GCC and the PROGMEM Attribute

Dean Camera

January 15, 2015

****

Contents

Chapter 1

Introduction

The PROGMEM attribute is always a source of confusion for those beginning with AVR-GCC. The PROGMEM attribute is a powerful one and holds the potential to save a lot of RAM, which is something of a limited commodity on many AVRs. Before you can use the PROGMEM attribute, you must first understand what it does and why it is useful.

When strings are used in a program, they are commonly "hard-coded" into the firmware source code:

```
LCD_puts("Hardcoded String");
```

While this seems the most logical way of using strings, it is *not* the most optimal. Common sense and intuition would dictate that the compiler would store the string in the program memory, and read it out byte-by-byte inside the `LCD_puts()` routine. But this is not what happens.

Because the `LCD_puts()` routine (or other string routines) are designed to work with strings in RAM, the compiler is forced to read out the entire string constant from program memory into RAM, and then pass the string's RAM pointer to the routine (in this case `LCD_puts()`). It certainly works but the RAM wastage adds up to significant amounts with each string. Why?

Initial variable values and strings are copied out from program memory into RAM as part of the C startup routines, which execute before your `main()` function. Those startup routines give your globals their initial values, as well as ensure that all strings are inside RAM so they can be passed to your desired string handling routines. As more strings are added to your program, more data must be copied to RAM at startup and the more RAM is used up by holding static (unchanging) data.

The solution to the problem is forcing strings to stay in program memory and only be read out as they are needed. This is not a simple task and requires string routines specifically designed to handle strings kept solely inside the program memory space.

Using the `const` modifier on your string variables is a natural solution — but a wrong one. A `const` variable can only not be modified by the program code, but it does not explicitly prevent the variable value from being copied out to the AVRs RAM on startup. Some compilers implicitly do this for you, but GCC needs to be explicitly told that the contents must live in program memory for the entire duration of the program's execution.

Chapter 2

Basic PROGMEM Use

The *avr-libc* library contains a header file, `<avr/pgmspace.h>`, which contains all the interfacing information needed to allow you to specify data which is to be kept inside the AVR's flash memory. To use the `pgmspace.h` functions, you need to include the header at the start of your C file(s):

```
#include <avr/pgmspace.h>
```

To force a string into program memory, we can now use the `PROGMEM` attribute modifier on our string constants. An example of a global string which is stored into program memory and not copied out at execution time is:

```
char FlashString[] PROGMEM = "This is a string held completely in flash memory.";
```

In older AVR-GCC versions, variables tagged with the `PROGMEM` attribute don't have to also explicitly be marked as `const`, however this is changing in newer releases. To prevent compile time warnings, we add this to the variable's declaration:

```
const char FlashString[] PROGMEM = "This is a string held completely in flash memory.";
```

Now that the `PROGMEM` string has a `const` modifier, we cannot try to modify it in our code. The `pgmspace.h` header also exposes a neat little macro, `PSTR()`, which by some GCC magic allows you to create inline strings that are kept in FLASH memory:

```
LCD_puts(PSTR("Program Memory String"));
```

This stops you from having to clutter your program up with hundreds of variables which hold one-time-used strings. The downside to using the PSTR macro rather than a `PROGMEM` modified string variable is that you can only use it within functions - you can't use it as a global variable initializer. Also be aware that the compiler treats all `PSTR()` declared strings as being completely separate to one-another, so declaring the same string contents in multiple places will waste extra program FLASH memory. If you find yourself using the same string multiple times in your application, use a globally declared `PROGMEM` string instead.

However, we now have a problem — now all your code doesn't work! What's wrong?

The reason behind this is that your string functions are still expecting the source strings to be held inside RAM. When you pass a string to a routine, you are in fact just passing a pointer (a memory address) to the start of the string located somewhere in RAM. The string handling routine then just loads the bytes of the string one-by-one, starting from the pointer's location. But our `PROGMEM` strings are not in RAM, so the pointer to them is invalid.

A pointer to a string stored in `PROGMEM` returns the address in flash memory at which the string is stored. It's still a valid pointer, it's just pointing to a different memory space. To use `PROGMEM` strings in our application, we need to make our string routines `PROGMEM`-pointer aware.

Again, `pgmspace.h` comes to our rescue. It contains several functions and macros which deal with `PROGMEM`-based strings. First, you have all your standard string routines (`memcpy()`, `strcmp()`, etc.) with a "`_P`" postfix denoting that the function deals with the FLASH memory space. For example, to compare a RAM-based string with a `PROGMEM`-based string, you would use the `strcmp_P()` function:

```
strcmp_P("RAM STRING", PSTR("FLASH STRING"));
```

For a full list of the available string functions, check out the *avr-libc* documentation which is installed with your WinAVR or Atmel Toolchain installation.

But what if you have your own string function, like a USART-transmitting routine? Lets look at a typical example:

```
void USART_TxString(const char *data)
{
   while (*data != '\0')
     USART_Tx(*data++);
}
```

This relies on the routine `USART_Tx()`, which for the purposes of this example we will assume to be predefined as a function which transmits a single passed character through the AVR's USART. Now, how do we change our routine to use a `PROGMEM` string?

The `pgmspace.h` header exposes a macro which is important for `PROGMEM`-aware routines; `pgm_read_byte()`. This macro takes a `PROGMEM` pointer as its argument, and returns the byte located at that pointer value. To mark that our new routine deals in `PROGMEM` strings, let's append a "`_P`" to it's name just like the other routines in `pgmspace.h`:

```
void USART_TxString_P(const char *data)
{
   while (pgm_read_byte(data) != 0x00)
     USART_Tx(pgm_read_byte(data++));
}
```

Now we have our `PROGMEM`-aware routine, we can use `PROGMEM`-attributes strings:

```
USART_TxString_P(PSTR("FLASH STRING"));
```

Or:

```
const char TestFlashStr[] PROGMEM = "FLASH STRING";
USART_TxString_P(TestFlashStr);
```

This should give you a basic idea of how to store strings and keep them in flash memory space. What follows is the second half of this tutorial, for more advanced uses of the `PROGMEM` attribute.

Chapter 3

More advanced uses of PROGMEM

By now you should be able to create and use your own simple strings stored in program memory. But that's not the end of the `PROGMEM` road — there's still plenty more to learn! In this chapter, I'll cover two slightly more advanced techniques using the `PROGMEM` attribute. The first part will lead on to the second, so please read both.

3.1 Storing data arrays in program memory

It's important to realize that all static data in your program can be kept in program memory without being read out into RAM. While strings are by far the most common reason for using the `PROGMEM` attribute, you can also store arrays too.

This is especially the case when you are dealing with font arrays for a LCD, or the like. Consider the following (trimmed) code for an Atmel AVR Butterfly LCD font table:

```
static unsigned int LCD_SegTable[] PROGMEM =
{
    0xEAA8,     // '*'
    0x2A80,     // '+'
    0x4000,     // ','
    0x0A00,     // '-'
    0x0A51,     // '.' Degree sign
    0x4008,     // '/'
}
```

Because we've added the `PROGMEM` attribute, the table is now stored firmly in flash memory. The overall savings for a table this size is negligible, but in a practical application the data to be stored may be many times the size shown here — and without the `PROGMEM` attribute, all that will be copied into RAM.

First thing to notice here is that the table data is of the type `unsigned int`. This means that our data is two bytes long (for the AVR), and so our `pgm_read_byte()` macro won't suffice for this instance.

Thankfully, another macro exists; `pgm_read_word()`. A word for the AVR is two bytes long — the same size as an `int`. Because of this fact, we can now make use of it to read out our table data. If we wanted to grab the fifth element of the array, the following extract would complete this purpose:

```
pgm_read_word(&LCD_SegTable[4])
```

Note that we are taking the address of the fourth element of `LCD_SegTable`, which is an address in flash and not RAM due to the table having the `PROGMEM` attribute.

3.2 PROGMEM Pointers

It's hard to remember that there is a layer of abstraction between flash memory access — unlike RAM variables we can't just reference the contents at will without macros to manage the separate memory space — macros such as `pgm_read_byte()`. But when it all really starts confusing is when you have to deal with pointers to `PROGMEM` strings which are also held in `PROGMEM`.

Why could this possibly be useful you ask? Well, I'll start as usual with a short example.

```
char MenuItem1[] = "Menu Item 1";
char MenuItem2[] = "Menu Item 2";
char MenuItem3[] = "Menu Item 3";

char* MenuItemPointers[] = {MenuItem1, MenuItem2, MenuItem3};
```

Here we have three strings containing menu function names. We also have an array which points to each of these three items. Let's use our pretend function `USART_TxString()` from part I of this tutorial. Say we want to print out the menu item corresponding with a number the user presses, which we'll assume is returned by an imaginary function named `USART_GetNum()`. Our (pseudo-)code might look like this:

```
#include <avr/io.h>

char MenuItem1[] = "Menu Item 1";
char MenuItem2[] = "Menu Item 2";
char MenuItem3[] = "Menu Item 3";

char* MenuItemPointers[] = {MenuItem1, MenuItem2, MenuItem3};

void main(void)
{
   while (1) // Eternal Loop
   {
      char EnteredNum = USART_GetNum();

      USART_TxString(MenuItemPointers[EnteredNum]);
   }
}
```

Those confident with the basics of C will see no problems with this - it's all non-`PROGMEM` data and so it all can be interacted with in the manner of which we are accustomed. But what happens if the menu item strings are placed in `PROGMEM`?

```
#include <avr/io.h>
#include <avr/pgmspace.h>

const char MenuItem1[] PROGMEM = "Menu Item 1";
const char MenuItem2[] PROGMEM = "Menu Item 2";
const char MenuItem3[] PROGMEM = "Menu Item 3";

const char* const MenuItemPointers[] = {MenuItem1, MenuItem2, MenuItem3};

void main(void)
{
   while (1) // Eternal Loop
   {
      char EnteredNum = USART_GetNum();

      USART_TxString_P(MenuItemPointers[EnteredNum]);
   }
}
```

Easy! We add the `PROGMEM` attribute to our strings and then substitute the RAM-aware `USART_TxString()` routine with the `PROGMEM`-aware one we delt with in part I, `USART_TxString_P()`. I've also added in the `const` modifier as explained in part I, since it's good practice. But what if we want to save a final part of RAM and store our pointer table in `PROGMEM` also? This is where it gets slightly more complicated.

Let's try to modify our program to put the pointer array into `PROGMEM` and see if it works.

```
#include <avr/io.h>
#include <avr/pgmspace.h>

const char MenuItem1[] PROGMEM = "Menu Item 1";
const char MenuItem2[] PROGMEM = "Menu Item 2";
const char MenuItem3[] PROGMEM = "Menu Item 3";

const char* const MenuItemPointers[] PROGMEM = {MenuItem1, MenuItem2, MenuItem3};

void main(void)
{
   while (1) // Eternal Loop
   {
      char EnteredNum = USART_GetNum();

      USART_TxString_P(MenuItemPointers[EnteredNum]);
   }
}
```

Hmm, no luck. The reason is simple: although the pointer table contains valid pointers to strings in `PROGMEM`, now that it is also in `PROGMEM` we need to read it via the `pgm_read_x()` macros.

First, it's important to know how pointers in GCC are stored.

Pointers in GCC are usually two bytes long, i.e. 16 bits. A 16-bit pointer is the smallest sized integer capable of holding the address of any byte within 64kb of memory. Because of this, our `uint8_t*` typed pointer table's elements are all 16-bits long. The data the pointer is addressing is an unsigned character of 8 bits, but the pointer itself is 16 bits wide.

Ok, so now we know our pointer size. How can we read our 16-bit pointer out of `PROGMEM`? With the `pgm_read_word()` macro of course!

```
#include <avr/io.h>
#include <avr/pgmspace.h>

const char MenuItem1[] PROGMEM = "Menu Item 1";
const char MenuItem2[] PROGMEM = "Menu Item 2";
const char MenuItem3[] PROGMEM = "Menu Item 3";

const char* const MenuItemPointers[] PROGMEM = {MenuItem1, MenuItem2, MenuItem3};

void main(void)
{
   while (1) // Eternal Loop
   {
      char EnteredNum = USART_GetNum();

      USART_TxString_P(pgm_read_word(&MenuItemPointers[EnteredNum]));
   }
}
```

Almost there! GCC will probably give you warnings with code like this, but will most likely generate the correct code. The problem is the typecast; `pgm_read_word()` is returning a word-sized value while `USART_TxString_P()` is expecting (if you look back at the parameters for the function in part I) a `char*` pointer type. While the sizes of the two data types are the same (remember, the pointer is 16-bits!) to the compiler it looks like we are trying to do the code equivalent of shoving a square peg in a round hole.

To instruct the compiler that we really do want to use that 16-bit return value of `pgm_read_word()` as a pointer to a const char in flash memory, we need to typecast it to a pointer in the form of `char*`. Our final program will now look like this:

```
#include <avr/io.h>
#include <avr/pgmspace.h>

const char MenuItem1[] PROGMEM = "Menu Item 1";
const char MenuItem2[] PROGMEM = "Menu Item 2";
const char MenuItem3[] PROGMEM = "Menu Item 3";

const char* const MenuItemPointers[] PROGMEM = {MenuItem1, MenuItem2, MenuItem3};

void main(void)
{
   while (1) // Eternal Loop
   {
      char EnteredNum = USART_GetNum();

      USART_TxString_P((char*)pgm_read_word(&MenuItemPointers[EnteredNum]));
   }
}
```

I recommend looking through the *avr-libc* documentation of the `<avr/pgmspace.h>` header documentation section for information about the other `PROGMEM` related library functions available. Other functions — such as `pgm_read_dword()`, which reads four bytes (double that of a word for the AVR architecture) — behave in a similar manner to the functions detailed in this tutorial and might be of use for your end-application.

3.3 PROGMEM Strings and `printf()`

Finally, let's cover another advanced use case of `PROGMEM` strings; printing them out using the C string formatting library function `printf()`. Not everyone is a fan of the string formatting functions located in the C standard library's `<stdio.h>` header due to their large size and slow execution speed, but they can make complex applications quite a bit simpler if you can stomach the size/performance penalties. Lots of applications will typically contain code that looks something like this:

```
char ParameterA[] = "Parameter A";
char ParameterB[] = "Parameter B";

void PrintParameterValue(char* ParameterName, uint8_t ParameterValue)
{
   printf("The value of %s is %d", ParameterName, ParameterValue);
}
```

Note that for clarity, the initialization code of the program is omitted – if you aren't sure how to do this, see the *avr-libc* library documentation. Here we have a simple function that formats a string and prints it to the standard output; the call site passes in a pointer to a string and a 8-bit integer, and the function formats it into a nice human-readable string and writes it out. The call site might look something like this:

```
void WriteOutParameters(void)
{
   PrintParameterValue(ParameterA, 10);
   PrintParameterValue(ParameterB, 20);
}
```

As we have done before, let's modify the code so that our fixed-strings `ParameterA` and `ParameterB` can be located in FLASH memory rather than RAM. First, we add the `const` and `PROGMEM` qualifiers to the global variables and the pointer passed into the function:

```
const char ParameterA[] PROGMEM = "Parameter A";
const char ParameterB[] PROGMEM = "Parameter B";

void PrintParameterValue(const char* ParameterName, uint8_t ParameterValue)
{
   printf("The value of %s is %d", ParameterName, ParameterValue);
}
```

This application now compiles, but running it will produce incorrect output. Once again, we have the problem of a function (in this case, the *avr-libc* provided `printf()`) expecting a string that is located in RAM, while we're instead passing it a string located in FLASH. In the C language specification for the `printf()` function a number of special formatting strings (that's the `%s`, `%d`, `%g`, etc. that tells the function how to format the input parameters) are defined, but no specification was made to deal with multiple memory spaces. This isn't so suprising when you consider the origins of C and its single-memory space model, but it's a problem for us nonetheless.

To fix up our code, we need to yet-again consult the *avr-libc* manual, where we will find that the library implementors have helpfully provided some special extensions to the standard `printf()` function - specifically, there's a non-standard `%S` – note the capital "S"! – format specifier which tells the function the source string is located in `PROGMEM`. Changing our format string to use `%S` instead of the previous `%s` fixes the application and allows us to merrily print out strings from either RAM or FLASH in our code using `printf()`, `sprintf()` and its related functions.

```
const char ParameterA[] PROGMEM = "Parameter A";
const char ParameterB[] PROGMEM = "Parameter B";

void PrintParameterValue(const char* ParameterName, uint8_t ParameterValue)
{
   printf("The value of %S is %d", ParameterName, ParameterValue);
}
```

Unfortunately no such extensions exist for non-string inputs; so if you have integers for example that you have located into FLASH with `PROGMEM`, you will need to read them out and buffer them into a RAM location before using `printf()`.

One caveat to the above: newer versions of the C specification reserve the `%S` formatting specifier for wide (Unicode) character strings, which is at odds with avr-libc's use of it for standard strings located in FLASH memory. This can trigger compile warnings on newer versions of avr-gcc:

```
warning: format '%S' expects argument of type 'wchar_t *', but argument 2 has type 'const char
    *' [-Wformat=]
```

The only way to work around this for now is to compile with the `-Wno-format` compile option, which disables the in-built compile-time checking of `printf()` and `scanf()` style formatting strings.