# sprintf

## Summary

```c
#include <stdio.h>

int sprintf (
    char *buffer,            /* storage buffer */
    const char *fmtstr       /* format string */
    ...
    );  /* additional arguments */
```

## Description

The `sprintf` function formats a series of strings and numeric values and stores the resulting string in `buffer`. This function is similar to the `printf` routine, but it stores the formatted output in `buffer` rather than sending it to the output stream.

The `fmtstr` argument is a pointer to a format string which has the same form and function as the `printf` function's format string. The list of arguments are converted and output according to the corresponding format specifications in `fmtstr`.

## Return Value

The `sprintf` function returns the number of characters actually written to `buffer`.

## See Also

`gets`, `puts`, `vprintf`, `vsprintf`

## Example

```c
#include <stdio.h>

void tst_sprintf (void) {
    char buf [100];
    int n;
    int a,b;
    float pi;
    a = 123;
    b = 456;
    pi = 3.14159;
    n = sprintf (buf, "%f\n", 1.1);
    n += sprintf (buf+n, "%d\n", a);
    n += sprintf (buf+n, "%d %s %g", b, "---", pi);
    puts (buf);
}
```
### printf Summary

```c
#include <stdio.h>

int printf (
    const char *fmtstr /* format string */
    <[>, arguments ... <]>); /* additional arguments */
```

### Description

The **printf** function formats a series of strings and numeric values and builds a string to write to the output stream using the **putchar** function. The *fmtstr* argument is a format string that may be composed of characters, escape sequences, and format specifications.

Ordinary characters and escape sequences are copied to the stream in the order in which they are interpreted. Format specifications always begin with a percent sign ('%') and require that additional *arguments* are included in the **printf** function call.

The format string is read from left to right. The first format specification encountered references the first *argument* after *fmtstr* and converts and outputs it using the format specification. The second format specification accesses the second *argument* after *fmtstr*, and so on. If there are more *arguments* than format specifications, extra *arguments* are ignored. Results are unpredictable if there are not enough *arguments* for the format specifications or if the argument types do not match those specified by *fmtstr*.

Format specifications have the following general format:

```
% <[flags]> <[width]> <[.precision]> <[b|B|l|L]<] type
```

Each field in the format specification may be a single character or a number which specifies a particular format option.

The *type* field is a single character that specifies whether the argument is interpreted as a character, string, number, or pointer, as shown in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument Type</th>
<th>Input Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>int</td>
<td>Signed decimal number.</td>
</tr>
<tr>
<td>u</td>
<td>unsigned int</td>
<td>Unsigned decimal number.</td>
</tr>
<tr>
<td>o</td>
<td>unsigned int</td>
<td>Unsigned octal number.</td>
</tr>
<tr>
<td>x</td>
<td>unsigned int</td>
<td>Unsigned hexadecimal number using &quot;0123456789abcdef&quot;.</td>
</tr>
<tr>
<td>X</td>
<td>unsigned int</td>
<td>Unsigned hexadecimal number using &quot;0123456789ABCDEF&quot;.</td>
</tr>
</tbody>
</table>
f float | Floating-point number formatted as 
<[>-]dddd.dddd.
---|---
e float | Floating-point number formatted as 
<[>-]d.ddde<[>-]dd.
---|---
E float | Floating-point number formatted as 
<[>-]d.ddde<[>-]dd.
---|---
g float | Floating-point number using either the e or f 
format, whichever is more compact for the 
specified value and precision.
---|---
G float | Floating-point number using either the E or f 
format, whichever is more compact for the 
specified value and precision.
---|---
c char | A single character.
---|---
s * | A string of characters terminated by a null 
character ("\0").
---|---
p * | A generic pointer formatted as t:aaaa where t is 
the memory type and aaaa is the hexadecimal 
address.
---|---

**Note**

The optional characters l or L may immediately precede the type 
character to respectively specify long types for d, i, u, o, x, and X.

The optional characters b or B may immediately precede the type 
character to respectively specify char types for d, i, u, o, x, and X.

Characters following a percent sign that are not recognized as a format 
specification are treated as ordinary characters. For example, "%" writes 
a single percent sign to the output stream.

The **flags** field is a single character used to justify the output and to print 
+/- signs and blanks, decimal points, and octal and hexadecimal prefixes, 
as shown in the following table.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left justify the output in the specified field width.</td>
</tr>
</tbody>
</table>
| +    | Prefix the output value with a + or - sign if the output is a 
signed type. |
| blank (‘ ’) | Prefix the output value with a blank if it is a signed positive 
value. Otherwise, no blank is prefixed. |
| #    | Prefixes a non-zero output value with 0, 0x, or 0X when used 
with o, x, and X field types, respectively. 
When used with the e, E, f, g, and G field types, the # flag 
forces the output value to include a decimal point. 
The # flag is ignored in all other cases. |

The **width** field is a non-negative number that specifies the minimum 
number of characters printed. If the number of characters in the output 
value is less than width, blanks are added on the left (by default) or right
(when the - flag is specified) to pad to the minimum width. If width is prefixed with a '0', zeros are padded instead of blanks. The width field never truncates the output. If the length of the output value exceeds the specified width, all characters are output.

The width field may be an asterisk ('*'), in which case an int argument from the argument list provides the width value. Specifying a 'b' in front of the asterisk specifies that the argument is an unsigned char.

The precision field is a non-negative number that specifies the number of characters to print, the number of significant digits, or the number of decimal places. The precision field can cause truncation or rounding of the output value in the case of a floating-point number as specified in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Precision Field Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d,u,o,x,X</td>
<td>The precision field specifies the minimum number of digits that are included in the output value. Digits are not truncated if the number of digits in the argument exceeds that defined in the precision field. If the number of digits in the argument is less than the precision field, the output value is padded on the left with zeros.</td>
</tr>
<tr>
<td>f</td>
<td>The precision field specifies the number of digits to the right of the decimal point. The last digit is rounded.</td>
</tr>
<tr>
<td>e,E</td>
<td>The precision field specifies the number of digits to the right of the decimal point. The last digit is rounded.</td>
</tr>
<tr>
<td>g,G</td>
<td>The precision field specifies the maximum number of significant digits in the output value.</td>
</tr>
<tr>
<td>s</td>
<td>The precision field specifies the maximum number of characters in the output value. Excess characters are not output.</td>
</tr>
<tr>
<td>c,p</td>
<td>The precision field has no effect on these field types.</td>
</tr>
</tbody>
</table>

The precision field may be an asterisk ('*'), in which case an int argument from the argument list provides the value. Specifying a 'b' in front of the asterisk specifies that the argument is an unsigned char.

**Note**

You must ensure that the argument type matches that of the format specification. You may use type casts to ensure that the proper type is passed to printf.

This function is implementation-specific and is based on the operation of the _getkey and putchar functions. These functions, as provided in the standard library, read and write characters using the microcontroller's serial port. Custom functions may use other I/O devices.

The total number of bytes that may be passed to this function is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in SMALL or COMPACT model. A maximum of 40 bytes may be passed in LARGE model.

**Return Value** The printf function returns the number of characters actually written to the
Example

```c
#include <stdio.h>

void tst_printf (void) {
  char a = 1;
  int b = 12365;
  long c = 0x7FFFFFFF;

  unsigned char x = 'A';
  unsigned int y = 54321;
  unsigned long z = 0x4A6F6E00;

  float f = 10.0;
  float g = 22.95;

  char buf [] = "Test String";
  char *p = buf;

  printf ("char %bd int %d long %ld\n", a, b, c);
  printf ("Uchar %bu Uint %u Ulong %lu\n", x, y, z);
  printf ("xchar %bx xint %x xlong %lx\n", x, y, z);
  printf ("String %s is at address %p\n", buf, p);
  printf ("%f != %g\n", f, g);
  printf ("%*f != %*g\n", (int)8, f, (int)8, g);
}
```
The `scanf` function reads data from the input stream using the `getchar` routine. Data input are stored in the locations specified by `argument` according to the format string `fmtstr`. Each `argument` must be a pointer to a variable that corresponds to the type defined in `fmtstr`. The type controls the interpretation of the input data. The `fmtstr` may be composed of one or more whitespace characters, non-whitespace characters, and format specifications.

Whitespace characters, blank (' '), tab ('\t'), or newline ('\n'), cause `scanf` to skip whitespace characters in the input stream. A single whitespace character in the format string matches 0 or more whitespace characters in the input stream.

Non-whitespace characters, with the exception of the percent sign ('%'), cause `scanf` to read but not store a matching character from the input stream. The `scanf` function terminates if the next character in the input stream does not match the specified non-whitespace character.

Format specifications begin with a percent sign ('%') and cause `scanf` to read and convert characters from the input stream to the specified type values. The converted value is stored to an `argument` from the parameter list. Characters following a percent sign that are not recognized as a format specification are treated as ordinary characters. For example, `%%` matches a single percent sign in the input stream.

The format string is read from left to right. Characters that are not part of the format specifications must match characters in the input stream. These characters are read from the input stream but are discarded and not stored. If a character in the input stream conflicts with the format string, `scanf` terminates. Any conflicting characters remain in the input stream.

The first format specification encountered in the format string references the first `argument` after `fmtstr`. The `scanf` function converts input characters and stores the value using the format specification. The second format specification accesses the second `argument` after `fmtstr`, and so on. If there are more `arguments` than format specifications, the extra `arguments` are ignored. Results are unpredictable if there are not enough `arguments` for the format specifications.

Values in the input stream are called input fields and are delimited by whitespace characters. When converting input fields, `scanf` ends a conversion for an argument when a whitespace character or another unrecognized character is encountered.

Format specifications have the following format:
Each field in the format specification can be a single character or a number which specifies a particular format option.

The type field is where a single character specifies whether input characters are interpreted as a character, string, or number. This field can be any one of the characters in the following table.

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument Type</th>
<th>Input Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>int *</td>
<td>Signed decimal number.</td>
</tr>
<tr>
<td>i</td>
<td>int *</td>
<td>Signed decimal, hexadecimal, or octal integer.</td>
</tr>
<tr>
<td>u</td>
<td>unsigned int *</td>
<td>Unsigned decimal number.</td>
</tr>
<tr>
<td>o</td>
<td>unsigned int *</td>
<td>Unsigned octal number.</td>
</tr>
<tr>
<td>x</td>
<td>unsigned int *</td>
<td>Unsigned hexadecimal number.</td>
</tr>
<tr>
<td>e</td>
<td>float *</td>
<td>Floating-point number.</td>
</tr>
<tr>
<td>f</td>
<td>float *</td>
<td>Floating-point number.</td>
</tr>
<tr>
<td>g</td>
<td>float *</td>
<td>Floating-point number.</td>
</tr>
<tr>
<td>c</td>
<td>char *</td>
<td>A single character.</td>
</tr>
<tr>
<td>s</td>
<td>char *</td>
<td>A string of characters terminated by whitespace.</td>
</tr>
</tbody>
</table>

An asterisk ("*"’) as the first character of a format specification causes the input field to be scanned but not stored. The asterisk suppresses assignment of the format specification.

The width field is a non-negative number that specifies the maximum number of characters read from the input stream. No more than width characters are read and converted for the corresponding argument. However, fewer than width characters may be read if a whitespace or other unrecognized character is encountered first.

The optional characters b, h, and l may immediately precede the type character to respectively specify char, short, or long versions of the integer types d, i, u, o, and x.

**Note**

This function is implementation-specific and is based on the operation of the `_getkey` and `putchar` functions. These functions, as provided in the standard library, read and write characters using the microcontroller’s serial port. Custom functions may use other I/O devices.

The total number of bytes that may be passed to this function is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in SMALL or COMPACT model. A maximum of 40 bytes may be passed in LARGE model.

Problems may occur in some instances when the `_getkey` function requires overlayable data memory. Refer to the Keil Knowledgebase Article for more information.
The `scanf` function returns the number of input fields that were successfully converted. An EOF is returned if an error is encountered.

### See Also
- `gets`, `printf`, `printf517`, `putchar`, `puts`, `scanf517`, `sprintf517`, `sscanf`, `sscanf517`, `vprintf`, `vsprintf`

### Example
```c
#include <stdio.h>

void tst_scanf (void) {
    char a;
    int b;
    long c;

    unsigned char x;
    unsigned int y;
    unsigned long z;

    float f,g;
    char d, buf [10];
    int argsread;

    printf ("Enter a signed byte, int, and long\n");
    argsread = scanf ("%bd %d %ld", &a, &b, &c);
    printf ("%d arguments read\n", argsread);

    printf ("Enter an unsigned byte, int, and long\n");
    argsread = scanf ("%bu %u %lu", &x, &y, &z);
    printf ("%d arguments read\n", argsread);

    printf ("Enter a character and a string\n");
    argsread = scanf ("%c %9s", &d, buf);
    printf ("%d arguments read\n", argsread);
}
```
printf ("Enter two floating-point numbers\n");

argsread = scanf ("%f %f", &f, &g);

printf ("%d arguments read\n", argsread);

}