[TUT] [C] GCC and the PROGMEM Attribute

Author	Message
abcminiuser	**Posted:** Apr 26, 2006 - 08:15 AM

Moderator

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

PART I - PROGMEM BASICS

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:

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.

Entering pgmspace.h

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):

Code:

```
#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:

Code:

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

Although it is not an absolute requirement, we can remind ourselves - and possibly prevent bugs further down the track - that the string cannot be changed at all during execution by declaring it as a constant:

Code:

```
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:

Code:

```
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 the PSTR string once.

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

The problem is that your string functions are expecting the string to be inside RAM. When you pass a string to a routine, you are in fact just passing a pointer to the start of the string 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:

Code:

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

For a full list of the avaliable string functions, check out the AVRLibC documentation which is installed with your WinAVR installation.

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

Code:

```
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?

pgmspace.h 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*:

Code:

```
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:

Code:

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

Or:

Code:

```
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.

PART II - More advanced uses of PROGMEM

Ok, so 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 second tutorial section, 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.

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 AVRButterfly LCD font table:

Code:

```
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 out prgm_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:

Code:

```
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.

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 prgm_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.

Code:

```
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:

Code:

```
#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?

Code:

```
#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";

char* 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.

Code:

```
#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";

char* 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 prgm_read_x macros.

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

Pointers in GCC are usually two bytes long - 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!

Code:

```
#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";

char* 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. 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:

Code:

```
#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";

char* 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-lib-c documentation (included with the installation of WinAVR in the \docs directory) in the AVR/PgmSpace.h header documentation section for information about the other PROGMEM related library functions avaliable. 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.

And so ends the second part of this tutorial. Feedback and corrections welcome!

- Dean

Last edited by abcminiuser on Mar 29, 2008 - 02:07 AM; edited 19 times in total

profile pm email www quote

barnacle

Joined: Jan 03, 2006
Posts: 3300
Location: Hemel Hempstead, UK

Posted: Apr 26, 2006 - 09:49 AM

This is a very useful technique since it's easy to write code in which the stack bounces off the variables - and the real gotcha is that all those 'printf' statements you put in to try and trace it make it worse; they're eating the ram space. Indeed, my Coupe ECU monitor had to have a complete rewrite to cope with a change of language from English to Italian; the Italian was a couple of dozen characters longer and that was enough to break it.

Very embarrassing...

A minor correction: should

Code:

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

really be

Code:

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

(Oh, and 'solely' rather than 'souley')

Neil

Neil Barnes
www.nailed-barnacle.co.uk

profile pm email www quote

abcminiuser

Moderator

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

Posted: Apr 26, 2006 - 09:54 AM

Fixed, and fixed. For the former, I was missing the closing quote on the RAM string. Thanks for the comments (and great story).

- Dean

profile pm email www quote

JimW52

Joined: Jun 01, 2004
Posts: 350
Location: Brisbane, Australia

Posted: Apr 27, 2006 - 10:58 AM

Still not quite right.

Code:

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

should be

Code:

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

Jim

profile pm quote

abcminiuser

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

Posted: Apr 27, 2006 - 11:06 AM

Darnit! Fixed. Guess i'm not one to teach after all .

Is it worth me posting more advanced techniques like PROGMEM arrays of pointers to PROGMEM strings, or even creating a GCC EEMEM tutorial?

- Dean

profile pm email www quote

JimW52

Posted: Apr 27, 2006 - 11:18 AM

Don't feel too bad, it took me two goes to get the correction right.

As for the other tutorials - yes go ahead. There will always be someone here to pick up the odd typo.

Keep up the good work.

Joined: Jun 01, 2004
Posts: 350
Location: Brisbane, Australia

Jim

profile pm quote

svofski

Posted: Apr 27, 2006 - 11:30 AM

Joined: Jun 27, 2005
Posts: 2799
Location: St.Petersburg, Russia

Thanks for explaining this Dean! I've been lazy to dig up all this information myself and now I found this tutorial and it's all clear. Great job!

_____
The Dark Boxes are coming.

profile pm www quote

DO1THL

Posted: Apr 27, 2006 - 12:37 PM

Joined: Aug 29, 2002
Posts: 346
Location: Muenster, Germany

Quote:

> It's hard to remember that there is a layer of abstraction between flash memory access - unlike RAM variables we can't just manipulate the contents at will.

Well, the layer of abstraction is not that we can't **manipulate** the contents (flash data can't be manipulated at all), but that we can't use C's normal way or **referencing** them.

But maybe this is again splitting hairs ... but then it would be newbie's hair, which is much more worth than mine, 'cause it will be gone within some years

profile pm quote

abcminiuser

Posted: Apr 27, 2006 - 12:50 PM

Moderator

Fixed. Honestly, I should start putting revision numbers on all my written works .
Thanks for the comment.

- Dean :twisted

POST REV: 1.0

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

profile pm email www quote

lfmorrison

Posted: Apr 27, 2006 - 01:07 PM

Joined: Dec 08, 2004
Posts: 4517
Location: Nova Scotia, Canada

It may be worth adding a discussion of the pro's and cons of the two techniques that are available for tables of PROGMEM strings.

You can create a table of pointers, and each pointer points to an individual string, as discussed in the tutorial above.

eg.

Code:

```
char String1[] PROGMEM = "String 1";
char String2[] PROGMEM = "String 2";
char String3[] PROGMEM = "String 3";
char String4[] PROGMEM = "String 4";

PGM_P StringTable[NUMBER_OF_STRINGS] PROGMEM = {
String1,
String2,
String3,
String4
};
```

Alternatively, you can create a unified two-dimensional array, which is entirely contained in PROGMEM. Each entry in the top-level of the array is a fixed-length array of characters.

eg.

Code:

```
char StringTable[NUMBER_OF_STRINGS][FIXED_STRING_LENGTH] PROGMEM = {
{"String 1"},
{"String 2"},
{"String 3"},
{"String 4"}
};
```

There are pro's and cons for each approach.
The major drawback of the two-dimensional array versus the table-of-pointers-to-strings, is that the "inner" dimension of the array has to be large enough to accommodate the largest individual string, even if none of the other strings need that much space. In the table-of-pointers approach, each individual string only reserves exactly enough memory to accommodate itself.

The major drawback of the table-of-pointers approach is two-fold: First, you have these extra two bytes associated with each and every string, where the pointers physically reside. The two-dimensional array approach doesn't have to literally store these pointers.

Second, you have to dereference PROGMEM twice in order to access the contents of an individual string: Once to extract the start-address of the string you want from the "top-level" table, and a second time to get at the individual character you're interested in within the target string. With a two-dimensional table, the compiler can compute the starting-addresses for the indiviudal strings deterministically, so you only need to dereference PROGMEM once, at the point where you're actually extracting an individual character from the strings.

The trade-off in storage requirement can be decided rather easily: If most of the individual strings that you want to place in the table are within one or two characters in length, your best bet is to use the two-dimensional table. In that case, the wasted space involved in padding the ends of the shorter strings will be smaller than the space that would be wasted by adding an extra two bytes for pointers to each-and-every string in a look-up table.

If you have a string set where there is a wide discrepancy between the longest and shortest strings, and most strings would have more than one or two bytes of NULL padding at the end to match the length of the longest string, then you'd be better off using the table-of-pointers approach.

[edit 1] Inserted code examples [/edit]
[edit 2] Fixed code tags [/edit]

Last edited by lfmorrison on Apr 27, 2006 - 01:13 PM; edited 2 times in total

profile pm quote

JimW52

Posted: Apr 27, 2006 - 01:12 PM

Rev 2

All references to prgm_read_word should be pgm_read_word.

Jim

Joined: Jun 01, 2004
Posts: 350
Location: Brisbane, Australia

profile pm quote

abcminiuser

Posted: Apr 27, 2006 - 01:32 PM

JimW52 wrote:

> Rev 2

> All references to prgm_read_word should be pgm_read_word.
>
> Jim

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

Fixed. Sigh .

LFMorrison: Excellent points. I'm used to dealing with odd-ball sized strings and I hate the idea of padding, so I usually opt for this approach.

- Dean

profile pm email www quote

timwhunt

Posted: Apr 27, 2006 - 06:16 PM

People might not want to fill this up with lots of "thanks" posts, but I want Dean and all those that helped to know that the time and attention that went into this are VERY MUCH APPRECIATED!! Stuff like this is great and exactly what I needed!

Joined: Nov 14, 2005
Posts: 163
Location: Philadelphia, USA

profile pm quote

abcminiuser

Posted: Apr 28, 2006 - 06:18 AM

Moderator

Cheers! I thought it was high time the tutorials forum was used for its intended purpose!

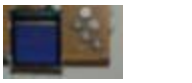
- Dean

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

profile pm email www quote

Lajon

Posted: Apr 28, 2006 - 10:00 AM



Joined: Mar 12, 2004
Posts: 1122
Location: Linköping, Sweden

In the examples where the pointer array is still in RAM (if anyone cares about them in this context) you should not have the '&' operator, it will not work. So

Code:

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

should be

Code:

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

and

Code:

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

should be

Code:

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

/Lars

abcminiuser

Posted: Apr 28, 2006 - 10:09 AM

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

You sure? USART_TxString is expecting an address for it's pointer. When I tested out the code:

Code:

```
uint8_t Data[5] = "Test";
   USART_TxString(Data[1]);
```

Produced a warning while:

Code:

```
   uint8_t Data[5] = "Test";
   USART_TxString(&Data[1]);
```

Did not. What am I missing?

- Dean

PS: Thanks for the feedback guys - it may look like i'm grumbing about all the errors, but I'm secretly pleased that there are people reading it in detail

LUFA - Lightweight USB Framework for AVRs 4WC
BLOG

profile pm email www quote

Rotamat

Posted: Apr 28, 2006 - 11:14 AM

I agree with Lajon.
@abcminiuser: in your last post you confuse *Data* (the string) and *MenuItemPointers* (the pointers to the strings)!

Joined: May 19, 2004
Posts: 6
Location: Pordenone, Italy

> *USART_TxString()* and *USART_TxString_P()* expects an *unsigned char**.
> *MenuItemPointers[x]* is an *unsigned char**.

So you need to pass the menu item (as stated by Lajon) without the '&'.
If you use *&MenuItemPointers[x]* you will pass the address of the pointer to the string; this is wrong.

Many thanks for your tutorial Dean!

Patrik

profile pm quote

abcminiuser

Posted: Apr 28, 2006 - 11:24 AM

Moderator

Whoops! You're right Patrick and Lars, sorry. I was indeed confusing the string array with the pointer array. Fixed, and thanks.

Should I combine the two parts at the top so that it's easier to read? I could also attach it in PDF form, if required.

- Dean

Joined: Jan 23, 2004
Posts: 7141
Location: Melbourne, Victoria, Australia

LUFA - Lightweight USB Framework for AVRs 4WC
BLOG

profile pm email www quote

Rotamat

Joined: May 19, 2004
Posts: 6
Location: Pordenone, Italy

Posted: Apr 28, 2006 - 12:56 PM

It's a good idea to put tutorials + notes from posts in one pdf file!

Another errata:

Code:

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

Corrige:

Code:

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

Patrik

profile pm quote

twidget2

Joined: Sep 04, 2004
Posts: 103
Location: Los Angeles

Posted: Jul 18, 2006 - 07:51 PM

minor typo shouldnt #include <avr/pgmspace.h.h>? be #include <avr/pgmspace.h>

profile pm quote

Display posts from previous: All Posts Oldest First Go

Jump to: |-- AVR Tutorials Go

All times are GMT + 1 Hour

1, 2, 3, 4, 5 ... 7 Next All

AVR Freaks Forum Index » AVR (8-bit) Technical Forums » AVR Tutorials