[TUT][SOFT][C][ASM] Data stored in FLASH.
by CmdrZin

This tutorial contains all the useful information I can think of that someone would need to store and access data stored in FLASH (Program Memory). It will start with simple data elements and progress to complex data structures. It will also cover how to handle arrays of these data objects.
The development environment will be Atmel Studio 7 and AVR-GCC compiler. Example targets will use the ATtiny84 and ATmega328 processors.

So why store data in FLASH? Probably the most common reason is that the processor does not have enough RAM to store the data. Another reason is to protect data that is declared as constant from being modified accidentally or by malicious intent.

First, some fundamentals.
1. FLASH can be accessed at the byte level through the Z register.
2. The macro PROGMEM is used to tell the compiler to place the data into FLASH.[1]
3. Whatever you store into FLASH has to be of a constant length. The compiler will require you to put a `const` declaration in front of the data type when using the `PROGMEM` macro.
4. The header file `<avr/pgmspace.h>` is needed to use the `PROGMEM` macro and provides several macros to read data from FLASH .[1]

Simple data storage
To store an elementary data type into FLASH, use the `PROGMEM` macro. The data has to be declared `const` as well.
Examples:
```
    const char x PROGMEM = 6;                       // one byte
    const uint16_t y PROGMEM = 0x1234;              // two bytes
    const uint32_t z PROGMEM = 0x98765432;          // four bytes
    const char mystring[] = “This is a string.”;    // null terminated.
```

There are several macros available to access these in FLASH.[3]
```
    pgm_read_byte()
    pgm_read_word()
    pgm_read_dword()
    pgm_read_float()
    pgm_read_ptr()
```

They are used by passing in the address of the variable that is stored in FLASH.

Example (see SimpleDataAccess : main.c):
```
const uint8_t x PROGMEM = 5;

DDRB = pgm_read_byte(&x);         // use the address of x to access FLASH
```

Read a string array using an index.
```
const char mystring[] PROGMEM = "This is a string.";   // null terminated.
```

```
index = 0;
while ( (data = pgm_read_byte(&mystring[index])) != 0 )
{
       DDRD = data;
       ++index;
}
```

Although these examples don't do anything useful, but they do show the use of the macros.

More useful examples are when arrays of data are stored. Similar to the string array, these data arrays are normally accessed by index. They can also be accessed by pointer adjustment.
(see ArrayDataAccess)

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

const uint8_t prime[] PROGMEM = {1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31};
const uint32_t noise[] PROGMEM = {0x11111111, 0x33333333, 0x55555555, 0x07070707,
        0x0F0F0F0F, 0x1B1B1B1B};

int main(void)
{
   uint32_t* dptr;
   uint32_t data;

    while (1)
    {
       // Access byte array by index
       for (int i = 0; i<sizeof(prime); i++) {
           DDRD = pgm_read_byte(&prime[i]);
       }

       // Access double word array by pointer adjust
       // NOTE the adjustment needed in the sizeof() calculation.
       dptr = noise;
       for (int i = 0; i<(sizeof(noise)/sizeof(uint32_t)); i++) {
           data = pgm_read_dword(dptr++);
           DDRD = data;
           DDRC = data>>8;
           DDRB = data>>16;
       }
    }
}
```

Multiple dimensional arrays can also be accessed in similar ways. NOTE the lower dimension(s) have to be declared.

String arrays have to be handled a little differently since data in FLASH has to be a constant size and all elements of an array have to be of the same size. The three step process for declaring an array of strings is to
1. Declare the individual string.
2. Create a struct to hold the pointer.
2. Declare an array of structs holding pointers to the individual strings.

Example (see StringArray)

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

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

// These are arrays of char.
const char string0[] PROGMEM = "This is a string1.";   // null terminated.
const char string1[] PROGMEM = "A string2.";                  // null terminated.
const char string2[] PROGMEM = "And a string3.";       // null terminated.
// This is an array of pointers.
// const char *strings[3] PROGMEM = { string0, string1, string2 };  // compiler error

// Make a struct to hold the pointer.
typedef struct {
   const char* strPtr;
} STRING_PTR;
// then make an array of structs.
const STRING_PTR strings[] PROGMEM =
{
   { string0 },
   { string1 },
   { string2 }
};

int main(void)
{
   char data;
   char* sptr;

    while (1)
    {
      // Pointer has to be cast to the proper type.
      sptr = (char*)pgm_read_ptr(&strings[1].strPtr);
      while ( (data = pgm_read_byte(sptr++)) != 0 )
      {
         DDRD = data;
      }
    }
}
```

As to the first attempt that generated an error, my guess is that the compiler does not know if the pointer is near or far and therefore does not treat it as an element of a constant size. Possibly the __attribute__ directive could be used in the declaration to set the pointer type.

Making an array of function pointers. Use as an indexed function table.

SAMPLE CODE

```
//const void (*func)() PROGMEM = test;                 // This generates a compiler error.
//const void (*afunc)()[] PROGMEM = {test};            // This also generates a compiler error.

/* +++ This does not generate a compiler error. +++ */
typedef struct
{
      void (*func)();
} FUNCTION_PTR;

const FUNCTION_PTR pfunc[] PROGMEM =
{
	{ test }
};
/* --- */

call = (void (*)())pgm_read_ptr(&pfunc[0]);	// call a function pointer stored in FLASH
call();
PORTB = pgm_read_byte(&w);				// output a byte from FLASH

void test() {
	PORTD = 0;
}
```

APPENDIX I – References

1. https://www.nongnu.org/avr-libc/user-manual/pgmspace.html - Data in Program Space

2. https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Variable-Attributes.html - 6.36 Specifying Attributes of Variables

3. https://www.nongnu.org/avr-libc/user-manual/group__avr__pgmspace.html - Program Space Utilities