POSIX/C Liaison Report

Document: WG14 N1257
Date: 2007-09-07
Author: Nick Stoughton

1. Null Pointer Issues

The POSIX committee notes the current definition of "null pointer" in the C standard (6.3.2.3p3):

> An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

The current draft of the revised POSIX standard rewords this as:

> The value that is obtained by converting the number 0 into a pointer; for example, (void *) 0. The C language guarantees that this value does not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

The crucial part of the difference here is the desired requirement in POSIX for the value 0 to be converted to a pointer. It is not felt sufficient, especially in the light of recently developed ABIs for newer architectures (varargs promoted to integer), to simply require the value 0 to a valid null pointer.

While the POSIX committee is concerned about the "integer constant expression" part of the requirement <<e.g. what should happen for setlocale(LC_ALL, 1/2)>>, this is felt to be a lesser part of the problem than the requirement that the value be converted to a pointer. So, at very least, the POSIX committee would like to require that the definition of "null pointer constant" be updated:

> An integer constant expression with the value 0 cast to a pointer type such as void *, is called a null pointer constant.

(note, the rest of the description on what a "null pointer" is will also need updating, but the Austin Group felt that the C committee would be the better place to wordsmith such a definition).

Note the current draft of POSIX requires NULL to be defined as:
NULL Null pointer constant. The macro shall expand to an integer constant expression
     with the value 0 cast to type void *.

2. Thread APIs

The Austin Group notes the sentiment expressed in WG14 N1167, and N1196 for the C++ committee to take the lead in concurrency issues. The Austin Group strongly agrees that the alignment, memory model, atomic types and operations, and thread local storage proposals from the C++ committee are applicable and adaptable to C, and encourages WG 14 to adopt a C binding to these proposals. However, it notes that the interfaces proposed in the current Thread API proposal (N2320) are not compatible with the long established C style existing practice in POSIX, and proposes that should WG 14 desire to add thread APIs to their standard that the POSIX pthread* interfaces be the primary starting point. These interfaces fit the existing practice criteria required for new C proposals (while the incompatible inventions from WG 21 does not).

The Austin Group recommends that WG14 does not add any threading APIs. However, the Austin Group is prepared to make a proposal to WG 14 to add pthread interfaces to their next revision if WG 14 indicate that they wish to go in that direction. The Austin Group further notes that should C require some form of the C++ thread interfaces, it is likely to meet strong resistance from the Austin Group.

3. Problem with errno.h

An inconsistency has been spotted with the words for errno.h (TC2 page 198, 7.5 p1). In p1, <errno.h> defines several macros. In p2 "The macros are ... errno". But later in p2 "It is unspecified whether errno is a macro or an identifier declared with external linkage." So is it a macro or not?

In POSIX, where multi-threading will be mandatory in the next revision, it is typically a macro, though by use of compiler extensions, it could be implemented as thread local storage. Therefore we preserve the "unspecified whether or not..." wording.

Proposed new wording for a C revision:

7.5 Errors <errno.h>

The header <errno.h> defines several macros, all relating to the reporting of error conditions.

The macros are
```
        EDOM
        EILSEQ
        ERANGE
```
which expand to integer constant expressions with type int, distinct positive values, and which are suitable for use in #if preprocessing directives.

The <errno.h> header also provides a declaration or definition for errno. The symbol errno shall expand to a modifiable lvalue of type int. It is unspecified whether errno is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name errno, the behavior is undefined.

The value of errno is zero at program start-up, but is never set to zero by any library function. The value of errno may be set to nonzero by a library function call whether or not there is an error, provided the use of errno is not documented in the description of the function in this International Standard.

Additional macro definitions, beginning with E and a digit or E and an uppercase letter, may also be specified by the implementation.