Save ESP8266EX RAM with PROGMEM

Version 1.0

About This Guide

The document outlines how `PROGMEM` may be used to save RAM on the ESP8266EX when using the Arduino IDE. `PROGMEM` is used to store constant, read-only data on flash memory.

Chapter	Title	Content
Chapter 1	Overview	A brief introduction to ESP8266EX and `PROGMEM`.
Chapter 2	`PROGMEM` User Guide	How to use `PROGMEM` to store data to flash.
Chapter 3	Summary	A summary of the utilization of `PROGMEM`.

Release Notes

Date	Version	Release notes
2017.03	V1.0	First release.

Table of Contents

1. Overview 1
    1.1. ESP8266EX 1
    1.2. PROGMEM 1
2. PROGMEM User Guide 3
    2.1. Store Strings in Flash 3
        2.1.1. Declare a Global String to be Stored in Flash 3
        2.1.2. Declare a Flash String Within Code Block 3
    2.2. Functions to Read Back from PROGMEM 3
    2.3. __FlashStringHelper Wrapper Class 4
    2.4. Write a Function to Use __FlashStringHelper 5
    2.5. Declare and Use a Global Flash String 5
    2.6. Use Inline Flash Strings 6
    2.7. Declare and Use Data in PROGMEM 6
    2.8. Retrieve Single Byte from PROGRAM Data 6
3. Summary 7

1. Overview

1.1. ESP8266EX

ESP8266EX Wi-Fi SoC integrates memory controller and memory units including SRAM and ROM. The size of the RAM memory is 160 KB in total, among which 64 KB is IRAM and 96 KB is DRAM.

There is no programmable ROM in the SoC, therefore, user program must be stored in an external SPI flash memory.

ESP8266EX uses external SPI flash to store user programs, and supports up to 16 MB memory capacity.

📖 ***Note:***

For more information on ESP8266EX, please refer to ESP8266EX Datasheet.

1.2. PROGMEM

There are various types of memory available at different address offsets in an AVR and equivalent 8-bit microcontrollers, including flash, SRAM and EEPROM.

The `PROGMEM` keyword is a variable modifier most often seen in development environments that are built over or use the AVR-GCC compiler, including Arduino IDE.

`PROGMEM` instructs the compiler to "put this information into flash memory" instead of putting it into the SRAM, where it would normally go. `PROGMEM` should generally be used only with the datatypes defined in ***pgmspace.h***.

`PROGMEM` should not to be confused with the `const` keyword, which simply notifies the compiler that `const` data will not change during program execution. The `const` keyword identifies data as read-only and by definition, does not specify where the data is to be stored. Declaring data as `const` may only result in execution speed improvement or help identify accidental writes to read-only data.

📖 ***Note:***

For more information on PROGMEM, please refer to https://www.arduino.cc/en/Reference/PROGMEM.

In case of the ESP8266EX, which also features flash memory and internal SRAM, the equivalent macro for `PROGMEM` is simply:

```
#define PROGMEM   ICACHE_RODATA_ATTR
```

Which in turn is defined by:

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

Which places the variable in the `.irom.text` section, i.e., program flash memory.

The key to understanding `PROGMEM` is to understand how the strings are stored and then how they are retrieved from flash. The details are provided in the following chapter.

2. PROGMEM User Guide

Chapter 2 and Chapter 3 are adapted from Guide to PROGMEM on ESP8266 and Arduino IDE, which is written by sticilface.

2.1. Store Strings in Flash

2.1.1. Declare a Global String to be Stored in Flash

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

2.1.2. Declare a Flash String Within Code Block

For this you can use the PSTR macro. Which are all defined in ***pgmspace.h***.

```
#define PGM_P       const char *

#define PGM_VOID_P  const void *

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

In practice:

```
void myfunction(void) {

PGM_P xyz = PSTR("Store this string in flash");

const char * abc = PSTR("Also Store this string in flash");

}
```

Retrieval and manipulation of the data stored in flash is not similar to 8-bit processors because the ESP8266EX must access the flash in 4-byte words. In the Arduino IDE for ESP8266EX, there are several functions that can help retrieve strings from flash that have been stored using PROGMEM. Both of the examples above will produce a `const char *` back, however these pointers may not be used freely or cast to another type as it may cause a segmentation fault and cause the ESP8266EX to crash with an exception. Data from the flash must be accessed after making sure that the read address is 4-byte word-aligned.

2.2. Functions to Read Back from PROGMEM

Which are all defined in ***pgmspace.h***.

```
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);
```

There are a lot of functions there but in reality they are `_P` versions of standard c functions that are adapted to read from the ESP8266EX's 32-bit aligned flash. All of them take a `PGM_P` which is essentially a `const char *`. Under the hood these functions all use:

```
#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;                                                                   \
}))
```

which reads back the bytes without causing a segmentation fault.

This works well when you have designed a function as above that is specialized for dealing with `PROGMEM` pointers but there is no type checking except against `const char *`. This means that it is totally legitimate, as far as the compiler is concerned, for you to pass it any `const char *` string, which is obviously not true and will lead to undefined behavior. This makes it impossible to create any overloaded functions that can use flash strings when they are defined as `PGM_P`. If you try you will get an ambiguous overload error as `PGM_P == const char *`.

2.3. `__FlashStringHelper` Wrapper Class

This is a wrapper class that allows flash strings to be used as a class, this means that type checking and function overloading can be used with flash strings. Most people will be familiar with the `F()` macro and possibly the `FPSTR()` macro. These are defined in ***WString.h***:

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

`FSPTR()` gets the `PROGMEM` pointer referring to a string and casts it to the `__FlashStringHelper` class. If you have a flash string (such as `xyz` above), you can use `FPSTR()` to convert it to `__FlashStringHelper` for passing to functions that accept it.

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

The F() combines both of these methods to create an easy and quick way to store an inline string in flash, and return the type __FlashStringHelper. For example:

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

Although these two functions provide a similar function, they serve different roles. FPSTR() allows you to define a global flash string and then use it in any function that takes __FlashStringHelper. F() allows you to define these flash strings in place, but you can't use them anywhere else. The consequence of this is sharing common strings is possible using FPSTR() but not when using F(). __FlashStringHelper is what the String class uses to overload its constructor:

```
String(const char *cstr = "");              // constructor from const char *

String(const String &str);                  // copy constructor

String(const __FlashStringHelper *str);     // constructor for flash strings
```

This allows you to write:

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

2.4. Write a Function to Use __FlashStringHelper

Cast the pointer back to a PGM_P and use the _P functions shown above. This an example implementation for String for the concat function.

```
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. Declare and Use a Global Flash String

```
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. Use Inline Flash Strings

```
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. Declare and Use Data in PROGMEM

```
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. Retrieve Single Byte from PROGRAM Data

Declare the data as done previously, then use `pgm_read_byte` to get the value back.

```
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. Summary

It is easy to store strings in flash using `PROGMEM` and `PSTR` but you have to create functions that specifically use the pointers they generate as they are basically `const char *`. On the other hand `FPSTR` and `F()` give you a class that you can do implicit conversions from, very useful when overloading functions, and doing implicit type conversions. It is worth adding that if you wish to store an `int`, `float` or pointer these can be stored and read back directly as they are 4 bytes in size and therefore will be always aligned.