baeldung.com

A Guide to POSIX | Baeldung on Linux

12–15 minutes

---

1. Overview

In this tutorial, we'll get to know what POSIX is and why it came into being. Then, we'll take a look at some history behind POSIX and how it evolved. Afterward, we'll go through some of the important POSIX standards.

Finally, we'll see the operating systems that are either completely or partially POSIX compliant.

2. What Is POSIX?

POSIX stands for Portable Operating System Interface. It's a family of standards specified by IEEE for maintaining compatibility among operating systems. Therefore, **any software that conforms to POSIX standards should be compatible with other operating systems that adhere to the POSIX standards**.

For that reason, most of the tools we use on Linux and Unix-like operating systems behave almost the same. For example, if we use the *ps* command, it should behave the same under OpenBSD, Debian, and macOS.

2.1. A Little History

Early in the Information Age, programmers would write software for operating systems with different system interfaces and environments. Therefore, porting the software to other operating systems would require a lot of heavy tweaks and costs.

To overcome this problem, POSIX was born. **POSIX is the standardization of the original UNIX, which came back in 1988 to resolve issues not only between different UNIX variants but also Non-UNIX operating systems as well.**

The work on POSIX began in the early 1980s to standardized the rapidly developing UNIX system interface. It covers a variety of systems in concise terms. The important POSIX.1 standard became the internationally accepted standard ISO/IEC 9945-1:1990 while the POSIX.2 standard was internationally accepted as IEEE Std 1003.2-1992.

2.2. POSIX Versions

The family of standards that POSIX refers to is known as IEEE Std 1003.*n-yyyy*. The *n* can be replaced with the version number such as IEEE Std 1003.1. We can also refer to the IEEE Std 1003.1 as POSIX.1. **The current version of POSIX.1 is IEEE Std 1003.1-2017.**

POSIX 1003.1 is the base standard upon which the POSIX family of standards has been developed, and there are currently more than 20 standards and drafts under the POSIX umbrella.

POSIX.1 defines application portability, as well as the C interface and the behavior of system services for fundamental tasks such as process creation and termination, process environment manipulation, file and directory access, and simple I/O.

On the other hand, POSIX.2 describes the command interpreter, portable shell programming, user environment, and related utilities. Both standards are strongly influenced by existing UNIX practice and experience.

3. POSIX Defined Standards

In this section, we'll take a look at some of the essential POSIX standards. Additionally, we can use The Open Group Base Specifications Issue as an in-depth reference.

3.1. The C API

POSIX defines its standards in terms of the C language. Therefore, **programs are portable to other operating systems at the source code level**. Nonetheless, we can also implement it in any standardized language.

The POSIX C API adds more functions on top of the ANSI C

Standard for a number of aspects:

- File operations
- Processes, threads, shared memory, and scheduling parameters
- Networking
- Memory management
- Regular expressions

The complete description of the functions is defined in the POSIX headers.

3.2. General Concepts

In addition to the C API, POSIX also added rules for writing programs, like safety for the initialization of pointer types and concurrent executions. It also enforces rules for memory synchronization, such as restricting memory modification when it's already in use. Apart from that, it also states security mechanisms for directory protection and file access.

3.3. File Formats

POSIX defines rules for formatting strings that we use in files, standard output, standard error, and standard input. As an example, let's consider the description for an output string:

```
"<format>", <arg1>, ..., <argN>
```

The format can contain regular characters, escape sequence characters, and conversion specifications. The conversion specifications indicate the output format of the provided arguments and are prefixed by a percent symbol followed by argument type.

As an example, let's suppose we want to output a string that contains today's date. We'll use the *printf* utility because it follows the POSIX file format standard:

```
$ printf "Today's Date: %d %s, %d" 18 September 2021
Today's Date: 18 September, 2021
```

The format specifies three conversion specifications: *%d*, *%s*, and *%d*. The *printf* utility processes these conversion specifications and substitutes them with the arguments.

3.4. Environment Variables

An environment variable is a variable that we can define in the environment file, which the login shell processes upon successful login. As a convention, the variable name should merely contain uppercase letters and an underscore. The name can also include a digit, although the POSIX standard doesn't recommend putting the digit at the start of the name.

For each environment variable, the value should only be a string of characters as defined in the portable character set. For instance, we can define the environment variable for our base user directory in the form of:

```
XDG_BASE_DIRECTORY="/home/user/"
```

We can name the environment variables however we like, although **we should avoid defining our own environment variables with names that conflict with the environment variables of standard utilities.** Additionally, our conforming implementation should be case-sensitive to environment variable names. For instance, the names *home* and *Home* are two different environment variables.

Our implementation should respect the reserved environment variables:

- *COLUMN* defines the width of the terminal screen.
- *HOME* defines the pathname of the user's home directory.
- *LOGNAME* defines the user's login name.
- *LINES* defines the user's preferred lines on the terminal screen.
- *PATH* defines binary colon-separated paths for executables.
- *PWD* defines the current working directory.
- *SHELL* defines the current shell in use.
- *TERM* defines the terminal type.

3.5. Locale

A locale defines the language and cultural convention that is used in the user environment. Each locale consists of categories that define the behavior of software components, such as date-time formatting, monetary formatting, and numeric formatting.

A program implementation shall conform to the POSIX locale, which is identical to the C locale. The program implementation should make use of the currently defined locale environment variables to retain coherence. **If there is no locale set, then the implementation should specify its POSIX compliant locale.**

The POSIX standard defines the following environment variables for each category:

- *LC_TYPE* for character classification
- *LC_COLLATE* defines the order for characters
- *LC_MONETARY* for monetary formatting
- *LC_NUMERIC* for formatting numbers
- *LC_TIME* for date and time formatting
- *LC_MESSAGES* for program messages such as information messages and logs

3.6. Character Set

A character set is a collection of characters that consists of codes and bit patterns for each character. **As we know, computers only understand binary, so a character set represents the symbols that computers can process. For that reason, we need a standard character set that conforms to the one defined by POSIX.**

POSIX recommends that our implementation should contain at least one character set and a portable character set. The first eight entries in the character set shall be control characters. The POSIX locale should include at least 256 characters from both portable and non-portable character sets.

3.7. Regular Expressions

A regular expression, or RE, is a string of characters that defines a search pattern for finding text. **The standard C library implements REs and is used by programs such as *awk, sed*, and *grep* as a backend.**

POSIX conforming implementations can make use of Basic Regular Expressions (BRE) or Extended Regular Expressions (ERE). **BRE provides basic notations for searching text, while ERE supports more notations.** Most POSIX conforming utilities rely heavily on BRE, although advanced text manipulation utilities also support ERE.

Moreover, several requirements apply to both BRE and ERE:

- BRE and ERE should operate on a string of characters that ends with a *NUL* character.
- The literal *escape sequence* and *newline* character produce an undefined result. Therefore, our programs should treat them as ordinary characters.
- POSIX does not permit the use of an explicit *NUL* character in the REs or the text to be matched.
- Our implementation should be able to perform a case-insensitive search by default.
- The length of our REs should not exceed 256 bytes in length.

3.8. Directory Structure

Most major Linux distributions conform to the Filesystem Hierarchy Standard (FHS). FHS defines a configurable tree-like directory structure. The first directory in the hierarchy is the *root* directory, and all the other directories, files, and special files branch out from it.

The *tree* utility can give us a better look at the directory structure:

```
$ tree / -d -L 1
/
├── bin -> usr/bin
├── boot
├── dev
├── etc
├── home
├── lib64 -> usr/lib
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin -> usr/bin
├── sys
├── usr
└── var
```

While the hierarchy is configurable, our program shouldn't create files or directories under the *root* and */dev* directories. However, it allows for creating files and directories under any user directory, such as the XDG Base Directory. Additionally, we can also use the */tmp* directory to create temporary files and directories.

3.9. Utilities

When we get familiar with the utilities in the UNIX/Linux environment, we can see that most utilities behave the same. For instance, we know that the *-h* option prints a help text for almost every UNIX/Linux utility. This consistency owes to the conventions described by POSIX. **POSIX defines several conventions for programmers about how we should implement our utility programs.**

POSIX recommends that we implement the following argument syntax in our utility programs:

```
utility_name [-a][-b][-c option_argument]
    [-d|-e][-f[option_argument]][operand...]
<parameter name>
```

Let's break it down:

- *utility_name* is the name of our utility followed by options and arguments.
- Items inside the square brackets are optional, and we can omit them.
- The *-a* and *-b* are flags that either enable or disable program features.
- The *-c* option expects an argument with a *blank character* in between.
- The *-d* and *-e* options separated by a pipe specify that both options

are mutually exclusive.

- The *-f* option specifies that the option and the option-argument should be placed together without a *blank character* in between, and the option without argument shall not treat the next argument as the option argument.
- The *operand* can be anything that the utility processes, such as a text file.
- The ellipsis (…) after the *operand* indicates that we can input zero or more operands to the utility.
- Items inside the angle brackets need to be substituted by actual values.

  Additionally, we can omit the brackets for the options that are required by the utility. As for complex utilities with a lot of options, we can group the options:

```
utility_name [-axyDnPo][-l arg][operand]
```

Apart from the utility syntax, POSIX encourages us to use a set of guidelines when implementing utilities.

4. Operating Systems and POSIX Compliancy

4.1. Linux

It's certainly possible to create a Linux-based operating system that is entirely POSIX compliant. EulerOS is a good example of that. However, **most modern programs, especially closed-source software, conform to the standard either partially or not at all**.

As an example, the *bash* shell used to be completely POSIX compliant. The recent versions of *bash*, however, don't conform to the POSIX standard by default. **So, one can say that most Linux distributions are partially POSIX-compliant.**

4.3. Darwin

Darwin is the core set for Apple's operating systems, such as macOS and iOS. It is partially POSIX compliant. However, **the recent releases of macOS are completely POSIX compliant.**

4.4. Windows NT

Microsoft Windows doesn’t conform to the standard at all because its whole design is completely different than UNIX-like operating systems. However, **we can set up a POSIX compliant environment by using the WSL compatibility layer or Cygwin**.

4.5. Others

A few proprietary operating systems are POSIX-certified:

- AIX
- HP-UX
- Oracle Solaris

5. Conclusion

In this article, we touched on the background of POSIX and how it was established. Afterward, we went through the essential standards that POSIX defines in its latest IEEE Std 1003.1 2017 issue.

Finally, we briefly discussed the current state of operating systems regarding their POSIX compliance.