

---

## nanoprintf



nanoprintf is an unencumbered implementation of `snprintf` and `vsnprintf` for embedded systems that, when fully enabled, aim for C11 standard compliance. The primary exceptions are **double** (they get casted to **float**), scientific notation (`%e`, `%g`, `%a`), and the conversions that require `wcrtomb` to exist. C23 binary integer output is optionally supported as per N2630. Safety extensions for `snprintf` and `vsnprintf` can be optionally configured to return trimmed or fully-empty strings on buffer overflow events.

Additionally, nanoprintf can be used to parse printf-style format strings to extract the various parameters and conversion specifiers, without doing any actual text formatting.

nanoprintf makes no memory allocations and uses less than 100 bytes of stack. It compiles to between ~760-2500 bytes of object code on a Cortex-M0 architecture, depending on configuration.

All code is written in a minimal dialect of C99 for maximal compiler compatibility, compiles cleanly at the highest warning levels on clang + gcc + msvc, raises no issues from UBSan or Asan, and is exhaustively tested on 32-bit and 64-bit architectures. nanoprintf does include C standard headers but only uses them for C99 types and argument lists; no calls are made into `stdlib` / `libc`, with the exception of any internal double-to-float conversion ABI calls your compiler might emit. As usual, some Windows-specific headers are required if you're compiling natively for msvc.

nanoprintf is a single header file in the style of the stb libraries. The rest of the repository is tests and scaffolding and not required for use.

nanoprintf is statically configurable so users can find a balance between size, compiler requirements, and feature set. Floating point conversion, "large" length modifiers, and size write-back are all configurable and are only compiled if explicitly requested, see Configuration for details.

### Usage

Add the following code to one of your source files to compile the nanoprintf implementation:

```
1 // define your nanoprintf configuration macros here (see "Configuration
   " below)
2 #define NANOPRINTF_IMPLEMENTATION
3 #include "path/to/nanoprintf.h"
```

Then, in any file where you want to use nanoprintf, simply include the header and call the `npf_` functions:

---

```
1 #include "nanoprintf.h"
2
3 void print_to_uart(void) {
4     npf_pprintf(&my_uart_putc, NULL, "Hello %s%c %d %u %f\n", "worl", 'd'
5         , 1, 2, 3.f);
6 }
7 void print_to_buf(void *buf, unsigned len) {
8     npf_snprintf(buf, len, "Hello %s", "world");
9 }
```

See the “Use nanoprintf directly” and “Wrap nanoprintf” examples for more details.

## Motivation

I wanted a single-file public-domain drop-in printf that came in at under 1KB in the minimal configuration (bootloaders etc), and under 3KB with the floating-point bells and whistles enabled.

In firmware work, I generally want stdio’s string formatting without the syscall or file descriptor layer requirements; they’re almost never needed in tiny systems where you want to log into small buffers or emit directly to a bus. Also, many embedded stdio implementations are larger or slower than they need to be- this is important for bootloader work. If you don’t need any of the syscalls or stdio bells + whistles, you can simply use nanoprintf and `nosys.specs` and slim down your build.

## Philosophy

This code is optimized for size, not readability or structure. Unfortunately modularity and “cleanliness” (whatever that means) adds overhead at this small scale, so most of the functionality and logic is pushed together into `npf_vpprintf`. This is not what normal embedded systems code should look like; it’s `#ifdef` soup and hard to make sense of, and I apologize if you have to spelunk around in the implementation. Hopefully the various tests will serve as guide rails if you hack around in it.

Alternately, perhaps you’re a significantly better programmer than I! In that case, please help me make this code smaller and cleaner without making the footprint larger, or nudge me in the right direction. :)

## API

nanoprintf has 4 main functions: \* `npf_snprintf`: Use like `snprintf`. \* `npf_vsnprintf`: Use like `vsnprintf` (`va_list` support). \* `npf_pprintf`: Use like `printf` with a per-character write callback (semihosting, UART, etc). \* `npf_vpprintf`: Use like `npf_pprintf` but takes a `va_list`.

---

The `pprintf` variations take a callback that receives the character to print and a user-provided context pointer.

Pass `NULL` or `nullptr` to `npf_[v]snprintf` to write nothing, and only return the length of the formatted string.

`nanoprintf` does *not* provide `printf` or `putchar` itself; those are seen as system-level services and `nanoprintf` is a utility library. `nanoprintf` is hopefully a good building block for rolling your own `printf`, though.

## Return Values

The `nanoprintf` functions all return the same value: the number of characters that were either sent to the callback (for `npf_pprintf`) or the number of characters that would have been written to the buffer provided sufficient space. The null-terminator 0 byte is not part of the count.

The C Standard allows for the `printf` functions to return negative values in case string or character encodings can not be performed, or if the output stream encounters EOF. Since `nanoprintf` is oblivious to OS resources like files, and does not support the `l` length modifier for `wchar_t` support, any runtime errors are either internal bugs (please report!) or incorrect usage. Because of this, `nanoprintf` only returns non-negative values representing how many bytes the formatted string contains (again, minus the null-terminator byte).

## Configuration

### Features

`nanoprintf` has the following static configuration flags.

- `NANOPRINTF_USE_FIELD_WIDTH_FORMAT_SPECIFIERS`: Set to 0 or 1. Enables field width specifiers.
- `NANOPRINTF_USE_PRECISION_FORMAT_SPECIFIERS`: Set to 0 or 1. Enables precision specifiers.
- `NANOPRINTF_USE_FLOAT_FORMAT_SPECIFIERS`: Set to 0 or 1. Enables floating-point specifiers.
- `NANOPRINTF_USE_LARGE_FORMAT_SPECIFIERS`: Set to 0 or 1. Enables oversized modifiers.
- `NANOPRINTF_USE_BINARY_FORMAT_SPECIFIERS`: Set to 0 or 1. Enables binary specifiers.

- 
- `NANOPRINTF_USE_WRITEBACK_FORMAT_SPECIFIERS`: Set to 0 or 1. Enables `%n` for write-back.
  - `NANOPRINTF_VISIBILITY_STATIC`: Optional define. Marks prototypes as `static` to sandbox `nanoprintf`.

If no configuration flags are specified, `nanoprintf` will default to “reasonable” embedded values in an attempt to be helpful: floats are enabled, but writeback, binary, and large formatters are disabled. If any configuration flags are explicitly specified, `nanoprintf` requires that all flags are explicitly specified.

If a disabled format specifier feature is used, no conversion will occur and the format specifier string simply will be printed instead.

## Sprintf Safety

By default, `npf_snprintf` and `npf_vsnprintf` behave according to the C Standard: the provided buffer will be filled but not overrun. If the string would have overrun the buffer, a null-terminator byte will be written to the final byte of the buffer. If the buffer is `null` or zero-sized, no bytes will be written.

If you define `NANOPRINTF_SNPRINTF_SAFE_EMPTY_STRING_ON_OVERFLOW` and your string is larger than your buffer, the *first* byte of the buffer will be overwritten with a null-terminator byte. This is similar in spirit to Microsoft’s `snprintf_s`.

In all cases, `nanoprintf` will return the number of bytes that would have been written to the buffer, had there been enough room. This value does not account for the null-terminator byte, in accordance with the C Standard.

## Thread Safety

`nanoprintf` uses only stack memory and no concurrency primitives, so internally it is oblivious to its execution environment. This makes it safe to call from multiple execution contexts concurrently, or to interrupt a `npf_` call with another `npf_` call (say, an ISR or something). If you use `npf_pprintf` concurrently with the same `npf_putc` target, it’s up to you to ensure correctness inside your callback. If you `npf_snprintf` from multiple threads to the same buffer, you will have an obvious data race.

## Formatting

Like `printf`, `nanoprintf` expects a conversion specification string of the following form:

---

[**flags**][**field width**][**.precision**][**length modifier**][**conversion specifier**]

- **Flags**

None or more of the following:

- 0: Pad the field with leading zero characters.
- -: Left-justify the conversion result in the field.
- +: Signed conversions always begin with + or - characters.
- : (space) A space character is inserted if the first converted character is not a sign.
- #: Writes extra characters (0x for hex, . for empty floats, '0' for empty octals, etc).

- **Field width** (if enabled)

A number that specifies the total field width for the conversion, adds padding. If field width is \*, the field width is read from the next vararg.

- **Precision** (if enabled)

Prefixed with a ., a number that specifies the precision of the number or string. If precision is \*, the precision is read from the next vararg.

- **Length modifier**

None or more of the following:

- h: Use **short** for integral and write-back vararg width.
- L: Use **long double** for float vararg width (note: it will then be casted down to **float**)
- l: Use **long, double**, or wide vararg width.
- hh: Use **char** for integral and write-back vararg width.
- ll: (large specifier) Use **long long** for integral and write-back vararg width.
- j: (large specifier) Use the [u] **intmax\_t** types for integral and write-back vararg width.
- z: (large specifier) Use the **size\_t** types for integral and write-back vararg width.
- t: (large specifier) Use the **ptrdiff\_t** types for integral and write-back vararg width.

- **Conversion specifier**

Exactly one of the following:

- %: Percent-sign literal
- c: Character
- s: Null-terminated strings
- i/d: Signed integers
- u: Unsigned integers

- 
- `o`: Unsigned octal integers
  - `x` / `X`: Unsigned hexadecimal integers
  - `p`: Pointers
  - `n`: Write the number of bytes written to the pointer vararg
  - `f`/`F`: Floating-point decimal
  - `e`/`E`: Floating-point scientific (unimplemented, prints float decimal)
  - `g`/`G`: Floating-point shortest (unimplemented, prints float decimal)
  - `a`/`A`: Floating-point hex (unimplemented, prints float decimal)
  - `b`/`B`: Binary integers

## Floating Point

Floating point conversion is performed by extracting the value into 64:64 fixed-point with an extra field that specifies the number of leading zero fractional digits before the first nonzero digit. This is done for simplicity, speed, and code footprint.

Because the float -> fixed code operates on the raw float value bits, no floating point operations are performed. This allows `nanoprintf` to efficiently format floats on soft-float architectures like Cortex-M0, and to function identically with or without optimizations like “fast math”. Despite `nano` in the name, there’s no way to do away with double entirely, since the C language standard says that floats are promoted to double any time they’re passed into variadic argument lists. `nanoprintf` casts all doubles back down to floats before doing any conversions. No other single- or double- precision operations are performed.

The `%e/%E`, `%a/%A`, and `%g/%G` specifiers are parsed but not formatted. If used, the output will be identical to if `%f/%F` was used. Pull requests welcome! :)

## Limitations

No wide-character support exists: the `%lc` and `%ls` fields require that the arg be converted to a char array as if by a call to `wcrtomb`. When locale and character set conversions get involved, it’s hard to keep the name “nano”. Accordingly, `%lc` and `%ls` behave like `%c` and `%s`, respectively.

Currently the only supported float conversions are the decimal forms: `%f` and `%F`. Pull requests welcome!

---

## Measurement

The CI build is set up to use gcc and nm to measure the compiled size of every pull request. See the Presubmit Checks “size reports” job output for recent runs.

The following size measurements are taken against the Cortex-M0 build.

```
1 Configuration "Minimal":
2 arm-none-eabi-gcc -c -x c -Os -I/__/w/nanoprintf/nanoprintf -o npf.o -
    mcpu=cortex-m0 -DNANOPRINTF_IMPLEMENTATION -
    DNANOPRINTF_USE_FIELD_WIDTH_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_PRECISION_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_FLOAT_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_LARGE_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_BINARY_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_WRITEBACK_FORMAT_SPECIFIERS=0 -
3 arm-none-eabi-nm --print-size --size-sort npf.o
4 00000014 00000002 t npf_bufputc_nop
5 00000016 00000010 t npf_putc_cnt
6 00000000 00000014 t npf_bufputc
7 00000298 00000016 T npf_pprintf
8 000002e0 00000016 T npf_snprintf
9 000002ae 00000032 T npf_vsnprintf
10 00000026 00000272 T npf_vpprintf
11 Total size: 0x2f6 (758) bytes
12
13 Configuration "Binary":
14 arm-none-eabi-gcc -c -x c -Os -I/__/w/nanoprintf/nanoprintf -o npf.o -
    mcpu=cortex-m0 -DNANOPRINTF_IMPLEMENTATION -
    DNANOPRINTF_USE_FIELD_WIDTH_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_PRECISION_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_FLOAT_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_LARGE_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_BINARY_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_WRITEBACK_FORMAT_SPECIFIERS=0 -
15 arm-none-eabi-nm --print-size --size-sort npf.o
16 00000014 00000002 t npf_bufputc_nop
17 00000016 00000010 t npf_putc_cnt
18 00000000 00000014 t npf_bufputc
19 000002de 00000016 T npf_pprintf
20 00000328 00000016 T npf_snprintf
21 000002f4 00000034 T npf_vsnprintf
22 00000026 000002b8 T npf_vpprintf
23 Total size: 0x33e (830) bytes
24
25 Configuration "Field Width + Precision":
26 arm-none-eabi-gcc -c -x c -Os -I/__/w/nanoprintf/nanoprintf -o npf.o -
    mcpu=cortex-m0 -DNANOPRINTF_IMPLEMENTATION -
    DNANOPRINTF_USE_FIELD_WIDTH_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_PRECISION_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_FLOAT_FORMAT_SPECIFIERS=0 -
```

---

```

        DNANOPRINTF_USE_LARGE_FORMAT_SPECIFIERS=0 -
        DNANOPRINTF_USE_BINARY_FORMAT_SPECIFIERS=0 -
        DNANOPRINTF_USE_WRITEBACK_FORMAT_SPECIFIERS=0 -
27 arm-none-eabi-nm --print-size --size-sort npf.o
28 00000014 00000002 t npf_bufputc_nop
29 00000016 00000010 t npf_putc_cnt
30 00000000 00000014 t npf_bufputc
31 00000546 00000016 T npf_pprintf
32 00000590 00000016 T npf_snprintf
33 0000055c 00000034 T npf_vsnprintf
34 00000026 00000520 T npf_vpprintf
35 Total size: 0x5a6 (1446) bytes
36
37 Configuration "Field Width + Precision + Binary":
38 arm-none-eabi-gcc -c -x c -Os -I/__/w/nanoprintf/nanoprintf -o npf.o -
    mcpu=cortex-m0 -DNANOPRINTF_IMPLEMENTATION -
    DNANOPRINTF_USE_FIELD_WIDTH_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_PRECISION_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_FLOAT_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_LARGE_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_BINARY_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_WRITEBACK_FORMAT_SPECIFIERS=0 -
39 arm-none-eabi-nm --print-size --size-sort npf.o
40 00000014 00000002 t npf_bufputc_nop
41 00000016 00000010 t npf_putc_cnt
42 00000000 00000014 t npf_bufputc
43 00000590 00000016 T npf_pprintf
44 000005d8 00000016 T npf_snprintf
45 000005a6 00000032 T npf_vsnprintf
46 00000026 0000056a T npf_vpprintf
47 Total size: 0x5ee (1518) bytes
48
49 Configuration "Float":
50 arm-none-eabi-gcc -c -x c -Os -I/__/w/nanoprintf/nanoprintf -o npf.o -
    mcpu=cortex-m0 -DNANOPRINTF_IMPLEMENTATION -
    DNANOPRINTF_USE_FIELD_WIDTH_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_PRECISION_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_FLOAT_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_LARGE_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_BINARY_FORMAT_SPECIFIERS=0 -
    DNANOPRINTF_USE_WRITEBACK_FORMAT_SPECIFIERS=0 -
51 arm-none-eabi-nm --print-size --size-sort npf.o
52 00000014 00000002 t npf_bufputc_nop
53 00000016 00000010 t npf_putc_cnt
54 00000000 00000014 t npf_bufputc
55 0000059c 00000016 T npf_pprintf
56 000005e4 00000016 T npf_snprintf
57 000005b2 00000032 T npf_vsnprintf
58 00000026 00000576 T npf_vpprintf
59 Total size: 0x5fa (1530) bytes
60

```

---

---

```

61 Configuration "Everything":
62 arm-none-eabi-gcc -c -x c -Os -I/__/w/nanoprintf/nanoprintf -o npf.o -
    mcpu=cortex-m0 -DNANOPRINTF_IMPLEMENTATION -
    DNANOPRINTF_USE_FIELD_WIDTH_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_PRECISION_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_FLOAT_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_LARGE_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_BINARY_FORMAT_SPECIFIERS=1 -
    DNANOPRINTF_USE_WRITEBACK_FORMAT_SPECIFIERS=1 -
63 arm-none-eabi-nm --print-size --size-sort npf.o
64 00000014 00000002 t npf_bufputc_nop
65 00000016 00000010 t npf_putc_cnt
66 00000000 00000014 t npf_bufputc
67 00000934 00000016 T npf_pprintf
68 0000097c 00000016 T npf_snprintf
69 0000094a 00000032 T npf_vsnprintf
70 00000026 0000090e T npf_vpprintf
71 Total size: 0x992 (2450) bytes

```

## Development

To get the environment and run tests:

1. Clone or fork this repository.
2. Run `./b` from the root (or `py -3 build.py` from the root, for Windows users)

This will build all of the unit, conformance, and compilation tests for your host environment. Any test failures will return a non-zero exit code.

The nanoprintf development environment uses cmake and ninja. If you have these in your path, `./b` will use them. If not, `./b` will download and deploy them into `path/to/your/nanoprintf/external`.

nanoprintf uses GitHub Actions for all continuous integration builds. The GitHub Linux builds use this Docker image from my Docker repository.

The matrix builds [Debug, Release] x [32-bit, 64-bit] x [Mac, Windows, Linux] x [gcc, clang, msvc], minus the 32-bit clang Mac configurations.

One test suite is a fork from the printf test suite, which is MIT licensed. It exists as a submodule for licensing purposes- nanoprintf is public domain, so this particular test suite is optional and excluded by default. To build it, retrieve it by updating submodules and add the `--paland` flag to your `./b` invocation. It is not required to use nanoprintf at all.

---

## Acknowledgments

I implemented Float-to-int conversion using the ideas from Wojciech Muła's float -> 64:64 fixed algorithm.

I ported the printf test suite to nanoprintf. It was originally from the mpaland printf project codebase but adopted and improved by Eyal Rozenberg and others. (Nanoprintf has many of its own tests, but these are also very thorough and very good!)

The binary implementation is based on the requirements specified by Jörg Wunsch's N2630 proposal, hopefully to be accepted into C23!