<avr/pgmspace.h>: Program Space Utilities

Macros

#define **PROGMEM** __ATTR_PROGMEM__

#define **PGM_P** const char *

#define **PGM_VOID_P** const void *

#define **PSTR**(s) ((const **PROGMEM** char *)(s))

#define **pgm_read_byte_near**(address_short) __LPM((**uint16_t**)(address_short))

#define **pgm_read_word_near**(address_short) __LPM_word((**uint16_t**)(address_short))

#define **pgm_read_dword_near**(address_short) __LPM_dword((**uint16_t**)(address_short))

#define **pgm_read_float_near**(address_short) __LPM_float((**uint16_t**)(address_short))

#define **pgm_read_ptr_near**(address_short) (void*)__LPM_word((**uint16_t**)(address_short))

#define **pgm_read_byte_far**(address_long) __ELPM((**uint32_t**)(address_long))

#define **pgm_read_word_far**(address_long) __ELPM_word((**uint32_t**)(address_long))

#define **pgm_read_dword_far**(address_long) __ELPM_dword((**uint32_t**)(address_long))

#define **pgm_read_float_far**(address_long) __ELPM_float((**uint32_t**)(address_long))

#define **pgm_read_ptr_far**(address_long) (void*)__ELPM_word((**uint32_t**)(address_long))

#define **pgm_read_byte**(address_short) **pgm_read_byte_near**(address_short)

#define **pgm_read_word**(address_short) **pgm_read_word_near**(address_short)

#define **pgm_read_dword**(address_short) **pgm_read_dword_near**(address_short)

#define **pgm_read_float**(address_short) **pgm_read_float_near**(address_short)

#define **pgm_read_ptr**(address_short) **pgm_read_ptr_near**(address_short)

#define **pgm_get_far_address**(var)

Typedefs

typedef void **PROGMEM** **prog_void**

typedef char **PROGMEM** **prog_char**

typedef unsigned char **PROGMEM** **prog_uchar**

typedef **int8_t** **PROGMEM** **prog_int8_t**

typedef **uint8_t** **PROGMEM** **prog_uint8_t**

typedef **int16_t** **PROGMEM** **prog_int16_t**

typedef **uint16_t** **PROGMEM** **prog_uint16_t**

typedef **int32_t** **PROGMEM** **prog_int32_t**

typedef **uint32_t** **PROGMEM** **prog_uint32_t**

typedef **int64_t** **PROGMEM** **prog_int64_t**

typedef **uint64_t** **PROGMEM** **prog_uint64_t**

Functions

const void *	**memchr_P** (const void *, int __val, size_t __len)
int	**memcmp_P** (const void *, const void *, size_t) __ATTR_PURE__
void *	**memccpy_P** (void *, const void *, int __val, size_t)
void *	**memcpy_P** (void *, const void *, size_t)
void *	**memmem_P** (const void *, size_t, const void *, size_t) __ATTR_PURE__
const void *	**memrchr_P** (const void *, int __val, size_t __len)
char *	**strcat_P** (char *, const char *)
const char *	**strchr_P** (const char *, int __val)
const char *	**strchrnul_P** (const char *, int __val)
int	**strcmp_P** (const char *, const char *) __ATTR_PURE__
char *	**strcpy_P** (char *, const char *)
int	**strcasecmp_P** (const char *, const char *) __ATTR_PURE__
char *	**strcasestr_P** (const char *, const char *) __ATTR_PURE__
size_t	**strcspn_P** (const char *__s, const char *__reject) __ATTR_PURE__
size_t	**strlcat_P** (char *, const char *, size_t)
size_t	**strlcpy_P** (char *, const char *, size_t)
size_t	**strnlen_P** (const char *, size_t)
int	**strncmp_P** (const char *, const char *, size_t) __ATTR_PURE__
int	**strncasecmp_P** (const char *, const char *, size_t) __ATTR_PURE__
char *	**strncat_P** (char *, const char *, size_t)
char *	**strncpy_P** (char *, const char *, size_t)
char *	**strpbrk_P** (const char *__s, const char *__accept) __ATTR_PURE__
const char *	**strrchr_P** (const char *, int __val)
char *	**strsep_P** (char **__sp, const char *__delim)
size_t	**strspn_P** (const char *__s, const char *__accept) __ATTR_PURE__
char *	**strstr_P** (const char *, const char *) __ATTR_PURE__
char *	**strtok_P** (char *__s, const char *__delim)
char *	**strtok_rP** (char *__s, const char *__delim, char **__last)
size_t	**strlen_PF** (**uint_farptr_t** src)
size_t	**strnlen_PF** (**uint_farptr_t** src, size_t len)
void *	**memcpy_PF** (void *dest, **uint_farptr_t** src, size_t len)
char *	**strcpy_PF** (char *dest, **uint_farptr_t** src)
char *	**strncpy_PF** (char *dest, **uint_farptr_t** src, size_t len)
char *	**strcat_PF** (char *dest, **uint_farptr_t** src)
size_t	**strlcat_PF** (char *dst, **uint_farptr_t** src, size_t siz)
char *	**strncat_PF** (char *dest, **uint_farptr_t** src, size_t len)
int	**strcmp_PF** (const char *s1, **uint_farptr_t** s2) __ATTR_PURE__

int	**strncmp_PF** (const char *s1, **uint_farptr_t** s2, size_t n) __ATTR_PURE__
int	**strcasecmp_PF** (const char *s1, **uint_farptr_t** s2) __ATTR_PURE__
int	**strncasecmp_PF** (const char *s1, **uint_farptr_t** s2, size_t n) __ATTR_PURE__
char *	**strstr_PF** (const char *s1, **uint_farptr_t** s2)
size_t	**strlcpy_PF** (char *dst, **uint_farptr_t** src, size_t siz)
int	**memcmp_PF** (const void *, **uint_farptr_t**, size_t) __ATTR_PURE__
static size_t	**strlen_P** (const char *s)

Detailed Description

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

The functions in this module provide interfaces for a program to access data stored in program space (flash memory) of the device. In order to use these functions, the target device must support either the `LPM` or `ELPM` instructions.

Note

These functions are an attempt to provide some compatibility with header files that come with IAR C, to make porting applications between different compilers easier. This is not 100% compatibility though (GCC does not have full support for multiple address spaces yet).

If you are working with strings which are completely based in ram, use the standard string functions described in **<string.h>: Strings**.

If possible, put your constant tables in the lower 64 KB and use **pgm_read_byte_near()** or **pgm_read_word_near()** instead of **pgm_read_byte_far()** or **pgm_read_word_far()** since it is more efficient that way, and you can still use the upper 64K for executable code. All functions that are suffixed with a `_P` *require* their arguments to be in the lower 64 KB of the flash ROM, as they do not use ELPM instructions. This is normally not a big concern as the linker setup arranges any program space constants declared using the macros from this header file so they are placed right after the interrupt vectors, and in front of any executable code. However, it can become a problem if there are too many of these constants, or for bootloaders on devices with more than 64 KB of ROM. *All these functions will not work in that situation.*

For **Xmega** devices, make sure the NVM controller command register (`NVM.CMD` or `NVM_CMD`) is set to 0x00 (NOP) before using any of these functions.

Macro Definition Documentation

◆ pgm_get_far_address

#define pgm_get_far_address (var)

Value:

```
({                                                  \
    uint_farptr_t tmp;                              \
                                                    \
    __asm__ __volatile__(                           \
                                                    \
            "ldi    %A0, lo8(%1)"           "\n\t"  \
            "ldi    %B0, hi8(%1)"           "\n\t"  \
            "ldi    %C0, hh8(%1)"           "\n\t"  \
            "clr    %D0"                    "\n\t"  \
        :                                           \
            "=d" (tmp)                              \
        :                                           \
            "p"  (&(var))                           \
    );                                              \
    tmp;                                            \
})
```

This macro facilitates the obtention of a 32 bit "far" pointer (only 24 bits used) to data even passed the 64KB limit for the 16 bit ordinary pointer. It is similar to the '&' operator, with some limitations.

Comments:

- The overhead is minimal and it's mainly due to the 32 bit size operation.
- 24 bit sizes guarantees the code compatibility for use in future devices.
- hh8() is an undocumented feature but seems to give the third significant byte of a 32 bit data and accepts symbols, complementing the functionality of hi8() and lo8(). There is not an equivalent assembler function to get the high significant byte.
- 'var' has to be resolved at linking time as an existing symbol, i.e, a simple type variable name, an array name (not an indexed element of the array, if the index is a constant the compiler does not complain but fails to get the address if optimization is enabled), a struct name or a struct field name, a function identifier, a linker defined identifier,...
- The returned value is the identifier's VMA (virtual memory address) determined by the linker and falls in the corresponding memory region. The AVR Harvard architecture requires non overlapping VMA areas for the multiple address spaces in the processor: Flash ROM, RAM, and EEPROM. Typical offset for this are 0x00000000, 0x00800xx0, and 0x00810000 respectively, derived from the linker script used and linker options. The value returned can be seen then as a universal pointer.

◆ PGM_P

#define PGM_P const char *

Used to declare a variable that is a pointer to a string in program space.

◆ pgm_read_byte

```
#define pgm_read_byte ( address_short ) pgm_read_byte_near(address_short)
```

Read a byte from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_byte_far

```
#define pgm_read_byte_far ( address_long ) __ELPM((uint32_t)(address_long))
```

Read a byte from the program space with a 32-bit (far) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_byte_near

```
#define pgm_read_byte_near ( address_short ) __LPM((uint16_t)(address_short))
```

Read a byte from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_dword

```
#define pgm_read_dword ( address_short ) pgm_read_dword_near(address_short)
```

Read a double word from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_dword_far

```
#define pgm_read_dword_far ( address_long ) __ELPM_dword((uint32_t)(address_long))
```

Read a double word from the program space with a 32-bit (far) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_dword_near

```
#define pgm_read_dword_near ( address_short ) __LPM_dword((uint16_t)(address_short))
```

Read a double word from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_float

```
#define pgm_read_float ( address_short ) pgm_read_float_near(address_short)
```

Read a float from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_float_far

```
#define pgm_read_float_far ( address_long ) __ELPM_float((uint32_t)(address_long))
```

Read a float from the program space with a 32-bit (far) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_float_near

```
#define pgm_read_float_near ( address_short ) __LPM_float((uint16_t)(address_short))
```

Read a float from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_ptr

```
#define pgm_read_ptr ( address_short ) pgm_read_ptr_near(address_short)
```

Read a pointer from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_ptr_far

```
#define pgm_read_ptr_far ( address_long ) (void*)__ELPM_word((uint32_t)(address_long))
```

Read a pointer from the program space with a 32-bit (far) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_ptr_near

```
#define pgm_read_ptr_near ( address_short ) (void*)__LPM_word((uint16_t)(address_short))
```

Read a pointer from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_word

#define pgm_read_word (address_short) **pgm_read_word_near**(address_short)

Read a word from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_word_far

#define pgm_read_word_far (address_long) __ELPM_word((**uint32_t**)(address_long))

Read a word from the program space with a 32-bit (far) address.

Note

The address is a byte address. The address is in the program space.

◆ pgm_read_word_near

#define pgm_read_word_near (address_short) __LPM_word((**uint16_t**)(address_short))

Read a word from the program space with a 16-bit (near) address.

Note

The address is a byte address. The address is in the program space.

◆ PGM_VOID_P

#define PGM_VOID_P const void *

Used to declare a generic pointer to an object in program space.

◆ PROGMEM

#define PROGMEM __ATTR_PROGMEM__

Attribute to use in order to declare an object being located in flash ROM.

◆ PSTR

#define PSTR (s) ((const **PROGMEM** char *)(s))

Used to declare a static pointer to a string in program space.

Typedef Documentation

◆ prog_char

prog_char

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <**avr/pgmspace.h**> (either by a `#define` directive, or by a -D compiler option.)

Type of a "char" object located in flash ROM.

◆ prog_int16_t

prog_int16_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <**avr/pgmspace.h**> (either by a `#define` directive, or by a -D compiler option.)

Type of an "int16_t" object located in flash ROM.

◆ prog_int32_t

prog_int32_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <**avr/pgmspace.h**> (either by a `#define` directive, or by a -D compiler option.)

Type of an "int32_t" object located in flash ROM.

◆ prog_int64_t

prog_int64_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <**avr/pgmspace.h**> (either by a `#define` directive, or by a -D compiler option.)

Type of an "int64_t" object located in flash ROM.

Note
This type is not available when the compiler option -mint8 is in effect.

◆ prog_int8_t

prog_int8_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <avr/pgmspace.h> (either by a `#define` directive, or by a -D compiler option.)

Type of an "int8_t" object located in flash ROM.

◆ prog_uchar

prog_uchar

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <avr/pgmspace.h> (either by a `#define` directive, or by a -D compiler option.)

Type of an "unsigned char" object located in flash ROM.

◆ prog_uint16_t

prog_uint16_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <avr/pgmspace.h> (either by a `#define` directive, or by a -D compiler option.)

Type of an "uint16_t" object located in flash ROM.

◆ prog_uint32_t

prog_uint32_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <avr/pgmspace.h> (either by a `#define` directive, or by a -D compiler option.)

Type of an "uint32_t" object located in flash ROM.

◆ prog_uint64_t

prog_uint64_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <avr/pgmspace.h> (either by a `#define` directive, or by a -D compiler option.)

Type of an "uint64_t" object located in flash ROM.

Note
This type is not available when the compiler option -mint8 is in effect.

◆ prog_uint8_t

prog_uint8_t

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <avr/pgmspace.h> (either by a `#define` directive, or by a -D compiler option.)

Type of an "uint8_t" object located in flash ROM.

◆ prog_void

prog_void

Note
DEPRECATED

This typedef is now deprecated because the usage of the __progmem__ attribute on a type is not supported in GCC. However, the use of the __progmem__ attribute on a variable declaration is supported, and this is now the recommended usage.

The typedef is only visible if the macro __PROG_TYPES_COMPAT__ has been defined before including <**avr/pgmspace.h**> (either by a `#define` directive, or by a -D compiler option.)

Type of a "void" object located in flash ROM. Does not make much sense by itself, but can be used to declare a "void *" object in flash ROM.

Function Documentation

◆ memccpy_P()

```
void * memccpy_P ( void *       dest,
                   const void * src,
                   int          val,
                   size_t       len
                 )
```

This function is similar to **memccpy()** except that `src` is pointer to a string in program space.

◆ memchr_P()

```
const void * memchr_P ( const void *  s,
                         int           val,
                         size_t        len
                       )
```

Scan flash memory for a character.

The **memchr_P()** function scans the first `len` bytes of the flash memory area pointed to by `s` for the character `val`. The first byte to match `val` (interpreted as an unsigned character) stops the operation.

Returns

The **memchr_P()** function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

◆ memcmp_P()

```
int memcmp_P ( const void *  s1,
               const void *  s2,
               size_t        len
             )
```

Compare memory areas.

The **memcmp_P()** function compares the first `len` bytes of the memory areas `s1` and flash `s2`. The comparision is performed using unsigned char operations.

Returns

The **memcmp_P()** function returns an integer less than, equal to, or greater than zero if the first `len` bytes of `s1` is found, respectively, to be less than, to match, or be greater than the first `len` bytes of `s2`.

◆ memcmp_PF()

```
int memcmp_PF ( const void *    s1,
                uint_farptr_t   s2,
                size_t          len
              )
```

Compare memory areas.

The **memcmp_PF()** function compares the first `len` bytes of the memory areas `s1` and flash `s2`. The comparision is performed using unsigned char operations. It is an equivalent of **memcmp_P()** function, except that it is capable working on all FLASH including the exteded area above 64kB.

Returns

The **memcmp_PF()** function returns an integer less than, equal to, or greater than zero if the first `len` bytes of `s1` is found, respectively, to be less than, to match, or be greater than the first `len` bytes of `s2`.

◆ memcpy_P()

```
void * memcpy_P ( void *          dest,
                  const void *    src,
                  size_t          n
                )
```

The **memcpy_P()** function is similar to **memcpy()**, except the src string resides in program space.

Returns

The **memcpy_P()** function returns a pointer to dest.

◆ memcpy_PF()

```
void * memcpy_PF ( void *          dest,
                   uint_farptr_t   src,
                   size_t          n
                 )
```

Copy a memory block from flash to SRAM.

The **memcpy_PF()** function is similar to **memcpy()**, except the data is copied from the program space and is addressed using a far pointer.

Parameters

dest A pointer to the destination buffer

src A far pointer to the origin of data in flash memory

n The number of bytes to be copied

Returns

The **memcpy_PF()** function returns a pointer to *dst*. The contents of RAMPZ SFR are undefined when the function returns.

◆ memmem_P()

```
void * memmem_P ( const void *  s1,
                  size_t        len1,
                  const void *  s2,
                  size_t        len2
                )
```

The **memmem_P()** function is similar to **memmem()** except that `s2` is pointer to a string in program space.

◆ memrchr_P()

```
const void memrchr_P ( const void *  src,
                       int           val,
                       size_t        len
                     )
```

The **memrchr_P()** function is like the **memchr_P()** function, except that it searches backwards from the end of the `len` bytes pointed to by `src` instead of forwards from the front. (Glibc, GNU extension.)

Returns

The **memrchr_P()** function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

◆ strcasecmp_P()

```
int strcasecmp_P ( const char *  s1,
                   const char *  s2
                 )
```

Compare two strings ignoring case.

The **strcasecmp_P()** function compares the two strings `s1` and `s2`, ignoring the case of the characters.

Parameters

s1 A pointer to a string in the devices SRAM.

s2 A pointer to a string in the devices Flash.

Returns

The **strcasecmp_P()** function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by **strcasecmp_P()** is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

◆ strcasecmp_PF()

```
int strcasecmp_PF ( const char *    s1,
                    uint_farptr_t  s2
                  )
```

Compare two strings ignoring case.

The **strcasecmp_PF()** function compares the two strings *s1* and *s2*, ignoring the case of the characters.

Parameters

s1 A pointer to the first string in SRAM

s2 A far pointer to the second string in Flash

Returns

The **strcasecmp_PF()** function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strcasestr_P()

```
char * strcasestr_P ( const char *  s1,
                      const char *  s2
                    )
```

This funtion is similar to **strcasestr()** except that `s2` is pointer to a string in program space.

◆ strcat_P()

```
char * strcat_P ( char *         dest,
                  const char *  src
                )
```

The **strcat_P()** function is similar to **strcat()** except that the *src* string must be located in program space (flash).

Returns

The **strcat()** function returns a pointer to the resulting string *dest*.

◆ strcat_PF()

```
char * strcat_PF ( char *          dst,
                   uint_farptr_t  src
                 )
```

Concatenates two strings.

The **strcat_PF()** function is similar to **strcat()** except that the *src* string must be located in program space (flash) and is addressed using a far pointer

Parameters

dst A pointer to the destination string in SRAM

src A far pointer to the string to be appended in Flash

Returns

The **strcat_PF()** function returns a pointer to the resulting string *dst*. The contents of RAMPZ SFR are undefined when the function returns

◆ strchr_P()

```
const char * strchr_P ( const char *  s,
                        int            val
                      )
```

Locate character in program space string.

The **strchr_P()** function locates the first occurrence of `val` (converted to a char) in the string pointed to by `s` in program space. The terminating null character is considered to be part of the string.

The **strchr_P()** function is similar to **strchr()** except that `s` is pointer to a string in program space.

Returns

The **strchr_P()** function returns a pointer to the matched character or `NULL` if the character is not found.

◆ strchrnul_P()

```
const char * strchrnul_P ( const char *  s,
                           int           c
                         )
```

The **strchrnul_P()** function is like **strchr_P()** except that if `c` is not found in `s`, then it returns a pointer to the null byte at the end of `s`, rather than `NULL`. (Glibc, GNU extension.)

Returns

The **strchrnul_P()** function returns a pointer to the matched character, or a pointer to the null byte at the end of `s` (i.e., `s+strlen(s)`) if the character is not found.

◆ strcmp_P()

```
int strcmp_P ( const char *  s1,
               const char *  s2
             )
```

The **strcmp_P()** function is similar to **strcmp()** except that `s2` is pointer to a string in program space.

Returns

The **strcmp_P()** function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by **strcmp_P()** is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

◆ strcmp_PF()

```
int strcmp_PF ( const char *    s1,
                uint_farptr_t  s2
              )
```

Compares two strings.

The **strcmp_PF()** function is similar to **strcmp()** except that *s2* is a far pointer to a string in program space.

Parameters

s1 A pointer to the first string in SRAM

s2 A far pointer to the second string in Flash

Returns

The **strcmp_PF()** function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strcpy_P()

```
char * strcpy_P ( char *          dest,
                  const char *  src
                )
```

The **strcpy_P()** function is similar to **strcpy()** except that src is a pointer to a string in program space.

Returns

The **strcpy_P()** function returns a pointer to the destination string dest.

◆ strcpy_PF()

```
char * strcpy_PF ( char *           dst,
                   uint_farptr_t   src
                 )
```

Duplicate a string.

The **strcpy_PF()** function is similar to **strcpy()** except that *src* is a far pointer to a string in program space.

Parameters

dst A pointer to the destination string in SRAM

src A far pointer to the source string in Flash

Returns

The **strcpy_PF()** function returns a pointer to the destination string *dst*. The contents of RAMPZ SFR are undefined when the funcion returns.

◆ strcspn_P()

```
size_t strcspn_P ( const char *  s,
                   const char *  reject
                 )
```

The **strcspn_P()** function calculates the length of the initial segment of `s` which consists entirely of characters not in `reject`. This function is similar to **strcspn()** except that `reject` is a pointer to a string in program space.

Returns

The **strcspn_P()** function returns the number of characters in the initial segment of `s` which are not in the string `reject`. The terminating zero is not considered as a part of string.

◆ strlcat_P()

```
size_t strlcat_P ( char *       dst,
                   const char * src,
                   size_t       siz
                 )
```

Concatenate two strings.

The **strlcat_P()** function is similar to **strlcat()**, except that the `src` string must be located in program space (flash).

Appends `src` to string `dst` of size `siz` (unlike **strncat()**, `siz` is the full size of `dst`, not space left). At most `siz-1` characters will be copied. Always NULL terminates (unless `siz` <= `strlen(dst)`).

Returns

The **strlcat_P()** function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

◆ strlcat_PF()

```
size_t strlcat_PF ( char *          dst,
                    uint_farptr_t   src,
                    size_t          n
                  )
```

Concatenate two strings.

The **strlcat_PF()** function is similar to **strlcat()**, except that the *src* string must be located in program space (flash) and is addressed using a far pointer.

Appends src to string dst of size *n* (unlike **strncat()**, *n* is the full size of *dst*, not space left). At most *n-1* characters will be copied. Always NULL terminates (unless *n* <= strlen(*dst*)).

Parameters

dst A pointer to the destination string in SRAM

src A far pointer to the source string in Flash

n The total number of bytes allocated to the destination string

Returns

The **strlcat_PF()** function returns strlen(*src*) + MIN(*n*, strlen(initial *dst*)). If retval >= *n*, truncation occurred. The contents of RAMPZ SFR are undefined when the funcion returns.

◆ strlcpy_P()

```
size_t strlcpy_P ( char *       dst,
                   const char * src,
                   size_t       siz
                 )
```

Copy a string from progmem to RAM.

Copy `src` to string `dst` of size `siz`. At most `siz-1` characters will be copied. Always NULL terminates (unless `siz` == 0). The **strlcpy_P()** function is similar to **strlcpy()** except that the `src` is pointer to a string in memory space.

Returns

The **strlcpy_P()** function returns strlen(src). If retval >= siz, truncation occurred.

◆ strlcpy_PF()

```
size_t strlcpy_PF ( char *          dst,
                    uint_farptr_t   src,
                    size_t          siz
                  )
```

Copy a string from progmem to RAM.

Copy src to string dst of size siz. At most siz-1 characters will be copied. Always NULL terminates (unless siz == 0).

Returns

The **strlcpy_PF()** function returns strlen(src). If retval >= siz, truncation occurred. The contents of RAMPZ SFR are undefined when the function returns.

◆ strlen_P()

size_t strlen_P (const char * src) static

The **strlen_P()** function is similar to **strlen()**, except that src is a pointer to a string in program space.

Returns

The **strlen_P()** function returns the number of characters in src.

Note

strlen_P() is implemented as an inline function in the **avr/pgmspace.h** header file, which will check if the length of the string is a constant and known at compile time. If it is not known at compile time, the macro will issue a call to __strlen_P() which will then calculate the length of the string as normal.

◆ strlen_PF()

size_t strlen_PF (**uint_farptr_t** s)

Obtain the length of a string.

The **strlen_PF()** function is similar to **strlen()**, except that *s* is a far pointer to a string in program space.

Parameters

s A far pointer to the string in flash

Returns

The **strlen_PF()** function returns the number of characters in *s*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strncasecmp_P()

```
int strncasecmp_P ( const char *  s1,
                    const char *  s2,
                    size_t        n
                  )
```

Compare two strings ignoring case.

The **strncasecmp_P()** function is similar to **strcasecmp_P()**, except it only compares the first `n` characters of `s1`.

Parameters

s1 A pointer to a string in the devices SRAM.

s2 A pointer to a string in the devices Flash.

n The maximum number of bytes to compare.

Returns

The **strncasecmp_P()** function returns an integer less than, equal to, or greater than zero if `s1` (or the first `n` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by **strncasecmp_P()** is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

◆ strncasecmp_PF()

```
int strncasecmp_PF ( const char *    s1,
                     uint_farptr_t  s2,
                     size_t         n
                   )
```

Compare two strings ignoring case.

The **strncasecmp_PF()** function is similar to **strcasecmp_PF()**, except it only compares the first *n* characters of *s1* and the string in flash is addressed using a far pointer.

Parameters

s1 A pointer to a string in SRAM

s2 A far pointer to a string in Flash

n The maximum number of bytes to compare

Returns

The **strncasecmp_PF()** function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strncat_P()

```
char * strncat_P ( char *         dest,
                   const char *  src,
                   size_t        len
                 )
```

Concatenate two strings.

The **strncat_P()** function is similar to **strncat()**, except that the *src* string must be located in program space (flash).

Returns

The **strncat_P()** function returns a pointer to the resulting string dest.

◆ strncat_PF()

```
char * strncat_PF ( char *          dst,
                    uint_farptr_t  src,
                    size_t         n
                  )
```

Concatenate two strings.

The **strncat_PF()** function is similar to **strncat()**, except that the *src* string must be located in program space (flash) and is addressed using a far pointer.

Parameters

dst A pointer to the destination string in SRAM

src A far pointer to the source string in Flash

n The maximum number of bytes to append

Returns

The **strncat_PF()** function returns a pointer to the resulting string *dst*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strncmp_P()

```
int strncmp_P ( const char *  s1,
                const char *  s2,
                size_t        n
              )
```

The **strncmp_P()** function is similar to **strcmp_P()** except it only compares the first (at most) n characters of s1 and s2.

Returns

The **strncmp_P()** function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

◆ strncmp_PF()

```
int strncmp_PF ( const char *    s1,
                 uint_farptr_t  s2,
                 size_t         n
               )
```

Compare two strings with limited length.

The **strncmp_PF()** function is similar to **strcmp_PF()** except it only compares the first (at most) *n* characters of *s1* and *s2*.

Parameters

s1 A pointer to the first string in SRAM

s2 A far pointer to the second string in Flash

n The maximum number of bytes to compare

Returns

The **strncmp_PF()** function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strncpy_P()

```
char * strncpy_P ( char *       dest,
                   const char * src,
                   size_t       n
                 )
```

The **strncpy_P()** function is similar to **strcpy_P()** except that not more than n bytes of src are copied. Thus, if there is no null byte among the first n bytes of src, the result will not be null-terminated.

In the case where the length of src is less than that of n, the remainder of dest will be padded with nulls.

Returns

The **strncpy_P()** function returns a pointer to the destination string dest.

◆ strncpy_PF()

```
char * strncpy_PF ( char *          dst,
                    uint_farptr_t  src,
                    size_t         n
                  )
```

Duplicate a string until a limited length.

The **strncpy_PF()** function is similar to **strcpy_PF()** except that not more than *n* bytes of *src* are copied. Thus, if there is no null byte among the first *n* bytes of *src*, the result will not be null-terminated.

In the case where the length of *src* is less than that of *n*, the remainder of *dst* will be padded with nulls.

Parameters

dst A pointer to the destination string in SRAM

src A far pointer to the source string in Flash

n The maximum number of bytes to copy

Returns

The **strncpy_PF()** function returns a pointer to the destination string *dst*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strnlen_P()

```
size_t strnlen_P ( const char *  src,
                   size_t        len
                 )
```

Determine the length of a fixed-size string.

The **strnlen_P()** function is similar to **strnlen()**, except that `src` is a pointer to a string in program space.

Returns

The strnlen_P function returns strlen_P(src), if that is less than `len`, or `len` if there is no '\0' character among the first `len` characters pointed to by `src`.

◆ strnlen_PF()

```
size_t strnlen_PF ( uint_farptr_t  s,
                    size_t         len
                  )
```

Determine the length of a fixed-size string.

The **strnlen_PF()** function is similar to **strnlen()**, except that *s* is a far pointer to a string in program space.

Parameters

s A far pointer to the string in Flash

len The maximum number of length to return

Returns

The strnlen_PF function returns strlen_P(*s*), if that is less than *len*, or *len* if there is no '\0' character among the first *len* characters pointed to by *s*. The contents of RAMPZ SFR are undefined when the function returns.

◆ strpbrk_P()

```
char * strpbrk_P ( const char *  s,
                   const char *  accept
                 )
```

The **strpbrk_P()** function locates the first occurrence in the string `s` of any of the characters in the flash string `accept`. This function is similar to **strpbrk()** except that `accept` is a pointer to a string in program space.

Returns

The **strpbrk_P()** function returns a pointer to the character in `s` that matches one of the characters in `accept`, or `NULL` if no such character is found. The terminating zero is not considered as a part of string: if one or both args are empty, the result will `NULL`.

◆ strrchr_P()

```
const char * strrchr_P ( const char *  s,
                         int            val
                       )
```

Locate character in string.

The **strrchr_P()** function returns a pointer to the last occurrence of the character `val` in the flash string `s`.

Returns

The **strrchr_P()** function returns a pointer to the matched character or `NULL` if the character is not found.

◆ strsep_P()

```
char * strsep_P ( char **       sp,
                  const char *  delim
                )
```

Parse a string into tokens.

The **strsep_P()** function locates, in the string referenced by `*sp`, the first occurrence of any character in the string `delim` (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or `NULL`, if the end of the string was reached) is stored in `*sp`. An ``empty'' field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in `*sp` to '\0'. This function is similar to **strsep()** except that `delim` is a pointer to a string in program space.

Returns

The **strsep_P()** function returns a pointer to the original value of `*sp`. If `*sp` is initially `NULL`, **strsep_P()** returns `NULL`.

◆ strspn_P()

```
size_t strspn_P ( const char *  s,
                  const char *  accept
                )
```

The **strspn_P()** function calculates the length of the initial segment of `s` which consists entirely of characters in `accept`. This function is similar to **strspn()** except that `accept` is a pointer to a string in program space.

Returns

The **strspn_P()** function returns the number of characters in the initial segment of `s` which consist only of characters from `accept`. The terminating zero is not considered as a part of string.

◆ strstr_P()

```
char * strstr_P ( const char *  s1,
                  const char *  s2
                )
```

Locate a substring.

The **strstr_P()** function finds the first occurrence of the substring `s2` in the string `s1`. The terminating '\0' characters are not compared. The **strstr_P()** function is similar to **strstr()** except that `s2` is pointer to a string in program space.

Returns

The **strstr_P()** function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

◆ strstr_PF()

```
char * strstr_PF ( const char *    s1,
                   uint_farptr_t  s2
                 )
```

Locate a substring.

The **strstr_PF()** function finds the first occurrence of the substring `s2` in the string `s1`. The terminating '\0' characters are not compared. The **strstr_PF()** function is similar to **strstr()** except that `s2` is a far pointer to a string in program space.

Returns

The **strstr_PF()** function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`. The contents of RAMPZ SFR are undefined when the function returns.

◆ strtok_P()

```
char * strtok_P ( char *         s,
                  const char *  delim
                )
```

Parses the string into tokens.

strtok_P() parses the string `s` into tokens. The first call to **strtok_P()** should have `s` as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to **strtok_P()**. The delimiter string `delim` may be different for each call.

The **strtok_P()** function is similar to **strtok()** except that `delim` is pointer to a string in program space.

Returns

The **strtok_P()** function returns a pointer to the next token or NULL when no more tokens are found.

Note

strtok_P() is NOT reentrant. For a reentrant version of this function see **strtok_rP()**.

◆ strtok_rP()

```
char * strtok_rP ( char *       string,
                   const char * delim,
                   char **      last
                 )
```

Parses string into tokens.

The **strtok_rP()** function parses `string` into tokens. The first call to **strtok_rP()** should have string as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to **strtok_rP()**. The delimiter string `delim` may be different for each call. `last` is a user allocated char* pointer. It must be the same while parsing the same string. **strtok_rP()** is a reentrant version of **strtok_P()**.

The **strtok_rP()** function is similar to **strtok_r()** except that `delim` is pointer to a string in program space.

Returns

The **strtok_rP()** function returns a pointer to the next token or NULL when no more tokens are found.