Standard C Input/Output Functions

The prototypes for these functions are placed in the file **stdio.h**, located in the .\INC subdirectory. This file must be **#include** -d before using the functions.

The standard C language I/O functions were adapted to work on embedded microcontrollers with limited resources.

The lowest level Input/Output functions are:

char getchar(void)

returns a character received by the UART, using polling.

void putchar(char c)

transmits the character c using the UART, using polling.

Prior to using these functions you must:

- initialize the UART's Baud rate
- enable the UART transmitter
- enable the UART receiver.

Example:

```
#include <mega8515.h>
#include <stdio.h>

/* quartz crystal frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600

void main(void) {
char k;

/* initialize the USART control register
   TX and RX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x18;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

while (1) {
        /* receive the character */
        k=getchar();
        /* and echo it back */
        putchar(k);
        };
}
```

If you intend to use other peripherals for Input/Output, you must modify accordingly the **getchar** and **putchar** functions like in the example below:

```
#include <stdio.h>

/* inform the compiler that an alternate version
   of the getchar function will be used */
#define _ALTERNATE_GETCHAR_
```

```
/* now define the new getchar function */
char getchar(void) {
/* write your code here */

}

/* inform the compiler that an alternate version
   of the putchar function will be used */
#define _ALTERNATE_PUTCHAR_

/* now define the new putchar function */
void putchar(char c) {
/* write your code here */

}
```

For the XMEGA chips the **getchar** and **putchar** functions use by default the **USARTC0**.
If you wish to use another USART, you must define the **_ATXMEGA_USART_** preprocessor macro prior to #include the **stdio.h** header file, like in the example below:

```
/* use the XMEGA128A1 USARTD0 for getchar and putchar functions */
#define _ATXMEGA_USART_ USARTD0

/* use the Standard C I/O functions */
#include <stdio.h>
```

The **_ATXMEGA_USART_** macro needs to be defined only once in the whole program, as the compiler will treat it like it is globally defined.

All the high level Input/Output functions use **getchar** and **putchar**.

void puts(char *str)

outputs, using **putchar**, the null terminated character string **str**, located in RAM, followed by a new line character.

void putsf(char flash *str)

outputs, using **putchar**, the null terminated character string **str**, located in FLASH, followed by a new line character.

int printf(char flash *fmtstr [, arg1, arg2, ...])

outputs formatted text, using **putchar**, according to the format specifiers in the **fmtstr** string.
The format specifier string **fmtstr** is constant and must be located in FLASH memory.
The implementation of **printf** is a reduced version of the standard C function.
This was necessary due to the specific needs of an embedded system and because the full implementation would require a large amount of FLASH memory space.
The function returns the number of outputed characters.

The format specifier string has the following structure:

```
%[flags][width][.precision][l]type_char
```

The optional **flags** characters are:
'-' left-justifies the result, padding on the right with spaces. If its not present, the result will be right-justified, padded on the left with zeros or spaces;
'+' signed conversion results will always begin with a '+' or '-' sign;
' ' if the value isn't negative, the conversion result will begin with a space. If the value is negative then it will begin with a '-' sign.

The optional **width** specifier sets the minimal width of an output value. If the result of the conversion is wider than the field width, the field will be expanded to accommodate the result, so not to cause field truncation.

The following width specifiers are supported:

n - at least n characters are outputted. If the result has less than n characters, then its field will be padded with spaces. If the '-' flag is used, the result field will be padded on the right, otherwise it will be padded on the left;
0n - at least n characters are outputted. If the result has less than n characters, it is padded on the left with zeros.

The optional **precision** specifier sets the maximal number of characters or minimal number of integer digits that may be outputted.
For the 'e', 'E' and 'f' conversion type characters the precision specifier sets the number of digits that will be outputted to the right of the decimal point.
The precision specifier always begins with a '.' character in order to separate it from the width specifier.
The following precision specifiers are supported:
none - the precision is set to 1 for the 'i', 'd', 'u', 'x', 'X' conversion type characters. For the 's' and 'p' conversion type characters, the char string will be outputted up to the first null character;
.0 - the precision is set to 1 for the 'i', 'd', 'u', 'x', 'X' type characters;
.n - n characters or n decimal places are outputted.
For the 'i', 'd', 'u', 'x', 'X' conversion type characters, if the value has less than n digits, then it will be padded on the left with zeros. If it has more than n digits then it will not be truncated.
For the 's' and 'p' conversion type characters, no more than n characters from the char string will be outputted.
For the 'e', 'E' and 'f' conversion type characters, n digits will be outputted to the right of the decimal point.
The precision specifier has no effect on the 'c' conversion type character.

The optional '**l**' input size modifier specifies that the function argument must be treated as a long int for the 'i', 'd', 'u', 'x', 'X' conversion type characters.

The **type_char** conversion type character is used to specify the way the function argument will be treated.
The following conversion type characters are supported:
'i' - the function argument is a signed decimal integer;
'd' - the function argument is a signed decimal integer;
'u' - the function argument is an unsigned decimal integer;
'e' - the function argument is a float, that will be outputted using the [-]*d.dddddd* e[±]*dd* format
'E' - the function argument is a float, that will be outputted using the [-]*d.dddddd* E[±]*dd* format
'f' - the function argument is a float, that will be outputted using the [-]*ddd.dddddd* format
'x' - the function argument is an unsigned hexadecimal integer, that will be outputted with lowercase characters;
'X' - the function argument is an unsigned hexadecimal integer, that will be outputted with with uppercase characters;
'c' - the function argument is a single character;
's' - the function argument is a pointer to a null terminated char string located in RAM;

'p' - the function argument is a pointer to a null terminated char string located in FLASH;
'%' - the '%' character will be outputted.

int sprintf(char *str, char flash *fmtstr [, arg1, arg2, ...])

this function is identical to **printf** except that the formatted text is placed in the null terminated character string **str**.
The function returns the number of outputed characters.

int snprintf(char *str, unsigned char size, char flash *fmtstr [, arg1, arg2, ...])
for the TINY memory model.

int snprintf(char *str, unsigned int size, char flash *fmtstr [, arg1, arg2, ...])
for the other memory models.

this function is identical to **sprintf** except that at most **size** (including the null terminator) characters are placed in the character string **str**.
The function returns the number of outputed characters.

In order to reduce program code size, there is the Project|Configure|C Compiler|Code Generation|(s)printf Features option.
It allows linking different versions of the printf and sprintf functions, with only the features that are really required by the program.

The following **(s)printf features** are available:

- **int** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', no width or precision specifiers are supported, only the '+' and ' ' flags are supported, no input size modifiers are supported
- **int, width** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', the width specifier is supported, the precision specifier is not supported, only the '+', '-', '0' and ' ' flags are supported, no input size modifiers are supported
- **long, width** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%' the width specifier is supported, the precision specifier is not supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported
- **long, width, precision** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', the width and precision specifiers are supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported
- **float, width, precision** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'e', 'E', 'f', 'x', 'X', '%', the width and precision specifiers are supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported.

The more features are selected, the larger is the code size generated for the printf and sprintf functions.

int vprintf(char flash *fmtstr, va_list argptr)

this function is identical to **printf** except that the **argptr** pointer, of **va_list** type, points to the variable list of arguments. The **va_list** type is defined in the stdarg.h header file.
The function returns the number of outputed characters.

int vsprintf(char *str, char flash *fmtstr, va_list argptr)

this function is identical to **sprintf** except that the **argptr** pointer, of **va_list** type, points to the variable list of arguments. The **va_list** type is defined in the stdarg.h header file.
The function returns the number of outputed characters.

int vsnprintf(char *str, unsigned char size, char flash *fmtstr, va_list argptr)
for the TINY memory model.

int vsnprintf(char *str, unsigned int size, char flash *fmtstr, va_list argptr)
for the other memory models.

this function is identical to **vsprintf** except that at most **size** (including the null terminator) characters are placed in the character string **str**.
The function returns the number of outputed characters.

char *gets(char *str, unsigned char len)

inputs, using **getchar**, the character string **str** terminated by the new line character.
The new line character will be replaced with 0.
The maximum length of the string is **len**. If **len** characters were read without encountering the new line character, then the string is terminated with 0 and the function ends.
The function returns a pointer to **str**.

signed char scanf(char flash *fmtstr [, arg1 address, arg2 address, ...])

formatted text input by scanning, using **getchar**, a series of input fields according to the format specifiers in the **fmtstr** string.
The format specifier string **fmtstr** is constant and must be located in FLASH memory.
The implementation of **scanf** is a reduced version of the standard C function.
This was necessary due to the specific needs of an embedded system and because the full implementation would require a large amount of FLASH memory space.
The format specifier string has the following structure:

```
%[width][l]type_char
```

The optional **width** specifier sets the maximal number of characters to read. If the function encounters a whitespace character or one that cannot be converted, then it will continue with the next input field, if present.

The optional '**l**' input size modifier specifies that the function argument must be treated as a long int for the 'i', 'd', 'u', 'x' conversion type characters.

The **type_char** conversion type character is used to specify the way the input field will be processed.

The following conversion type characters are supported:
'd' - inputs a signed decimal integer in a pointer to int argument;
'i' - inputs a signed decimal integer in a pointer to int argument;
'u' - inputs an unsigned decimal integer in a pointer to unsigned int argument;
'x' - inputs an unsigned hexadecimal integer in a pointer to unsigned int argument;
'f', 'e', 'E', 'g', 'G' - inputs a floating point value, with an optional exponent ('e' or 'E' followed by an optionally signed decimal integer value), in a pointer to float argument;
'c' - inputs an ASCII character in a pointer to char argument;
's' - inputs an ASCII character string in a pointer to char argument;
'%' - no input is done, a '%' is stored.

The function returns the number of successful entries, or -1 on error.

signed char sscanf(char *str, char flash *fmtstr [, arg1 address, arg2 address, ...])

this function is identical to **scanf** except that the formatted text is inputted from the null terminated character string **str**, located in RAM.

In order to reduce program code size, there is the Project|Configure|C Compiler|Code Generation|(s)scanf Features option.
It allows linking different versions of the scanf and sscanf functions, with only the features that are really required by the program.

The following **(s)scanf features** are available:
- **int, width** - the following conversion type characters are supported: 'c', 's', 'i', 'd', 'u', 'x', '%', the width specifier is supported, no input size modifiers are supported
- **long, width** - the following conversion type characters are supported: 'c', 's', 'i', 'd', 'u', 'x', '%', the width specifier is supported, only the 'l' input size modifier is supported
- **long, float, width** - the following conversion type characters are supported: 'c', 's', 'i', 'd', 'u', 'x', 'f', 'e', 'E', 'g', 'G', '%', the width specifier is supported, the 'l' input size modifier is supported only for integer types.

The more features are selected, the larger is the code size generated for the scanf and sscanf functions.

The following Standard C Input/Output functions are used for file access on MMC/SD/SD HC FLASH memory cards.
Before using any of these functions, the logical drive that needs to be accessed must be mounted using the f_mount function and the appropriate file must be opened using the f_open function.

unsigned char feof(FIL* fp)

establishes if the end of file was reached during the last file access.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

If the end of file was reached, a non-zero value (1) will be returned.
Otherwise, the function will return 0.

unsigned char ferror(FIL *fp)

establishes if an error occurred during the last file access.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

If an error occurred, a non-zero value (1) will be returned.
Otherwise, the function will return 0.

int fgetc(FIL *fp)

reads a character from a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value: an 8bit character in the LSB, the MSB beeing 0.
In case of error or if the end of file was reached, the function returns the value of the predefined macro **EOF** (-1).
The **ferror** function must be used to establish if an error occured.
In order to check if the end of file was reached, the **feof** function must be called.

char *fgets(char *str,unsigned int len,FIL *fp)

reads from a file maximum **len**-1 characters to the char array pointed by **str.**
The read process stops if a new-line '\n' character is encountered or the end of file was reached. The new-line character is also saved in the char string.
The string is automatically NULL terminated after the read process is stopped.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns the same pointer as **str**.
If the end of file was encountered and no characters were read, a NULL pointer is returned.
The same happens in case of file access error.
The **ferror** function must be used to establish if an error occured.
In order to check if the end of file was reached, the **feof** function must be called.

int fputc(char k,FIL* fp)

writes the character **k** to a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value which represents the same character as **k**, the MSB beeing 0.
In case of error the function returns the value of the predefined macro **EOF** (-1).

int fputs(char *str,FIL* fp)

writes to a file the NULL terminated char string, stored in RAM, pointed by **str**.
The terminating NULL character is not written to the file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value (1).
In case of error the function returns the value of the predefined macro **EOF** (-1).

int fputsf(char flash *str,FIL* fp)

writes to a file the NULL terminated char string, stored in FLASH memory, pointed by **str**.
The terminating NULL character is not written to the file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value (1).
In case of error the function returns the value of the predefined macro **EOF** (-1).

int fprintf(FIL *fp, char flash *fmtstr,...)

this function is identical to **printf** except that the formatted text is written to a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value: the number of written characters.
In case of error the function returns the value of the predefined macro **EOF** (-1).

int fvprintf(FIL *fp, char flash *fmtstr, va_list argptr)

this function is identical to **vprintf** except that the formatted text is written to a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value: the number of written characters.
In case of error the function returns the value of the predefined macro **EOF** (-1).

int fscanf(FIL *fp, char flash *fmtstr,...)

this function is identical to **scanf** except that the formatted text is read from a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value: the number of read entries.
In case of error the function returns the value of the predefined macro **EOF** (-1).