IEEE Standards Interpretation for IEEE Std 1003.1™-1990 IEEE Standard for Information Technology--Portable Operating System Interfaces (POSIX®)

Interpretation Request #34

Topic: portable use of POSIX constants **Relevant Sections:** 2.9.4 **Classification:** Editorial defect

A question has been raised with regard to the portable use of the "#if" ANSI C construct in POSIX.1 portable applications when used with the symbols defined in ISO/IEC 9945-1, page 39, Table 2-11.

Ex:
```
#ifdef _POSIX_VDISABLE
#if _POSIX_VDISABLE == -1
...
...
#endif
#endif
```

Does the example provide acceptable code for POSIX.1 "portable" conforming applications? In other words, will the #if in the example above always compile without error? We believe the correct response is YES.

Rationale:
POSIX.1, page 39, line 1148 says:
"If any of the constants in Table 2-11" ... We interpret The term "constant" to mean a constrained "integral constant expression" which allows the identifiers in Table 2-11 to be used with the syntax "#if" in ANSI C constructs. In other words, using the terminology of Section 3.8 of the ANSI C Standard, an identifier in Table 2-11 must have a replacement list. In this case, the replacement list is a constrained integral constant expression (see Section 3.8.1 of the ANSI C Standard). The language of the ANSI C

Standard gives a more precise specification of the same concepts as described in Kernighan and Richie (1978) Section 12.3 and Section 15. Thus, this syntax is also present in common C. This interpretation also aligns directly with drafts P1003.1LIS/D2 and P1003.16/D2.

P1003.1LIS/D2, page 78, lines 2598-2599.
"language bindings shall specify how an application can distinguish these cases at compile time."

POSIX.1, page 111, lines 1005-1007. P1003.16/D2, page 134, lines 788-791.
"The value returned shall not be more restrictive than the corresponding value DESCRIBED TO THE APPLICATION WHEN IT WAS COMPILED with the implementation's <limits.h> or <unistd.h>." (emphasis added as capitalization)

This language says that at compilation time, the identifiers of Table 2-11 when defined in the header <unistd.h> are available for interrogation. Furthermore, the fact that Section 2.9.3 of POSIX.1 refers to "Compile Time Symbolic Constants" and Section 2.9.4 refers to "Execution-Time Symbolic Constants" does not imply that the identifiers in Table 2-11 are NOT portably "usable" at compile time. We feel that the reason for differentiating the identifiers in Table 2-10 and Table 2-11 is to highlight the fact that the identifiers in Table 2-11 need not be specified by an implementation at compile time since they can always be obtained from pathconf() and fpathconf().

Nevertheless, for those implementations where the value of an identifier in Table 2-11 is included in the header, the usefulness of this header value at RUN-TIME is limited. Even though it indicates the value of the identifier for all applicable files, the pathconf() code, to check for each applicable file, must be an integral part of the portable application even when this pathconf() code is not executed as a result of obtaining at run-time the identifier value from the header.

By using these values at COMPILE-TIME, a portable POSIX.1 application can avoid loading all pathconf() related code associated with a symbol in Table 2-11 when the symbol is defined. This allows some credence to the existence of these symbols in the header.

Interpretation Response
The example code in the request is not acceptable for a POSIX.1 conforming portable application. In other words, the standard does not require a conforming implementation to compile the #if in the example without error.

Rationale for Interpretation
The standard makes no requirement that the constant _POSIX_VDISABLE be a preprocessor number. The requirements relating this constant in section 2.9.4 relate only to use at execution time. It is understandable why an application might like to be able to use _POSIX_VDISABLE as a preprocessor constant. The wording in section 2.9.4:

If any of the constants in Table 2-11 are defined to have value -1 in the header .... can suggest, on casual reading, code like the following to minimize size and optimize efficiency for each implementation:

```
#ifdef _POSIX_VDISABLE
#if _POSIX_VDISABLE == -1
     /* code that assumes no vdisable capability */
#else
    /* code that assumes vdisable capability */
#endif
#else
   /* code that uses pathconf() to determine vdisable capability */
#endif
```

However, there is no wording in the standard to actually back up that suggestion, and silence on the part of the standard means no requirement.

There are reasons why an implementor might want to define a value that is not a pre-processor number, such as including a type cast to avoid problems in comparing the value to a member of the c_cc array member of a termios struct (which is constrained by the standard to be an unsigned integer type). Since no wording in the standard prohibits this, it is implicitly permitted.

Thus, rather than the above fragment, an implementation could include code like:

```
#ifdef _POSIX_VDISABLE
      if (_POSIX_VDISABLE == -1) {
         /* code that assumes no vdisable capability */
      } else {
        /* code that assumes vdisable capability */ }
#else
       /* code that uses pathconf() to determine vdisable capability */
#endif
```

Of course it is generally simplest, though potentially less efficient, to just write the code that uses pathconf().