

# ESP32 中的 分区表



淘宝店铺：

PC 端：

<http://n-xytrt8gqu585po94mwj5atokcyd4.taobao.com/index.htm>

手机端：

[https://shop.m.taobao.com/shop/shop\\_index.htm?sellerId=755668508&shopId=104493595&inShopPageId=423890608&pathInfo=shop/index2](https://shop.m.taobao.com/shop/shop_index.htm?sellerId=755668508&shopId=104493595&inShopPageId=423890608&pathInfo=shop/index2)



资料下载地址：

链接：[https://pan.baidu.com/s/1kCjD8yktZECsGmHomx\\_veg?pwd=q8er](https://pan.baidu.com/s/1kCjD8yktZECsGmHomx_veg?pwd=q8er)

提取码：q8er

源码下载地址：

<https://gitee.com/vi-iot/esp32-board.git>

## 一、esp-idf 默认分区表

本节大部分内容都是摘自乐鑫官方的资料，并以口语化描述出来并进行简化，如果大家想要获得最全面的资料，可以访问如下网站

[https://docs.espressif.com/projects/esp-idf/zh\\_CN/latest/esp32/api-guides/partition-tables.html](https://docs.espressif.com/projects/esp-idf/zh_CN/latest/esp32/api-guides/partition-tables.html)

为什么 ESP32 需要分区表？ESP32 的程序不是一般的程序，会把整包程序划分很多的区间，然后把这些区间的内容烧录到 flash 去，这样对我们大工程也较好管理，功能更清晰。先来看下在 esp-idf 中提供的一个最简单默认分区表有什么内容

打开 esp-idf/components/partition\_table/partitions\_singleapp.csv，内容如下

#	Name,	Type,	SubType,	Offset,	Size,	Flags
#	Note: if you have increased the bootloader siz					
	nvs,	data,	nvs,	,	0x6000,	
	phy_init,	data,	phy,	,	0x1000,	
	factory,	app,	factory,	,	1M,	

#号后面是注释，不参与实际内容，大家主要看标出的那 5 列，列头分别是 Name、Type、SubType、Offset、Size

逐个分析

Name: 表示分区名称（并不重要）

Type: 表示分区类型，可选值有 app、data，用户还可以自定义写 0x40-0xFE

app 一般用于表示运行程序

data 一般用于表示存储数据

SubType: 表示子类型，与类型有关，当 Type 定义为 app 时，SubType 字段可以指定为 factory (0x00)、ota\_0 (0x10) ... ota\_15 (0x1F)；当 Type 定义为 data 时，SubType 字段可以指定为 ota (0x00)、phy (0x01)、nvs (0x02)、nvs\_keys (0x04) 或者其他组件特定的子类型。

Offset: 表示在 Flash 中的偏移地址

Size: 表示分区总大小

现在我们基本知道这个文件的内容框架了，我们现在开始分析具体内容。

### 先看第三行 nvs

nvs 表示非易失存储区，掉电之后数据依然保存。在 ESP32 中预留了一块 NVS 区域，专门用于存储如下内容

- 1) 用于存储每台设备的 PHY 校准数据（注意，并不是 PHY 初始化数据）。
- 2) 用于存储 Wi-Fi 数据（如果使用了 esp\_wifi\_set\_storage(WIFI\_STORAGE\_FLASH) 初始化函数）。
- 3) 其他应用程序数据。

SubType=nvs，表示此分区专门用于 NVS 存储。

大家这里可能有个疑问，为什么 Offset 是空的？到底在 Flash 中的偏移地址是多少？可能 在其他 esp-idf 版本中这里的 Offset 不是空的，但在 v5.2 版本中，这里的确是空的，大家浏览 esp-idf/components/partition\_table/文件中其他分区表文件，大部分 Offset 也是空的。其实这里并非没有指定地址，在 esp-idf 中，还默认有个 bootloader 程序，这个程序在分区表中的地址是 0x1000，另外还有分区表本身也需要存储，分区表在 flash 中存储的地址是

0x8000，大小是 0x1000，也就是除开 bootloader 和分区表本身，可用的地址是从 0x9000 开始的，也就是说，esp-idf 中 nvs 分区其实地址是从 0x9000 开始存储，然后 Size=0x6000。

#### 再看第四行 phy\_init

此分区子类型是 phy，表示用于分区用于存放 PHY 初始化数据，从而保证可以为每个设备单独配置 PHY，默认配置下，phy 分区并不启用，而是直接将 phy 初始化数据编译至应用程序中，从而节省分区表空间（直接将此分区删掉）而非必须采用固件中的统一 PHY 初始化数据，我们在应用开发中一般用不到这个分区。

这里 Offset 也是空的，因为如果我们留空，esp-idf 就会自动帮我们计算，在上一个分区 Offset+Size 后，就会作为这个分区起始地址，本例中起始地址就是 0x9000+0x6000=0xF000，大小 Size = 0x1000

#### 再看第五行 factory

此分区子类型是 factory，是默认的 app 分区，上电后程序会从这里启动，但如果存在 Subtype=ota 的分区，bootloader 会检查 ota 分区里面的内容，再决定用哪个 app 分区里面的程序启动。app 分区的偏移地址必须与 0x10000 (64 KB) 对齐，本例中 Offset=0xF000+0x1000=0x10000，大小是 1M（esp-idf 中支持这种 1M、1K 的表示形式）

接下来我们来看下一个支持 OTA 的分区表有什么不同

路径：esp-idf/components/partition\_table/partitions\_two\_ota.csv

```
# Name, Type, SubType, Offset, Size, flags
# Note: if you have increased the bootloader, increase the bootloaders_offset accordingly
nvs, data, nvs, , 0x4000,
otadata, data, ota, , 0x2000,
phy_init, data, phy, , 0x1000,
factory, app, factory, , 1M,
ota_0, app, ota_0, , 1M,
ota_1, app, ota_1, , 1M,
```

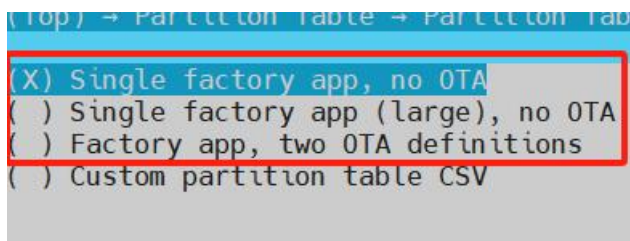
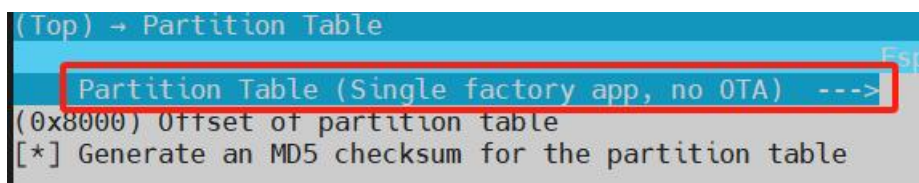
部分和刚才介绍的分区一致的就不过多介绍了。主要关注新增这几个 otadata、ota\_0、ota\_1

otadata, Subtype = ota,用于存储当前所选的 OTA 应用程序的信息。这个分区的大小需要设定为 0x2000

ota\_0, Subtype=ota\_0, 为 OTA 应用程序分区，启动加载器将根据 OTA 数据分区中的数据来决定加载哪个 OTA 应用程序分区中的程序。在使用 OTA 功能时，应用程序应至少拥有 2 个 OTA 应用程序分区（ota\_0 和 ota\_1）

ota\_1, 和 ota\_0 分区一样。

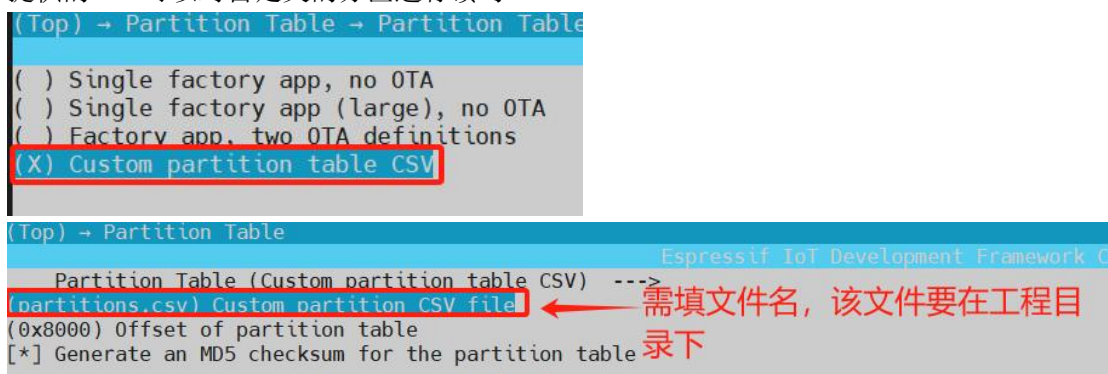
那么以上介绍的两个分区表，都是 esp-idf 中默认支持的，我们可以在 esp 工程中使用 idf.py menuconfig 配置命令调出配置菜单，如下图



第一和第三个分区表就是刚才介绍的 single app 和 ota 类型的分区表。至于第二个和第一个类似，不同点在于 app 区域的 size 会设定为 1.5M

## 二、自定义分区

对于简单的应用，我们只用上节提到的 esp-idf 中提供的分区表即可，但如果我们有其他东西想要存到 flash 中，又不想用 nvs 区域，那我们可以自定义一个分区，然后使用 esp-idf 中提供的 API 可以对自定义的分区进行读写。

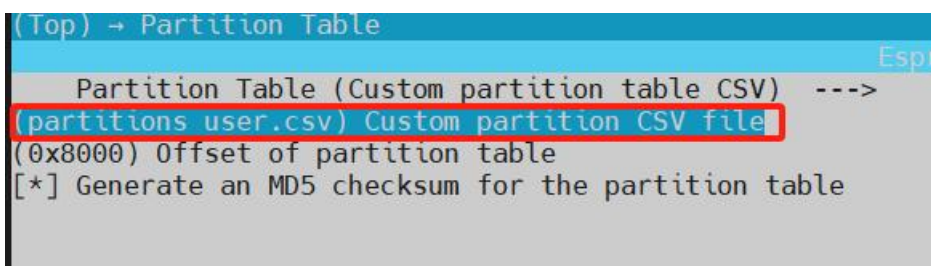


通过如上图选项，可以启用自定义分区表文件

由上节可知，Type 除了使用 app 和 data 外，还可以使用自定义类型 0x40-0xFE，这些类型我们可以用来做其他存储用途，现在我们新建一个自定义的 partitions\_user.csv 文件内容如下

```
partitions_user.csv
1  # Name,   Type, SubType, Offset,  Size, Flags
2  # Note: if you have increased the bootloader size,
3  nvs,      data, nvs,      ,        0x6000,
4  phy_init, data, phy,      ,        0x1000,
5  factory,  app,  factory, ,        1M,
6  user,     0x40, 0x01, ,        0x1000,
```

然后通过 idf.py menuconfig 中指定使用的 CSV 文件



我们在 partitions\_singleapp.csv 的基础上追加了一行我们自定义的分区类型 0x40，子类型是 0x01，因为我们指定了分区类型是自定义类型，所以子类型这里 esp 编译系统不会太关注这个值，我们只需要记得这个子类型是 0x01 即可，偏移 Offset 自动计算，大小 Size 为 0x1000。接下来请看例程

具体例程源码位于 esp32-board/partition

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_partition.h"
#include <string.h>

static const char* TAG = "main";
#define USER_PARTITION_TYPE    0x40        //自定义的分区类型
#define USER_PARTITION_SUBTYPE 0x01        //自定义的分区子类型
//分区指针
static const esp_partition_t* partition_res=NULL;
//读取缓存
static char g_esp_buf[1024];
void app_main(void)
{
    //找到自定义分区，返回分区指针，后续用到这个指针进行各种操作
    partition_res=esp_partition_find_first(USER_PARTITION_TYPE,USER_PARTITION_SUBTYPE,NULL);
    if(partition_res == NULL)
```



```

{
    ESP_LOGI(TAG,"Can't find partition,return");
    return;
}
//擦除
ESP_ERROR_CHECK(esp_partition_erase_range(partition_res,0x0,0x1000))
;

//测试字符串
const char* test_str = "this is for test string";
//从分区偏移位置 0x0 写入字符串
ESP_ERROR_CHECK(esp_partition_write(partition_res,0x00, test_str,
strlen(test_str)));
//从分区偏移位置 0x0 读取字符串
ESP_ERROR_CHECK(esp_partition_read(partition_res,0x00, g_esp_buf,
strlen(test_str)));
ESP_LOGI(TAG,"Read partition str:%s",g_esp_buf);
while(1)
{
    vTaskDelay(pdMS_TO_TICKS(1000));
}
}

```

本例程有四个关键的函数

**esp\_partition\_find\_first** 从分区中找到我们自定义的分区，第一个参数是类型 0x40，第二个参数是子类型 0x01，返回值是分区指针，后续的操作都是基于这个指针进行。

**esp\_partition\_erase\_range** 擦除一段内容，第一个参数是分区指针，第二个参数是偏移地址，第三个参数是大小。偏移地址和擦除大小需 0x1000 对齐（4096 的整数倍）。

**esp\_partition\_write** 写入数据，第一个参数是分区指针，第二个参数是偏移地址，第三个参数是写入数据，第四个参数是写入的数据内容，**需要注意的是，写入之前，需要执行擦除，否则写入无效**

**esp\_partition\_read** 读取数据，参数与写入类似

整个例程比较简单，演示了一下读写操作，大家可以自行的进行测试。