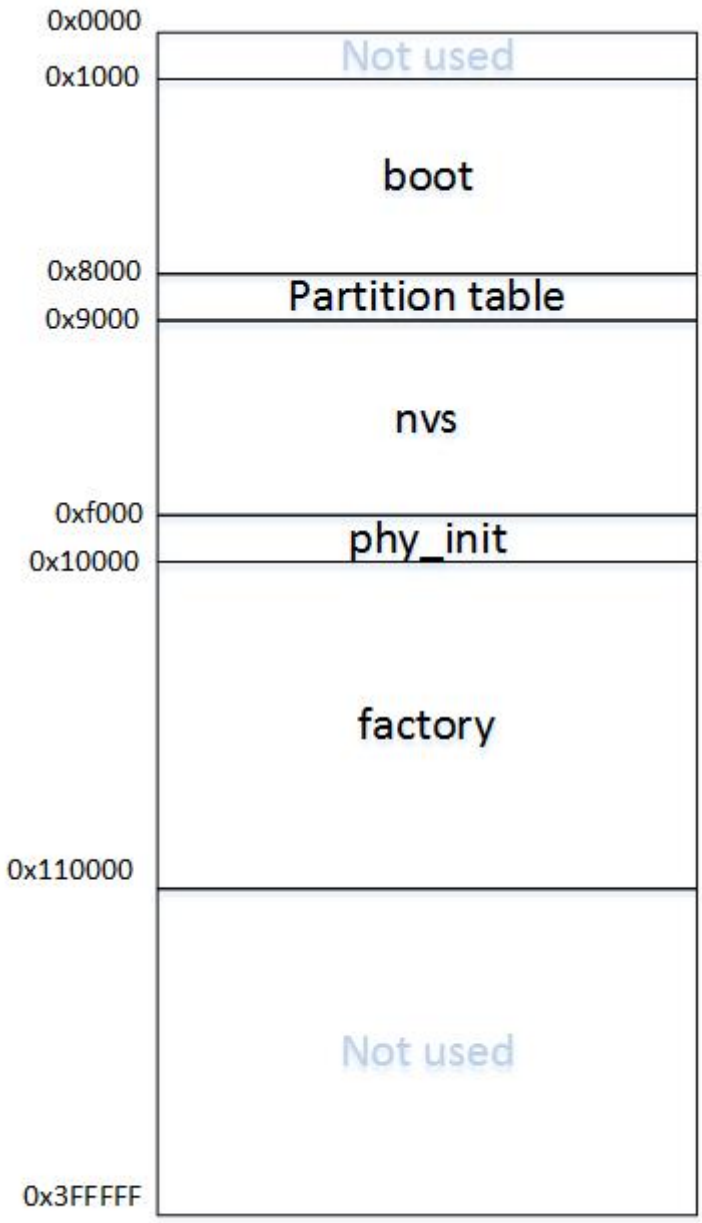
- **offset** & **size**

  分区若为指定偏移地址，则会紧跟着前一个分区之后开始。若此分区为首个分区，则将紧跟着分区表开始。

  对于 ESP32， app 分区的偏移地址必须 64K(0x10000) 对齐，对于 ESP8266，app 分区的偏移地址必须要与 0x1000 (4K) 对齐。

  app 分区的大小和偏移地址可以采用十进制数、以 0x 为前缀的十六进制数，且支持 K 或 M 的倍数单位（分别代表 1024 和 1024*1024 字节）。

分区表对应 ESP32 4M SPI Flash 的分区情况如下图所示：



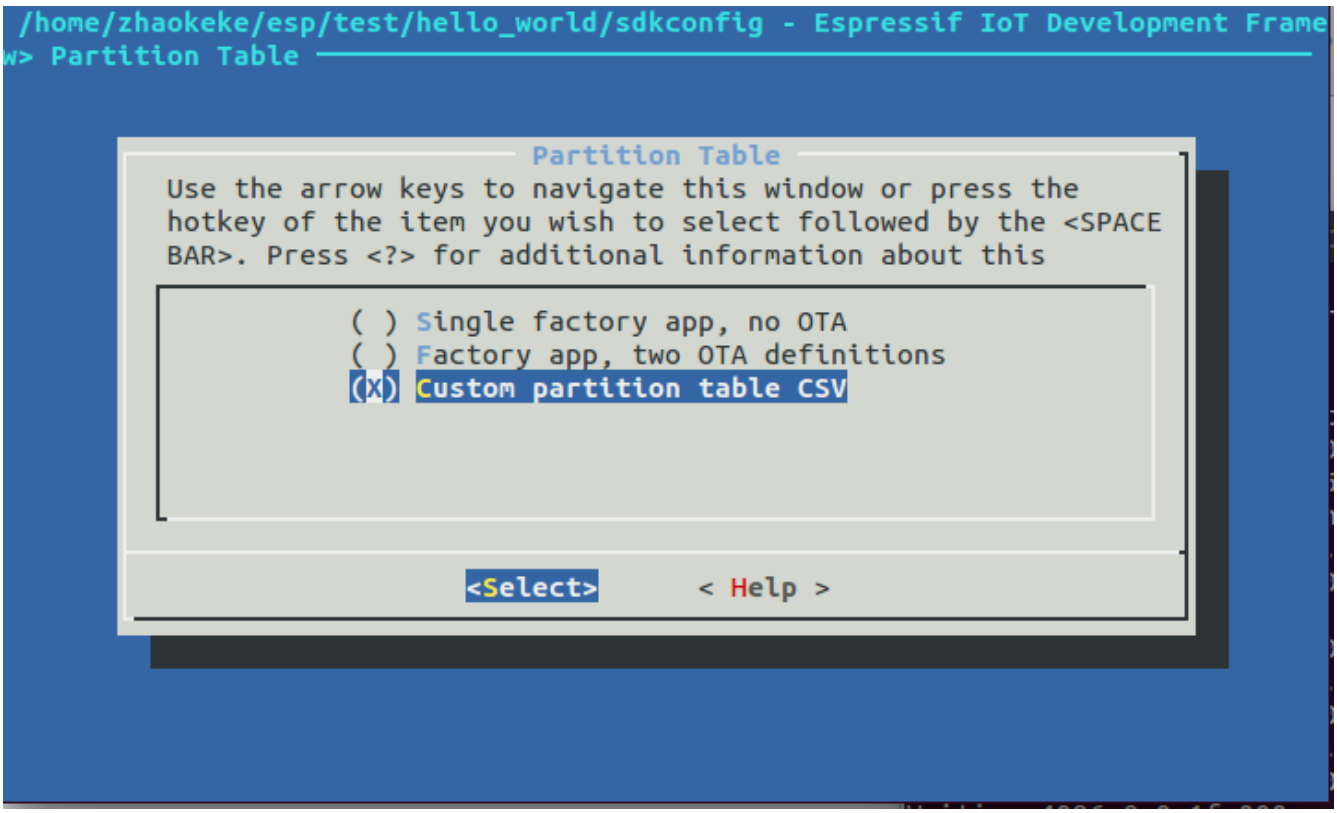
SPI flash 分区示意
（Single factory app, no OTA）

其中：

- boot 的地址是固定的 0x1000（此处指的是 ESP32，**ESP8266 的 boot 地址为固定的 0x0000**），而且 boot 地址的加载早于分区表的加载，因此无需在分区表中表现，大小与 boot 配置项有关，可以在编译完成后查看 `build/bootloader/bootloader.bin` 来确认当前配置项 boot 大小。
- partition table 的地址可以通过配置项（配置项可以通过 `make menuconfig` 打开） `Partition Table` -> `Offset of partition table` 进行修改，其大小固定为 4K（0x1000）。
- nvs 分区的地址可以根据需求任意设置，系统通过类型 `data` 和子类型 `nvs` 来加载。
- phy_init 的地址同样可以任意设置，系统通过类型 `data` 和子类型 `phy` 来加载，大小一般设置为 4k（0x1000），如果在配置项 `Component config` -> `PHY` 中没有使能 `Use a partition to store PHY init data`，则无需此列。
- 偏移地址 0x10000 处存放出厂固件，bootloader 启动时默认加载该偏移地址应用程序。

分区表类型

分区表配置可以通过 `make menuconfig` -> `Partition Table` -> `Partition Table` 列进行配置， 支持的分区表配置有三种：工厂程序（无OTA分区），工厂程序（双 OTA 分区）以及自定义分区表。



如果不知道自己所使用的分区表类型，可以通过 `make partition_table` 命令来查看当前使用分区表的信息摘要。

工厂程序：

以下是不包含 OTA 分区的工厂程序分区表，

```
# Espressif ESP32 Partition Table
# Name,   Type, SubType, Offset,  Size
nvs,      data, nvs,     0x9000,  0x6000
phy_init, data, phy,     0xf000,  0x1000
factory,  app,  factory, 0x10000, 1M
```

通过此分区表可以看出，从 0x110000 ~ 0x3fffff 的 flash 空间没有使用，这极大的浪费了 flash 空间，因此这种分区表一般只用在 demo 中。

工厂程序（双 OTA 分区）：

以下是工厂程序（双 OTA 分区）的分区表：

```
# Espressif ESP32 Partition Table
# Name,   Type, SubType, Offset,  Size
nvs,      data, nvs,     0x9000,  0x4000
otadata,  data, ota,     0xd000,  0x2000
phy_init, data, phy,     0xf000,  0x1000
factory,  0,    0,       0x10000, 1M
ota_0,    0,    ota_0,   ,        1M
ota_1,    0,    ota_1,   ,        1M
```

其中：

- 分区表中定义了三个应用程序分区，这三个分区的类型（type）都被设置为 “app”（即 0 ），但具体 app 类型不同。其中，位于 0x10000 偏移地址处的为出厂应用程序（factory），其余两个为 OTA 应用程序（ota_0，ota_1）。
- 偏移地址为空代表紧挨着上一列的结尾，如 `ota_0` 的偏移地址为 0x10000 + 1M = 0x110000。
- 新增了一个名为 “otadata” 的数据分区，用于保存 OTA 升级时候需要的数据。Bootloader 会查询该分区的数据，以判断该从哪个 OTA 应用程序分区加载程序。如果 `otadata` 分区为空，则会执行出厂程序。

双 OTA 分区和无 OTA 分区的分区表基本上相同，仅仅增加了 OTA 分区可以用来测试 OTA，但是其 APP 的分区列最大只支持 1M 的固件，而 `0x310000` 之后的 flash 分区完全没有使用，因此这种分区表也不适合用在实际程序中。

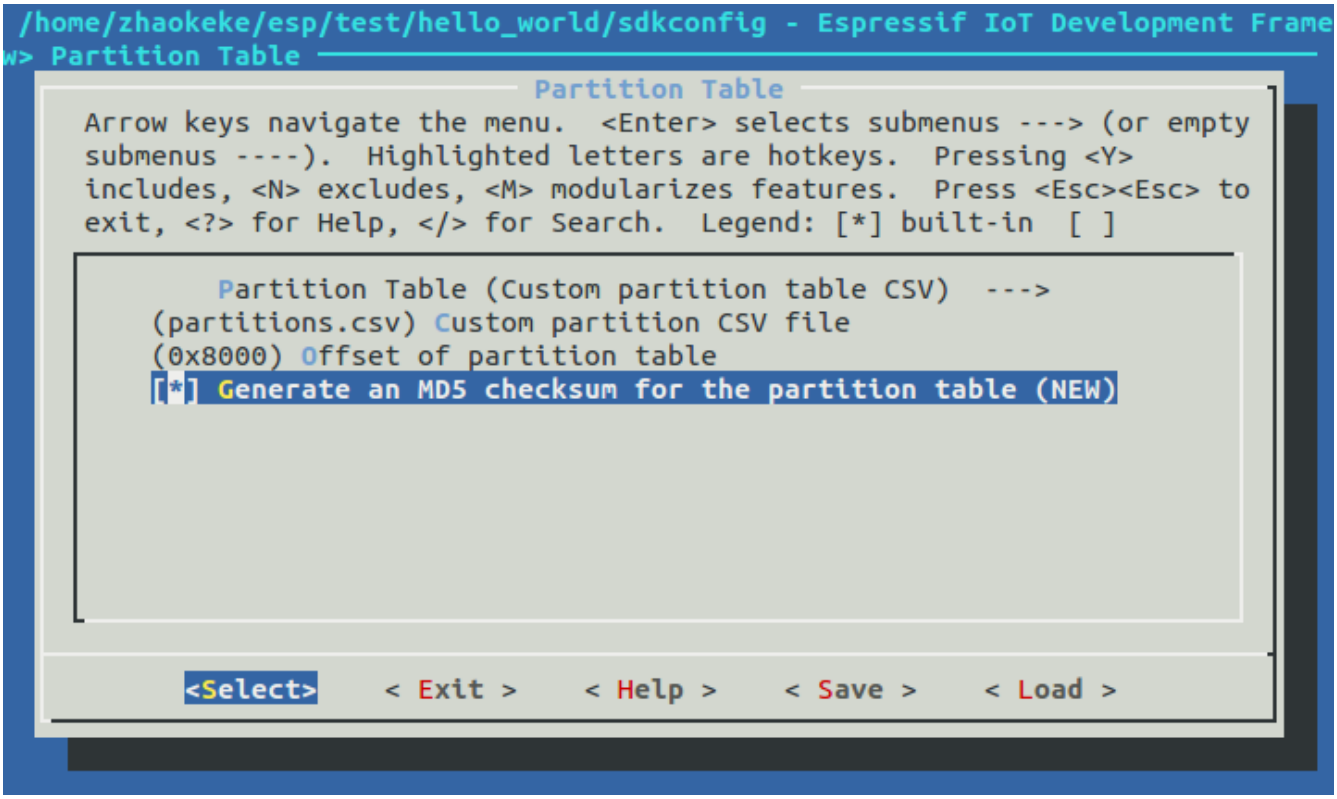
自定义分区表：

自定义分区表是 ESP-IDF 提供的用户自由分配 flash 空间的方式，相对于前两种固化的分区表，更加的灵活，也更容易贴合用户自身习惯。

ESP32 和 ESP8266 的自定义分区表稍有差异：

ESP32:

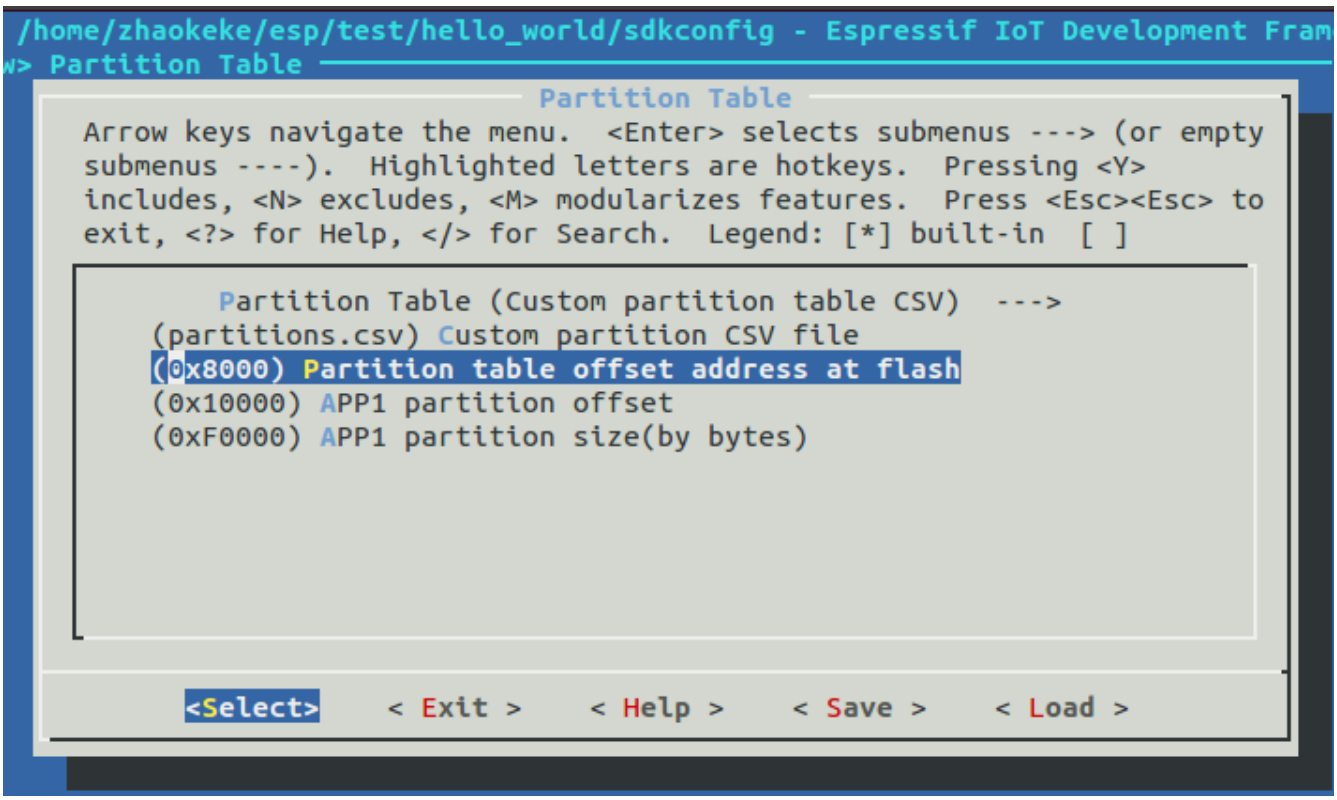
ESP32 选择自定义分区表后，会多出来两列，自定义分区表文件名，以及给分区表生成一个 MD5 签名，如下图所示。



- 自定义分区表默认是当前路径名为 partition.csv 的文件，可以根据自己的需求更改
- 分区表生成 MD5 签名是从 IDF3.1 开始加入的，可以检查分区表是否存在损坏，如果希望跟以前的分区表固件相同，可以取消签名。

ESP8266：

ESP2866 相对于 ESP32 取消了 MD5 签名选项，同时增加了 APP1 的偏移地址和大小，如下图所示



- APP1 偏移地址和大小需要与分区表中的 `ota_0` 对应的偏移地址和大小相同

如何自定义分区表

在选择自定义分区表后，系统默认指定 `partition.csv` 文件作为分区表文件，目录下并没有此文件，因此需要在当前目录下新建一个名为 `partition.csv` 的文件，文件内容即分区方案，以下是一个典型的自定义 ESP32 4M flash 分区表。

```
# Espressif ESP32 Partition Table, partition.csv
# Name,   Type, SubType, Offset,  Size
nvs,      data, nvs,     0x9000,  0x4000
otadata,  data, ota,     0xd000,  0x2000
phy_init, data, phy,     0xf000,  0x1000
ota_0,    0,    ota_0,   0x10000, 0x160000
ota_1,    0,    ota_1,   0x170000,0x160000
custom_data, 0x40,  0,     0x2d0000,0x130000
```

- 在此分区表中，删除了 `factory` 列，因为实际产品使用中一般都需要集成 OTA 功能，而 OTA 是在 `ota_0` 以及 `ota_1` 直接互相升级，系统可以直接跑在 ota 分区，为节省 flash 空间，可以去除 `factory` 列
- `custom_data` 列是自定义分区，将 flash 从偏移地址 `0x2d0000` 开始的 `0x130000` 长度的空间作为用户存储空间。
- 此分区表 ota 分区最大支持 1.375MB （0x160000）的固件，可以通过压缩 `custom_data` 分区的方式（如果不需要使用自定义分区来存储数据，可以直接去掉）来放大 `ota_0` 和 `ota_1` 分区。不过需要注意的是自定义分区时**一定不能**存在地址覆盖问题，比如 `custom_data` 列如果偏移起始地址为 `0x2c0000`，那么与 `ota_1` 的区域重叠。

读写自定义分区内容

代码

以下是一个很简单的读写 partition 的示例，依赖于上一章节中的自定义分区表，此示例读取 Type 为 0x40, SubType 为 0x0 的自定义分区，然后向此分区写入一些数据并读取出来。

```
/* Partition table Example

   This example code is in the Public Domain (or CC0 licensed, at your option.)

   Unless required by applicable law or agreed to in writing, this
   software is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
   CONDITIONS OF ANY KIND, either express or implied.
*/
#include <stdio.h>
#include <string.h>

#include "esp_partition.h"
```

```
int8_t test_custom_partition()
{
    const char* data = "Test read amd write partition";
    uint8_t dest_data[1024] = {0};
    const esp_partition_t *find_partition = NULL;
    find_partition = esp_partition_find_first(0x40, 0x0, NULL);
    if(find_partition == NULL){
        printf("No partition found!\r\n");
        return -1;
    }

    printf("Erase custom partition\r\n");
    if (esp_partition_erase_range(find_partition, 0, 0x1000) != ESP_OK) {
        printf("Erase partition error");
        return -1;
    }

    printf("Write data to custom partition\r\n");
    if (esp_partition_write(find_partition, 0, data, strlen(data) + 1) != ESP_OK) {   //
incude '\0'
        printf("Write partition data error");
        return -1;
    }

    printf("Read data from custom partition\r\n");
    if (esp_partition_read(find_partition, 0, dest_data, 1024) != ESP_OK) {
        printf("Read partition data error");
        return -1;
    }

    printf("Receive data: %s\r\n", (char*)dest_data);

    return 0;
}

void app_main()
{
    test_custom_partition();
}
```

注解

下面将会对里面用到的一些重要函数进行说明。

```
    find_partition = esp_partition_find_first(0x40, 0x0, NULL);
    if(find_partition == NULL){
    printf("No MINVS partition found!\r\n");
    return -1;
    }
```

在分区表中查找第一个满足 `Type` 为 0x40， `SubType` 为 0x0 的分区，第3个参数 `name` 为 NULL 代表不指定，如果找到就会返回分区句柄，否则返回空值。

```
    printf("Erase custom partition\r\n");
    if (esp_partition_erase_range(find_partition, 0, 0x1000) != ESP_OK) {
        printf("Erase partition error");
        return -1;
    }

    printf("Write data to custom partition\r\n");
    if (esp_partition_write(find_partition, 0, data, strlen(data) + 1) != ESP_OK) {   //
incude '\0'
        printf("Write partition data error");
        return -1;
    }
```

在写 flash 之前，需要先将写的区域擦除，擦除的第一个参数为 `esp_partition_find_first` 返回的分区句柄； 第二个参数为此分区的起始地址，地址必须 4K 对其；第三个参数是需要擦除的长度，同样需要 4K 对其。

写分区表函数的第一个参数也是分区句柄，第二个参数为需要写入的分区其实地址，后面两个参数是写入的数据地址以及数据长度。

```
    if (esp_partition_read(find_partition, 0, dest_data, 1024) != ESP_OK) {
        printf("Read partition data error");
        return -1;
    }
```

读分区表与写分区表的函数接口类似，需要指明分区句柄，读取的分区起始地址，数据需要存储的 buffer 指针，以及需要读取的长度。