使用 PROGMEM 节省 ESP8266EX RAM 空间

版本 1.0

关于本手册

本文介绍了在 Arduino IDE 的平台下如何使用 PROGMEM 来节省 ESP8266EX 的 RAM 空间。PROGMEM 可以用来将不变的、只读的数据存入 Flash 中。

章节	标题	内容
第 1 章	概述	简要介绍 ESP8266EX 和 PROGMEM。
第 2 章	PROGMEM 使用指南	详细介绍怎样使用 PROGMEM 将数据存入 Flash。
第 3 章	总结	使用 PROGMEM 的总结。

发布说明

日期	版本	发布说明
2017.03	V1.0	首次发布

目录

1. 概述

1.1. ESP8266EX 简介

乐鑫的 ESP8266EX 芯片内置了存储控制器，包含 ROM 和 SRAM。ESP8266EX 的 RAM 空间总共 160 KB，其中 IRAM 为 64 KB，DRAM 为 96 KB。

芯片内部无可编程存储器，用户程序必须由外部 Flash 存储。

ESP8266EX 使用外置 SPI Flash 存储用户程序，最大可支持 16 MB 的存储。

> 📖 **说明：**
>
> 更多关于 *ESP8266EX* 的硬件信息，请查看 *ESP8266EX 技术规格表*。

1.2. PROGMEM 简介

AVR 和其他类似的 8-bit 单片机上有多种存储空间可供使用，包括 Flash、SRAM 和 EEPROM 等。

`PROGMEM` 关键字是一个变量修饰符，常见于使用 AVR-GCC 编译器进行构建的开发环境中，包括Arduino IDE。

通常数据会保存在 SRAM 中，但是使用 `PROGMEM` 可以实现将数据存储在 Flash 而不是 SRAM 中。 `PROGMEM` 应该仅与 ***pgmspace.h*** 中定义的数据类型一起使用。

`PROGMEM` 不应该与 `const` 关键字混淆，`const` 关键字只是在程序执行过程中通知编译器 `const` 数据不会改变。`const` 关键字将数据标识为只读，但不指定数据的存储位置。将数据声明为 `const` 可能只会提高执行速度或识别对只读数据的意外写入。

> 📖 **说明：**
>
> 更多关于 `PROGMEM` 的信息可通过点击 *https://www.arduino.cc/en/Reference/PROGMEM* 链接进行查看。

对于 ESP8266EX 而言，使用 `PROGMEM` 宏很简单，如下所示：

```
#define PROGMEM   ICACHE_RODATA_ATTR
```

它是由下面的函数定义的：

```
#define ICACHE_RODATA_ATTR  __attribute__((section(".irom.text")))
```

此函数将变量放在 `.irom.text` 字段，即 Flash 中。

理解 `PROGMEM` 的关键是了解字符串的存储方式，以及如何从 Flash 中检索它们。下文将进行详细描述。

2. PROGMEM 使用指南

第 2 章和第 3 章的内容改编自 *Guide to PROGMEM on ESP8266 and Arduino IDE*，原作者为 sticilface。

2.1. 将字符串存储在 Flash 中

2.1.1. 声明存储在 Flash 中的全局字符串

```
static const char xyz[] PROGMEM = "This is a string stored in flash";
```

2.1.2. 在代码块中声明一个 flash 字符串

可以使用 PSTR 宏，PSTR 宏在 ***pgmspace.h*** 中定义：

```
#define PGM_P       const char *
#define PGM_VOID_P  const void *
#define PSTR(s) (__extension__({static const char __c[] PROGMEM = (s); &__c[0];}))
```

在实际使用中：

```
void myfunction(void) {
PGM_P xyz = PSTR("Store this string in flash");
const char * abc = PSTR("Also Store this string in flash");
}
```

对于存储在 flash 中的数据的检索和操作不同于 8-bit 处理器，这是因为 ESP8266EX 以 4 字节对齐的方式读取 Flash。*Arduino IDE for ESP8266* 中介绍了几个函数，可以检索那些已经通过 PROGMEM 存储在 Flash 中的字符串。上述两个例子都会返回 `const char *`，但是这些指针不能随意使用，也不能转换为另一种类型，因为可能导致段错误，并导致 ESP8266EX 异常崩溃。因此，读取 Flash 中的数据必须确保 4 字节对齐。

2.2. 从 PROGMEM 读回函数

从 PROGMEM 中读回的函数都在 ***pgmspace.h*** 中定义：

```
int memcmp_P(const void* buf1, PGM_VOID_P buf2P, size_t size);
void* memccpy_P(void* dest, PGM_VOID_P src, int c, size_t count);
void* memmem_P(const void* buf, size_t bufSize, PGM_VOID_P findP, size_t findPSize);
void* memcpy_P(void* dest, PGM_VOID_P src, size_t count);
char* strncpy_P(char* dest, PGM_P src, size_t size);
#define strcpy_P(dest, src)       strncpy_P((dest), (src), SIZE_IRRELEVANT)
char* strncat_P(char* dest, PGM_P src, size_t size);
#define strcat_P(dest, src)       strncat_P((dest), (src), SIZE_IRRELEVANT)
```

```
int strncmp_P(const char* str1, PGM_P str2P, size_t size);
#define strcmp_P(str1, str2P)     strncmp_P((str1), (str2P), SIZE_IRRELEVANT)s
int strncasecmp_P(const char* str1, PGM_P str2P, size_t size);
#define strcasecmp_P(str1, str2P)  strncasecmp_P((str1), (str2P), SIZE_IRRELEVANT)
size_t strnlen_P(PGM_P s, size_t size);
#define strlen_P(strP)             strnlen_P((strP), SIZE_IRRELEVANT)
char* strstr_P(const char* haystack, PGM_P needle);
int printf_P(PGM_P formatP, ...);
int sprintf_P(char *str, PGM_P formatP, ...);
int snprintf_P(char *str, size_t strSize, PGM_P formatP, ...);
int vsnprintf_P(char *str, size_t strSize, PGM_P formatP, va_list ap);
```

这里有很多函数，但实际上它们是 _P 版本的标准 C 函数，适用于从 ESP8266EX 的四字节对齐的 Flash 中读取。它们都有 PGM_P，本质上是一个 const char *。在底层这些函数都使用下面的指令读回字节，而不会引起段错误。

```
#define pgm_read_byte(addr)                                                    \
(__extension__({                                                               \
    PGM_P __local = (PGM_P)(addr);  /* isolate varible for macro expansion */  \
    ptrdiff_t __offset = ((uint32_t)__local & 0x00000003); /* byte aligned mask */  \
    const uint32_t* __addr32 = (const uint32_t*)((const uint8_t*)(__local)-__offset);  \
    uint8_t __result = ((*__addr32) >> (__offset * 8));                        \
    __result;                                                                  \
}))
```

如果用户设计了一个专门用于处理 PROGMEM 指针的函数，但除了 const char * 之外没有类型检查时，这个函数很有效。这意味着，就编译器而言，向其传递任何 const char * 字符串是完全合法的。然而这显然不是真的，这将导致未定义的行为。当它们被定义为 PGM_P 时，不可能创建任何可以使用 flash 字符串的重载函数。如果用户去尝试，就会得到一个不明确的重载错误，比如 PGM_P == const char *。

2.3. __FlashStringHelper 包装类

这是一个包装类，允许 flash 字符串用作类，这意味着类型检查和函数重载可以与 flash 字符串一起使用。大多数用户很熟悉 F() 宏和 FPSTR() 宏，这些宏在 ***WString.h*** 中定义：

```
#define FPSTR(pstr_pointer) (reinterpret_cast<const __FlashStringHelper *>(pstr_pointer))
#define F(string_literal) (FPSTR(PSTR(string_literal)))
```

FSPTR() 获取一个指向字符串的 PROGMEM 指针，并将其转换为 __FlashStringHelper 类。因此如果用户已经定义了一个字符串（如上 xyz），则可以使用 FPSTR() 将其转换为 __FlashStringHelper 并传递给接受它的函数。

```
static const char xyz[] PROGMEM = "This is a string stored in flash";
Serial.println(FPSTR(xyz));
```

F() 结合了这两种方法来创建一个简单快捷的方法将内联字符串存储在 Flash 中，并返回类型 __FlashStringHelper。例如：

```
Serial.println(F("This is a string stored in flash"));
```

虽然这两个函数提供了类似的功能，但它们分饰不同的角色。FPSTR() 允许定义一个全局 flash 字符串，然后在任何使用 __FlashStringHelper 的函数中使用它。F() 允许定义这些 flash 字符串，但是不能在其他地方使用它们。其结果是可以使用 FPSTR() 而不能使用 F() 共享公共字符串。String 类就是使用 __FlashStringHelper 来重载其构造函数的：

```
String(const char *cstr = "");              // constructor from const char *
String(const String &str);                  // copy constructor
String(const __FlashStringHelper *str);     // constructor for flash strings
```

然后用户可以写：

```
String mystring(F("This string is stored in flash"));
```

2.4. 如何写一个函数来使用 __FlashStringHelper

将指针转换回 PGM_P 并使用上文所述的 _P 函数。以下是一个用于 concat 函数的字符串的示例：

```
unsigned char String::concat(const __FlashStringHelper * str) {
    if (!str) return 0;                   // return if the pointer is void
    int length = strlen_P((PGM_P)str);    // cast it to PGM_P, which is basically const char
*, and measure it using the _P version of strlen.
    if (length == 0) return 1;
    unsigned int newlen = len + length;
    if (!reserve(newlen)) return 0;       // create a buffer of the correct length
    strcpy_P(buffer + len, (PGM_P)str);  //copy the string in using strcpy_P
    len = newlen;
    return 1;
}
```

2.5. 如何声明并使用一个全局 flash 字符串

```
static const char xyz[] PROGMEM = "This is a string stored in flash. Len = %u";


void setup() {
    Serial.begin(115200); Serial.println();
    Serial.println( FPSTR(xyz) );             // just prints the string, must convert it to
FlashStringHelper first using FPSTR().
    Serial.printf_P( xyz, strlen_P(xyz));     // use printf with PROGMEM string
}
```

2.6. 如何使用内联 flash 字符串

```
void setup() {
    Serial.begin(115200); Serial.println();
    Serial.println( F("This is an inline string")); //
    Serial.printf_P( PSTR("This is an inline string using printf %s"), "hello");
}
```

2.7. 如何声明并使用 PROGMEM 里的数据

```
const size_t len_xyz = 30;
const uint8_t xyz[] PROGMEM = {
  0x53, 0x61, 0x79, 0x20, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20,
  0x74, 0x6f, 0x20, 0x4d, 0x79, 0x20, 0x4c, 0x69, 0x74, 0x74,
  0x6c, 0x65, 0x20, 0x46, 0x72, 0x69, 0x65, 0x6e, 0x64, 0x00};

 void setup() {
     Serial.begin(115200); Serial.println();
     uint8_t * buf = new uint8_t[len_xyz];
     if (buf) {
      memcpy_P(buf, xyz, len_xyz);
      Serial.write(buf, len_xyz); // output the buffer.
     }
 }
```

2.8. 如何声明 PROGMEM 里的一部分数据，并从中检索一个字节

像上面那样声明数据，然后使用 pgm_read_byte 来获取数值：

```
const size_t len_xyz = 30;
const uint8_t xyz[] PROGMEM = {
  0x53, 0x61, 0x79, 0x20, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20,
  0x74, 0x6f, 0x20, 0x4d, 0x79, 0x20, 0x4c, 0x69, 0x74, 0x74,
  0x6c, 0x65, 0x20, 0x46, 0x72, 0x69, 0x65, 0x6e, 0x64, 0x00};

 void setup() {
     Serial.begin(115200); Serial.println();
     uint8_t * buf = new uint8_t[len_xyz];
     if (buf) {
      memcpy_P(buf, xyz, len_xyz);
      Serial.write(buf, len_xyz); // output the buffer.
     }
 }
```

3. 总结

使用 PROGMEM 和 PSTR 很容易将字符串存储在 Flash 中，但是由于它们基本上是 const char *，所以必须创建专门使用它们生成的指针的函数。另外，FPSTR 和 F() 提供一个类，帮助用户做隐式转换，这在重载函数和隐式类型转换时非常有用。值得注意的是，如果要存储一个 int、float 或指针，这些都可以直接存储和读回，因为它们是四字节对齐的，并因此将始终四字节对齐。

乐鑫 IOT 团队

www.espressif.com