	0000327	Base Definitions and Headers	**Objection**	Under Review(ajosey)	Open
☐	0000258	System Interfaces	Comment	Interpretation Required (ajosey)	Accepted As Marked
☐	0000249	Shell and Utilities	**Objection**	Under Review(ajosey)	Open
☐	0000211	Base Definitions and Headers	**Objection**	Resolved (ajosey)	Accepted
☐	0000174	System Interfaces	**Objection**	Resolved (ajosey)	Accepted As Marked
☐	0000173	System Interfaces	**Objection**	Interpretation Required (ajosey)	Accepted
☐	0000170	System Interfaces	**Objection**	Resolved (ajosey)	Rejected
☐	0000162	System Interfaces	Comment	Under Review(ajosey)	Open
☐	0000148	System Interfaces	**Objection**	Resolved (ajosey)	Accepted As Marked
☐	0000110	System Interfaces	**Objection**	Resolved (ajosey)	Accepted As Marked
☐	0000109	System Interfaces	Editorial	Interpretation Required (ajosey)	Accepted As Marked
☐	0000105	System Interfaces	**Objection**	Resolved (ajosey)	Accepted
☐	0000090	System Interfaces	**Objection**	Interpretation Required (ajosey)	Accepted As Marked
☐	0000088	System Interfaces	Editorial	Resolved (ajosey)	Accepted
☐	0000086	System Interfaces	Comment	Resolved (ajosey)	Accepted

☐	0000074	System Interfaces	**Objection**	Interpretation Required (ajosey)	Accepted As Marked
☐	0000073	System Interfaces	Comment	Under Review(ajosey)	Open

0000073: wmemcmp C conflict?

STATUS:
Open, awaiting input from WG 14.

Description
This issue is for tracking purposes.

The following question is being discussed in the C committee at present, and highlights a difference between C-1990 with AMD-1 and C99. POSIX has followed the C89+AMD1 words, and so is possibly at odds with C99.

===> from Joseph Myers

When are wide string library functions required to handle values of type wchar_t that do not represent any value in the execution character set, and when does using such values with a library function result in undefined behavior?

Consider the following testcase as an example:

```
#include <stdlib.h>
#include <wchar.h>

wchar_t w0 = WCHAR_MIN;
wchar_t w1 = WCHAR_MAX;

int
main (void)
{
  if (wmemcmp (&w0, &w1, 1) < 0)
    return 0;
  else
    abort ();
}
```

Suppose that WCHAR_MIN and WCHAR_MAX do not both represent values in the execution character set. If the arguments to wmemcmp are valid, wmemcmp must return a value less than 0 because 7.24.4.4 says the comparison is done the same way as comparing integers of type wchar_t, so the program must execute successfully. With the GNU C Library, however, it aborts; wchar_t is UTF-32 but has a signed type so WCHAR_MIN is negative and does not represent a member of the execution character set.

C90 AMD1 had an explicit statement (7.16.4.6) that made clear that these inputs were valid (and so wmemcmp had to return a value less than 0 for the above example in C90 AMD1):

> These functions operate on arrays of type wchar_t whose size is specified by a separate count argument. These functions are not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid multibyte characters are not treated specially.

I cannot however find any equivalent statement in C99. Was this a deliberate change from AMD1, or a side-effect of how the functions were rearranged when added to C99?

POSIX repeats the above requirement from C90 AMD1, but I believe this is an accident of taking the specification from there originally and is not intended to impose any requirements beyond those of C99.

Much the same issue applies to wcscmp and wcsncmp, where the comparison semantics are specified but AMD1 has no mention of wide characters not corresponding to members of the execution character set, and in principle to other wcs* and wmem* functions that have no reason to need to consider the semantics of the characters they process (but are less likely than the comparison functions to have problems with the full set of wchar_t values in practice).

Desired Action

Await for decision from C and if necessary make whatever change to align with the emerging C standard.

Issue an interp to describe the discrepancy.

0000074: Pointer Types Problem

STATUS:

Approved Interpretation (No impact on WG14, for information only)

Description

In order to support the dynamic library functions (and dlsym() in particular), POSIX extends the C standard to require that a pointer to a function can be stored in a pointer to void.

This explicit extension opens the door to permit conversion (via other promises in the C standard) of a pointer to a data object into a pointer to a function, thus requiring code such as below

to work:

```
char buf[1024];
void *p;
int (*f)(void);

// assemble instructions into buf
p = buf;
f = p;
f();
```

In practice, many implementations will forbid execution of data

in this form, as it is frequently a source of security vulnerabilities.

Similarly, it need not necessarily be supported to try to examine the instructions that make a function by casting (or converting via a void *) a pointer-to-function into a pointer-to-object.

The intent is simply to permit dlsym to use a void * as its return type.

Agreed Action

Interpretation response

The standard states the requirements for pointer types , and conforming implementations must conform to this. However, concerns have been raised about this which are being referred to the sponsor.
Rationale:

None.

Notes to the Editor (not part of this interpretation):

After a lengthy discussion on the e-mail list based on the proposed resolution in Note: 0000129, the committee decided to revise the text. These changes solve:

1. the issue initially raised in this report,
2. the issue raised in 0000099 (which will be closed as a dup of this bug when this bug resolution is approved),
3. the issue raised in 0000100 (which will be closed as a dup of this bug when this bug resolution is approved),
4. some inconsistencies between the uses of the term object in this standard and in ISO C (as noted by Konrad in Note: 0000155 and Note: 0000156), and
5. the inconsistent use of the term symbol in this standard (in the rest of this standard, symbol refers either to a graphic representation of a character or to a symbolic constant; in the description of dlsym(), however, symbol refers to what is known by the term identifier in ISO C.

ISO/IEC JTC 1/SC 22/WG 14 N1529
Austin/501 Liaison Report
Nick Stoughton October 5, 2010

This note was discussed during the August 27, 2009 conference call and minor edits have made done in place (without creating a new note).

Delete subclause 2.12.3 from P541, L18881-18886.

Replace the NAME, SYNOPSIS, DESCRIPTION, RETURN VALUE, ERRORS, EXAMPLES, and APPLICATION USAGE sections of the dlclose() descripion on P728, L24469-24513 with:

NAME

dlclose -- close a symbol table handle

SYNOPSIS

```
#include <dlfcn.h>
int dlclose(void *handle);
```

DESCRIPTION

The dlclose() function shall inform the system that the symbol table handle specified by handle is no longer needed by the application.

An application writer may use dlclose() to make a statement of intent on the part of the process, but this statemen does not create any requirement upon the implementation. When the symbol table handle is closed, the implementation may unload the executable object files that were loaded by dlopen() when the symbol table handle was opened and those that were loaded by dlsym() when using the symbol table handle identified by handle. Once a symbol table handle has been closed, an application should assume that any symbols (function identifiers and data object identifiers) made visible using handle, are no longer available to the process.

Although a dlclose() operation is not required to remove any

functions or data objects from the address space, neither is an implementation prohibited from doing so. The only restriction on such a removal is that no function nor data object shall be removed to which references have been relocated, until or unless all such references are removed. For instance, an executable object file that had been loaded with a dlopen() operation specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in the processing of other relocatable objects--in such environments, an application may assume that no relocation, once made, shall be undone or remade unless the executable object file containing the relocated object has itself been removed.

RETURN VALUE

If the referenced symbol table handle was successfully closed, dlclose() shall return 0. If handle does not refer to an open symbol table handle or if the symbol table handle could not be closed, dlclose() shall return a non-zero value. More detailed diagnostic information shall be available through dlerror().

ERRORS

No errors are defined.

EXAMPLES

The following example illustrates use of dlopen() and dlclose():

```
 #include <dlfcn.h>
int    eret;
void   *mylib;
...
/* Open a dynamic library and then close it ... */

mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
```

```
    eret = dlclose(mylib);
    ...
```

APPLICATION USAGE

A conforming application should employ a symbol table handle returned from a dlopen() invocation only within a given scope bracketed by a dlopen() operation and the corresponding dlclose() operation. Implementations are free to use reference counting or other techniques such that multiple calls to dlopen() referencing the same executable object file may return a pointer to the same data object as the symbol table handle. Implementations are also free to reuse a handle. For these reasons, the value of a handle must be treated as an opaque data type by the application, used only in calls to dlsym() and dlclose().

Replace the NAME, SYNOPSIS, DESCRIPTION, and RETURN VALUE sections of the dlopen() descripion on P732-733, L24575-24666:

NAME

dlopen -- open a symbol table handle

SYNOPSIS

```
#include <dlfcn.h>
void *dlopen(const char *file, int mode);
```

DESCRIPTION

The dlopen() function shall make the symbols (function identifiers and data object identifiers) in the executable object file specified by file available to the calling program.

The class of executable object files eligible for this operation and the manner of their construction are implementation-defined, though typically such files are shared libraries or programs. Implementations may permit the construction of embedded dependencies in executable object files. In such cases, a dlopen() operation shall load those dependencies in addition to the executable object file specified by file. Implementations may also impose specific constraints on the construction of programs that can employ dlopen() and its related services. A successful dlopen() shall return a symbol table handle which the caller may use on subsequent calls to dlsym() and dlclose(). The value of this symbol table handle should not be interpreted in any way by the caller.

The file argument is used to construct a pathname to the executable object file. If file contains a <slash> character, the file argument is used as the pathname for the file. Otherwise, file is used in an implementation-defined manner to yield a pathname.

If file is a null pointer, dlopen() shall return a global symbol table handle for the currently running process image. This symbol table handle shall provide access to the symbols from an ordered set of executable object files consisting of the original program image file, any executable object files loaded at program start-up as specified by that process image file (for example, shared libraries), and the set of executable object files loaded using dlopen() operations with the RTLD_GLOBAL flag. As the latter set of executable object files can change during execution, the set of symbols made available by this

symbol table handle can also change dynamically.

Only a single copy of an executable object file shall be brought into the address space, even if dlopen() is invoked multiple times in reference to the executable object file, and even if different pathnames are used to reference the executable object file.

The mode parameter describes how dlopen() shall operate upon file with respect to the processing of relocations and the scope of visibility of the symbols provided within file. When an executable object file is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the executable object file is loaded. These references shall be relocated before the symbols can be accessed. The mode parameter governs when these relocations take place and may have the following values:

RTLD_LAZY Relocations shall be performed at an implementation-defined time, ranging from the time of the dlopen() call until the first reference to a given symbol occurs. Specifying RTLD_LAZY should improve performance on implementations supporting dynamic symbol binding since a process might not reference all of the symbols in an executable object file. And, for systems supporting dynamic symbol resolution for normal process execution, this behavior mimics the normal handling of process execution.

RTLD_NOW All necessary relocations shall be performed when the executable object file is first loaded. This may waste some processing if relocations are performed for symbols that are never

referenced. This behavior may be useful for applications that need to know that all symbols referenced during execution will be available before dlopen() returns.

Any executable object file loaded by dlopen() that requires relocations against global symbols can reference the symbols in the original process image file, any executable object files loaded at program start-up, from the initial process image itself, from any other executable object file included in the same dlopen() invocation, and any executable object files that were loaded in any dlopen() invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a dlopen() invocation, the mode parameter should be a bitwise-inclusive OR with one of the following values:

RTLD_GLOBAL The executable object file's symbols shall be made available for relocation processing of any other executable object file. In addition, symbol lookup using dlopen(NULL, mode) and an associated dlsym() allows executable object files loaded with this mode to be searched.
RTLD_LOCAL The executable object file's symbols shall not

be made available for relocation processing of
any other executable object file.

If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, the default behavior is unspecified.

If an executable object file is specified in multiple dlopen() invocations, mode is interpreted at each invocation.
If RTLD_NOW has been specified, all relocations
shall have been completed rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations irrelevant.

If RTLD_GLOBAL has been specified, the executable object file shall maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, as long as the executable object file remains in the address space (see dlclose()).

Symbols introduced into the process image through calls to dlopen() may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous dlopen() operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two such resolution orders are defined: load order and dependency order. Load order establishes an ordering among symbol definitions, such that the first definition loaded (including definitions from the process image file and any dependent executable object files loaded with it) has priority over executable object files added later (by dlopen()). Load ordering is used in relocation processing. Dependency ordering uses a breadth-first order starting with a given executable

object file, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol table handle obtained via a dlopen() operation with a null pointer as the file argument, dependency ordering is used by the dlsym() function. Load ordering is used in dlsym() operations upon the global symbol table handle.

When an executable object file is first made accessible via dlopen(), it and its dependent executable object files are added in dependency order. Once all the executable object files are added, relocations are performed using load order. Note that if an executable object file or its dependencies had been previously loaded, the load and dependency orders may yield different resolutions.

The symbols introduced by dlopen() operations and available through dlsym() are at a minimum those which are exported as identifiers of global scope by the executable object file. Typically such identifiers shall be those that were specified in (for example) C source code as having extern linkage. The precise manner in which an implementation constructs the set of exported symbols for an executable object file is implementation-defined.

RETURN VALUE

Upon successful completion, dlopen() shall return a symbol table handle. If file cannot be found, cannot be opened for reading, is not of an appropriate executable object file format for processing by dlopen(), or if an error occurs during the process

of loading file or relocating its symbolic references, dlopen() shall return a null pointer. More detailed diagnostic information shall be available through dlerror().

Replace the NAME, SYNOPSIS, DESCRIPTION, RETURN VALUE, ERRORS, EXAMPLES, APPLICATION USAGE, and RATIONALE sections of the dlsym() descripion on P735-736, L24689-24748 with:

NAME

dlsym -- get the address of a symbol from a symbol table handle

SYNOPSIS

#include <dlfcn.h>

void *dlsym(void *restrict handle, const char *restrict name);

DESCRIPTION

The dlsym() function shall obtain the address of a symbol (a function identifier or a data object identifier) defined in the symbol table identified by the handle argument. The handle argument is a symbol table handle returned from a call to dlopen() (and which has not since been released by a call to dlclose()), and name is the symbol's name as a character string. The return value from dlsym(), cast to a pointer to the type of the named symbol, can be used to call (in the case of a function) or access the contents of (in the case of a data object) the named symbol.

The dlsym() function shall search for the named symbol in the symbol table referenced by handle. If the symbol table was created with lazy loading (see RTLD_LAZY in dlopen()), load ordering shall be used in dlsym() operations to relocate executable object files needed to resolve the symbol. The

symbol resolution algorithm used shall be dependency order
as described in dlopen().

The RTLD_DEFAULT and RTLD_NEXT symbolic constants (which may be
defined in <dlfcn.h>) are reserved for future use as special
values that applications may be allowed to use for handle.

RETURN VALUE

Upon successful completion, if name names a function identifier,
dlsym() shall return the address of the function converted from
type pointer to function to type pointer to void; otherwise
dlsym() shall return the address of the data object associated
with the data object identfier named by name converted from a
pointer to the type of the data object to a pointer to void. If
handle does not refer to a valid symbol table handle or if the
symbol named by name cannot be found in the symbol table
associated with handle, dlsym() shall return a null pointer.
More detailed diagnostic information shall be available through
dlerror().

ERRORS

No errors are defined.

EXAMPLES

The following example shows how dlopen() and dlsym() can be used
to access either a function or a data object. For simplicity,
error checking has been omitted.

```
void   *handle;
int    (*fptr)(int),
       *iptr,
       result;        /* open the needed symbol table */
handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);
```

```
 /* find the address of the function my_function */
fptr = (int (*)(int))dlsym(handle, "my_function");
 /* find the address of the data object my_object */

iptr = (int *)dlsym(handle, "my_OBJ");
 /* invoke my_function, passing the value of my_OBJ as the parameter */
result = (*fptr)(*iptr);
```

APPLICATION USAGE

The following special purpose values for handle are reserved for future use and have the indicated meanings:

RTLD_DEFAULT The identifier lookup happens in the normal global scope; that is, a search for a identifier using handle would find the same definition as a direct use of this identifier in the program code.

RTLD_NEXT Specifies the next executable object file after this one that defines name. This one refers to the executable object file containing the invocation of dlsym(). The next executable object file is the one found upon the application of a load order symbol resolution algorithm (see dlopen()). The next symbol is either one of global scope (because it was introduced as part of the original process image or because it was added with a dlopen() operation including the RTLD_GLOBAL flag), or is in an executable object file that was included in the same dlopen() operation that loaded this

one.

The RTLD_NEXT flag is useful to navigate an
intentionally created hierarchy of
multiply-defined symbols created through
interposition. For example, if a program wished
to create an implementation of malloc() that
embedded some statistics gathering about memory
allocations, such an implementation could use
the real malloc() definition to perform the

memory allocation--and itself only embed the
necessary logic to implement the statistics

gathering function.

Note that conversion from a void * pointer to a function pointer
as in:

```
fptr = (int (*)(int))dlsym(handle, "my_function");
```

is not defined by the ISO C Standard. This standard requires
this conversion to work correctly on conforming implementations.

RATIONALE

None.

0000086: asctime rationale

STATUS:

Accepted (No impact on WG 14)

Description:

The asctime() rationale says:

> "The standard developers decided to mark the asctime() and asctime_r() functions obsolescent even though they are in the ISO C standard ..."

However, asctime_r() is not in ISO C.

Desired Action:

Change

> "even though they are in"

to

> "even though asctime() is in"

0000088: wcrtomb in wchar.h

STATUS

Accepted (No impact on WG14)

Description

Per C99, wcrtomb is declared in <wchar.h>, not <stdio.h>. XBD has the correct listings, only XSH needs adjustment.

Desired Action

At line 69147, replace

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

with

```
#include <wchar.h>
```

0000090: fscanf stds contradiction

STATUS

Accepted: Corresponding changes to C1x are expected.

Description

In austin-group-l 11809, Vincent Lefvre identified a conflict with the C Standard:

---- begin quote ----

I think there is a contradiction between POSIX.1-2008 and the ISO C standard concerning fscanf when an input failure occurs after the first conversion.

The ISO C standard (at least N1124 and N1336) says:

7.19.6.2 The fscanf function

4 The fscanf function executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

16 The fscanf function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

For instance, let us consider

```
n = fscanf (stdin, "%d %d", &x, &y);
```

where an input failure occurs *after* the first %d conversion (and before the second one). According to the C standard, n should be equal to 1 because
* the input failure occurred *after* the first conversion,
* 1 input item (exactly) has been assigned.

However POSIX.1-2008 says (page 934):

Upon successful completion, these functions shall return the number

> of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If any error occurs, EOF shall be returned, and errno shall be set to indicate the error. If a read error occurs, the error indicator for the stream shall be set.

I wonder what POSIX really means by "If any error occurs, EOF shall be returned [...]". I suppose this means "input failure" in C (as opposed to "matching failure"). But, as shown above, the C standard does the difference between these two kinds of failure only when the failure occurs before the first conversion (indeed one gets EOF in case of an input failure, and 0 in case of a matching failure).

---- end quote ----

An interpretation should be issued stating that POSIX defers to the C Standard here, and conforming implementations must behave as described in the C Standard. The POSIX text should be corrected in TC1 to remove the conflict.

During discussions it emerged that, by a strict reading, the phrase "before any conversion" is ambiguous, although the intention is clear (at least, it is to native English speakers). It was proposed to use the phrase "before the first successful conversion" instead, but this has a couple of problems:

1. If the error occurs before the first conversion, then that conversion never happens and cannot be thought of as "successful".

2. Even if the word "successful" is removed, it is still not clear

whether the error has to occur before reading the first byte of input that would be used in the conversion, or if the requirement also applies after at least one such byte has been read but with more bytes needed to finish the conversion.

I propose to use "without any successful conversions having been made" instead.

Desired Action

Change

> "If the input ends before the first matching failure or conversion, EOF shall be returned. If any error occurs, EOF shall be returned, [CX]and errno shall be set to indicate the error[/CX]. If a read error occurs, the error indicator for the stream shall be set."

to

> "If the input ends without any successful conversions having been made, and without a matching failure having occurred, EOF shall be returned. If an error occurs without any successful conversions

having been made, and without a matching failure having occurred, EOF shall be returned [CX]and errno shall be set to indicate the error[/CX]. If a read error occurs, the error indicator for the stream shall be set."

Make the same change to the fwscanf() page (P988 L33162).

Status

Interpretation response

The required behaviour of fscanf for this standard is dependent on the requirements of the ISO C standard and conforming implementations must behave as described in the C Standard. Concerns have been raised about the fscanf function which are being referred to the sponsor.

Rationale:

This is a conflict with the C Standard.

Notes to the Editor (not part of this interpretation):

Nick has progressed a proposal with WG14 which they appear to have accepted as an editorial. We agreed that since its been classified by WG14 as an editorial we can close this item.

Change from
"If the input ends before the first matching failure or conversion, EOF shall be returned. If any error occurs, EOF shall be returned, [CX]and errno shall be set to indicate the error[/CX]. If a read error occurs, the error indicator for the stream shall be set."
to
"If the input ends before the first conversion (if any) has

completed, and without a matching failure having occurred, EOF shall be returned. If an error occurs before the first conversion (if any) has completed , and without a matching failure having occurred, EOF shall be returned [CX]and errno shall be set to indicate the error[/CX]. If a read error occurs, the error indicator for the stream shall be set."

Add to Change History :
A change to the second sentence in RETURN VALUE
is made to align with expected wording
changes in the next revision of the C standard.

(recommended for TC1)

0000105: errno change on success

STATUS

Accepted (WG 14 should review)

Description

C99 says (7.5 paragraph 3) "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."

This implies that for all functions in C99 where the description documents the use of errno, the function can only set errno as per the description - it cannot change errno on success (unless its description says it can).

For some of these functions, POSIX has CX shading on the statement that the function does not change errno on success. This text should not have CX shading. Affected functions are: strtod(), strtof(), strtold(), strtol(), strtoll(), strtoul(), strtoull(), wcstod(), wcstof(), wcstold(), wcstol(), wcstoll(), wcstoul(), wcstoull().

There are also some functions for which POSIX omits a statement that it does not change errno on success. Affected functions are: fgetpos(), fsetpos(), ftell(), fgetwc(), fputwc(), signal(), mbrlen(), mbrtowc(), wcrtomb(), mbsrtowcs(), wcsrtombs().

(Note that, by a strict reading of C99, all the byte I/O and wide I/O functions are also affected. However, this matter has been raised with the C committee and it appears likely that a change will be made in C1x. Therefore I have not included changes for those functions.)

Desired Actiuon

Remove the CX shading from

> "The strtod() function shall not change the setting of errno if successful.
>
> Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set errno to 0, then call strtod(), strtof(), or strtold(), then check errno."

and remove the CX shading from similar text at

page 2044 line 64780 section strtol
page 2049 line 64884 section strtoul
page 2224 line 69996 section wcstod
page 2231 line 70204 section wcstol
page 2238 line 70365 section wcstoul

(In each case it is the entire CX shaded block that should be unshaded.)

Also for strtod(), strtol(), strtoul(), wcstod(), and wcstoul() change the first sentence of the affected text to match wcstol():

> "These functions shall not change the setting of errno if successful."

At
page 850 line 28241 section fgetpos
page 854 line 28343 section fgetwc
page 910 line 30406 section fputwc
page 940 line 31503 section fsetpos
page 1272 line 41808 section mbrlen
page 1274 line 41871 section mbrtowc
page 1277 line 41989 section mbsrtowcs
page 1937 line 61644 section signal

page 2195 line 69168 section wcrtomb
page 2219 line 69823 section wcsrtombs
add a new paragraph (substituting the relevant function name for "funcname") at the end of the DESCRIPTION:

> "The funcname() function shall not change the setting of errno if successful.

At page 956 line 32041 section ftell change

> "[CX]The ftello() function shall be equivalent to ftell(), except that the return value is of type off_t.[/CX]"

to

> "The ftell() function shall not change the setting of errno if successful.
>
> [CX]The ftello() function shall be equivalent to ftell(), except that the return value is of type off_t and the ftello() function may change the setting of errno if successful.[/CX]"

000109: mblen() not thread-safe

STATUS

Accepted. WG 14 should consider thread safety implications for C1x.

Description

mblen() and mbtowc() are expected to maintain an internal shift state from one call to the next. However, neither function is listed in 2.9.1, Thread-Safety, among functions that are not required to be thread-safe. Thus, the specification effectively requires that each function maintain a thread-local copy of the state. However, at least two recent quality implementations maintain a per-process state instead, and thus are not thread-safe. A discussion on austin-group-l indicates that the named functions are not, in fact, intended to be thread-safe.

Agreed Action

The standard states the requirements for thread safety of mblen() and mbtowc(), and conforming implementations must conform to this. However, concerns have been raised about this which are being referred to the sponsor."

Rationale:

None.

Notes to the Editor (not part of this interpretation):

Add mblen() and mbtowc() in subclause 2.9.1 in alphabetic order to the list of functions that need not be thread-safe in the table on P507, L17490-17510. Remove wcstombs() from the same table.

Change the list of functions that need not be thread-safe if called with a NULL ps argument in subclause 2.9.1 on P507, L17512 from:
"wcrtomb() and wcsrtombs()"
to:

"mbrlen(), mbrtowc(), mbsnrtowcs(), mbsrtowcs(), wcrtomb(), wcsnrtombs(), and wcsrtombs()"

Add a new paragraph to the mblen() DESCRIPTION after P1270, L41765:
"The mblen() function need not be thread-safe."
with CX shading.

Add a new paragraph to the mbrlen() DESCRIPTION after P1272, L41808:
"The mbrlen() function need not be thread-safe if called with a NULL ps argument."
with CX shading.

Add a new paragraph to the mbrtowc() DESCRIPTION after P1274, L41871:
"The mbrtowc() function need not be thread-safe if called with a NULL ps argument."
with CX shading.

Add a new paragraph to the mbsnrtowcs() and mbsrtowcs() DESCRIPTION after P1277, L41989:
"The mbsnrtowcs() and mbsrtowcs() functions need not be thread-safe if called with a NULL ps argument."
with CX shading.

Add a new paragraph to the mbtowc() DESCRIPTION after P1281, L42094:
"The mbtowc() function need not be thread-safe."
with CX shading.

Change:
"The wcsrtombs() function"
in the DESCRIPTION of wcsnrtombs() and wcsrtombs() on P2219, L69818 to:
"The wcsnrtombs() and wcsrtombs() functions"
keeping the CX shading.

On page 2235 remove line 70283 "The wcstombs() function need not be thread-safe"

0000110: memchr input process order

STATUS

Accepted. WG 14 review required.

Description

Traditional implementations of memchr process the input in ascending order. This has the advantage that when the object size of s is not known, but c occurs within the object, the caller can pass a value of n that is larger than the actual object size without dereferencing inaccessible memory. However, while the standard (and C99) is explicit that it is permissible to pass n smaller than the object size of s, it is silent on whether passing a larger n is well-defined.

In contrast, consider the wording for fprintf when dealing with the %.*s specifier, from line 29938:

"If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte."

Many implementations of the *printf family use memchr to implement this statement; for example, http://git.sv.gnu.org/cgit/gnulib.git/tree/lib/vasnprintf.c?id=d4ca645#n197 [^]

However, if memchr does not have any strict requirement on evaluation order, then this invokes undefined behavior. For example, here is a bug report showing what happens when memchr does not have the traditional behavior, but dereferences memory that fits with the n argument to memchr but not within the actual array passed to printf: http://www.alphalinux.org/archives/axp-list/March2001/0337.shtml [^]

Likewise, application writers have noticed that it is possible to write faster code for finding a NUL byte, if one is present within a bounded length, by using memchr rather than strnlen, since the former has fewer conditionals (bounds check and search for NUL) than the latter (bounds check, search for NUL, and search for c). For example: http://git.sv.gnu.org/cgit/gnulib.git/tree/lib/strnlen1.c?id=d4ca645 [^]

But again, this usage is rendered unsafe unless memchr is specified to behave like strnlen and not dereference past the match.

Agreed Action

In the DESCRIPTION remove "of the object" from

The memchr( ) function shall locate the first occurrence of c (converted to an unsigned char) in the initial n bytes (each interpreted as unsigned char) of the object pointed to by s.

In the RETURN VALUE section

The memchr( ) function shall return a pointer to the located byte, or a null pointer if the byte does not occur in the object.

to

The memchr( ) function shall return a pointer to the located byte,
or a null pointer if the byte is not found.

Also Nick will let the C committee know about the issue

Add to DESCRIPTION
Implementations shall behave as if they read the memory byte by byte
from the beginning of the bytes pointed to by s and stop at the first
occurrence of c (if it is found in the initial n bytes).

0000148: unclear return value description

STATUS

Accepted. No impact on WG 14, for information only.

Description

The pow man page currently says:

> On systems that support the IEC 60559 Floating-Point option, a pole error
> shall occur and pow(), powf(), and powl() shall return ±HUGE_VAL, ±HUGE_VALF,
> and ±HUGE_VALL, respectively if y is an odd integer, or HUGE_VAL, HUGE_VALF,
> and HUGE_VALL, respectively if y is not an odd integer.

The problem is that it is not described when the + or - in case of odd integers is used. In fact, I think the + is wrong.

Accepted Action

C99 (n1256) says in F.9.4.4:

-- pow(±0, y) returns ±inf and raises the ``divide-by-zero'' floating-point exception for y an odd integer < 0.
-- pow(±0, y) returns +inf and raises the ``divide-by-zero'' floating-point exception for y < 0 and not an odd integer.

Note that "pow(±0, y) returns ±inf" is shorthand for "pow(+0, y) returns +inf and pow(-0, y) returns -inf". (See F.9 para 12).

These requirements in C99 seem right to me.

POSIX doesn't quite match C99, in that it just says "zero" instead of "±0". I think we should change:

"On systems that support the IEC 60559 Floating-Point option, a pole error shall occur ..."

to:

"On systems that support the IEC 60559 Floating-Point option, if x is ±0, a pole error shall occur ..."

0000162: Determining System Endianess during Preprocessing

STATUS

Open. WG 14 input requested.

Description

Sometimes one wants to compile conditionally depending on the endianess of the target platform. I am unaware of a portable (standard-based) way of determining endianess at preprocessing time.

However, a minor change to the definition of ntohs() and related functions would achieve this. The man page currently reads:

"On some implementations, these functions are defined as macros."

The BSD document "An Advanced 4.4BSD Interprocess Communication Tutorial" (http://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf), [^] which introduced the ntohs() family of functions, states on page 15,

"On machines where unneeded these routines are defined as null macros."

I would like to have POSIX strengthened so that ntohs() etc. are macros exactly when the network byte order and host byte order are identical; that is, when the host is big-endian. The test for big-endianess could then be implemented as

```
# ifdef ntohs
/* big-endian specific code */
# else
/* non-big-endian, presumably little endian */
# endif
```

I realize that some machines, such as the PDP-11, are neither little nor big endian. However, such

machines, similar to non-octet byte machines, are so rare that POSIX should not have to cater to them.

Note that it is not possible to implement ntohs() etc. on a little-endian machine using only a macro, as the byte swap must evaluate its argument several times. In standard C, a function is required.

Current Status

Open.
Original Proposal:
Change the sentence

"On some implementations, these functions are defined as macros."

to

"On exactly those implementations where network and host byte order are identical these functions are defined as macros."
Comments:
"Note that it is not possible to implement ntohs() etc. on a little-endian machine using only a macro" - not true. Yes, it is not possible to do so using strictly C99 constructs, but an implementation is free to use extensions as part of implementing a macro, and many implementations DO use a macro on little-endian machines that outputs an assembly directive for a byte swap while evaluating the argument exactly once. For example, on one little-endian system I sampled:

```
# define ntohs(x) \
(__builtin_constant_p((short)(x)) ? \
 __constant_ntohs((x)) : \
 __ntohs((x)))
```

where the semantics of __builtin_constant_p are such that side effects of x are not evaluated.

If we are to support endianness detection, it should by standardizing existing implementations, which provide a new header <endian.h> that defines either LITTLE_ENDIAN or BIG_ENDIAN as appropriate.

If this clause of the Standard would be fullfilled by most Unices, we could already base on it:

> The BSD document "An Advanced 4.4BSD Interprocess Communication
> Tutorial" (http://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf), [^] [^] which
> introduced the ntohs() family of functions, states on page 15,
>
> "On machines where unneeded these routines are defined as null macros."

```
#ifdef ntohs
  #if ( ntohs ( 0x1234 ) == 0x1234 ) )
    #define BIGENDIAN_FOUND_AT_COMPILETIME = 1
  #endif
#else
  #define ENDIANNESS_UNKNONW_AT_COMPILETIME
#endif
```

This would also work in the case that ntohs is a macro
on a little endian platform.

Unfortunately on Linux (tested ppc and Intel) ntohs seems not to be a macro
so this test will mostly fail.

It may be preferable to standardize the __BYTE_ORDER, __BIG_ENDIAN and __LITTLE_ENDIAN macros found in glibc and some other UNICes.

The C committee hasn't shown enthusiasm to introduce macros for endianess handling. They are not opposed but somebody needs to do the work and preferably before the April C committee meeting.

Austin/501
Nick Stoughton

Liaison Report

October 5, 2010

If any of the people commenting here has interest, please provide a little paper describing the new feature with justification and changes to the spec. Either post it to the C committee mailing list directly or post it to the Austin Group mailing list and it will be appropriately forwarded.

0000170: Mis-alignment between date utility and strftime %j, and conversion specifier and tm_yday of struct tm

STATUS

Rejected. No WG 14 action required, for information only.

Descrip[tion

From Wayne Pollock's email, austin group sequence 12923:

I know the aardvark has been closed on aligning
the date utility with strftime (and strptime),
but I just noticed that there is disagreement
between these and struct tm for tm_yday.

tm_yday is defined in time.h as:

int tm_yday Day of year [0,365].

and %j in strftime (and strptime) is defined as:

j

Replaced by the day of the year as a decimal number [001,366]. [tm_yday]

Currently the date utility agrees with strftime:

%j
Day of the year as a decimal number [001,366].

I don't know if this "off by one" was on purpose or not.
I think it is too late to change this; as far as i know
strftime et. al. are implemented according to the standard,
and simply don't agree with tm_yday.

I would suggest that if strftime and friends
isn't to be changed to agree with tm_yday, then
the description for %j should be changed in all places,
from:
[tm_yday]
to:
[tm_yday + 1]

--
Wayne Pollock

Status

Rejected

On page 2009 line 63589, change
[tm_yday]

to
[tm_yday+1]

I think the proposed change is wrong. The standard is clear that the entire list of strftime conversion specifiers uses "zero or more members of the broken-down time structure pointed to by timeptr, as specified in brackets in the description" (line 63534), so it is an indication of _which_ members are used to compute the resulting string, and not _how_ those members are used in performing the calculation. The correct calculation for %j needs to be a string representation of the decimal value of tm_yday+1 with leading zeros, but that is already implied by the difference in ranges for what tm_yday can contain [0-365] and what %j will produce [001-366]. Meanwhile, the spec is correct as is in stating that %j relies on the value of the tm_yday. If we were to change brackets after %j to list a formula, then we would have to consider changing all the other conversion specifiers to also give formulas; and it gets complex fast (for example, what is a succinct formula for %V, which is shown on line 63609 as depending on the tm_year, tm_wday, and tm_yday fields?).
The suggested change would create a conflict with the C Standard.

0000173: Executing programs with "bad" file descriptors 0, 1 or 2

STATUS

Accepted. For information only.

Description

XSH7 section 2.5 states:

> "At program start-up, three streams are predefined and need not be opened explicitly: standard input (for reading conventional input), standard output (for writing conventional output), and standard error (for writing diagnostic output)."

The standard also provides the means to execute programs with fd 0 not open for reading, or fd 1 or 2 not open for writing. However,

it does not fully address the issue of what happens to stdin, stdout and stderr when a conforming application is executed in such a way. The security aspects of the issue were addressed in SUSv3 TC1, which resulted in implementations being allowed to automatically open these fds when executing set-user-ID or set-group-ID programs (although it only says this for exec, not for posix_spawn or the shell). At the same time a note was added to exec() application usage saying:

> "Applications should not depend on file descriptors 0, 1, and 2 being closed after an exec. A future version may allow these file descriptors to be automatically opened for any process."

I think we should make this change to allow implementations to do the automatic open for any process. However, the issue would still remain of what happens on implementations which do not open them, and what happens if they are open but not readable (fd 0) or not writable (fd 1 or 2). In practice, I believe what happens is that the stdin, stdout and stderr streams are created and their underlying file descriptors are set to 0, 1 and 2 respectively, regardless of whether they are open or closed, and regardless of whether fd 0 is readable or fd 1 or 2 is writable.

At first sight, it would seem that a simple solution is just to modify section 2.5 to describe this existing practice. The problem with this solution is that the guarantee in section 2.5 about stdin, stdout and stderr being open (and readable/writable) at program startup comes from the C Standard. I don't believe a modification

of this nature would be a valid extension to the C Standard - it would create a conflict.

I believe an appropriate solution would be to state that if a program is executed with fd 0 not open for reading or with fd 1 or 2 not open for writing then this means the program is not executed in a conforming environment. Thus the program is not required to behave as described in the standards (C and POSIX).

There is also a similar issue for the standard utilities, in that they may misbehave in odd ways if executed with fd 0 not open for reading or with fd 1 or 2 not open for writing. The solution for applications could be used to solve this issue as well.

Agreed Action

Interpretation response

The standard does not speak to this issue, and as such no conformance distinction can be made between alternative implementations based on this. This is being referred to the sponsor.

Applications should not execute programs with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing, as this might cause the executed program to misbehave.

Rationale:

None.

Notes to the Editor (not part of this interpretation):

At page 773 line 25757 section exec, change:

```
      If file descriptors 0, 1, and 2 would otherwise be closed after a
     successful call to one of the exec family of functions, and the
     new process image file has the set-user-ID or set-group-ID file
     mode bits set, and the ST_NOSUID bit is not set for the file
     system containing the new process image file, implementations may
     open an unspecified file for each of these file descriptors in the
     new process image.
```

to:

```
     If file descriptor 0, 1, or 2 would otherwise be closed after a
     successful call to one of the exec family of functions,

     implementations may open an unspecified file for the file
     descriptor in the new process image.  If a standard utility or a
     conforming application is executed with file descriptor 0 not
     open for reading or with file descriptor 1 or 2 not open for
     writing, the environment in which the utility or application is
     executed shall be deemed non-conforming, and consequently the
     utility or application might not behave as described in this
     standard.
```

At page 779 line 25993 section exec, change:

```
     Applications should not depend on file descriptors 0, 1, and 2
     being closed after an exec. A future version may allow these file
     descriptors to be automatically opened for any process.
```

to:

```
      Applications should not execute programs with file descriptor
```

0 not open for reading or with file descriptor 1 or 2 not open for writing, as this might cause the executed program to misbehave. In order not to pass on these file descriptors to an executed program, applications should not just close them but should reopen them on, for example, /dev/null. Some implementations may reopen them automatically, but applications should not rely on this being done.

After page 1423 line 46584 section posix_spawn, add a new paragraph:

If file descriptor 0, 1, or 2 would otherwise be closed in the new process image created by posix_spawn() or posix_spawnp(), implementations may open an unspecified file for the file descriptor in the new process image. If a standard utility or a conforming application is executed with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for

writing, the environment in which the utility or application is executed shall be deemed non-conforming, and consequently the utility or application might not behave as described in this standard.

After page 1425 line 46681 section posix_spawn, add:

See also the APPLICATION USAGE section for [xref to exec()].

Cross-volume change to XCU ...

At page 2318 line 73173 section 2.9.1.1, add a new paragraph after the numbered list (indented the same as line 73119):

If the utility would be executed with file descriptor 0, 1, or 2 closed, implementations may execute the utility with the file descriptor open to an unspecified file. If a standard utility or a conforming application is executed with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing, the environment in which the utility or application is

executed shall be deemed non-conforming, and consequently the utility or application might not behave as described in this standard.

Cross-volume changes to XRAT ...

After page 3660 line 124527 section C.2.7, add a new paragraph:

Applications should not use the [n]<&- or [n]>&- operators to execute a utility or application with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing, as this might cause the executed program (or shell builtin) to misbehave. In order not to pass on these file descriptors to an executed utility or application, applications should not just close them but should reopen them on, for example, /dev/null. Some implementations may reopen them automatically, but

applications should not rely on this being done.

After page 3517 line 118482 section B.2.5, add:

Although the C Standard guarantees that, at program start-up, stdin is open for reading and stdout and stderr are open for writing, this guarantee is contingent (as are all guarantees made by the C and POSIX standards) on the program being executed in a conforming environment. Programs executed with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing are executed in a non-conforming environment. Application writers are warned (in [xref to exec()], [xref to posix_spawn()], and [xref to C.2.7]) not to execute a standard utility or a conforming application with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing.

0000174: puts for strings longer than INT_MAX

STATUS

For information only; no WG14 action required.

Description

Strings do not have a length limit and on systems with, say, 64-bit address space and int being 32-bit, the result of calls to puts with strings as parameter of length > INT_MAX is unspecified. We cannot return the number of written bytes because all successful calls must return a non-negative value.

The same applies to fputs.

Agreed Action

Add the following to the Application Usage section of puts(), p1723 after line 55077:

"In some implementations puts() returned the number of bytes written.
However, this convention cannot be adhered to for strings
longer than INT_MAX bytes as the value would not be
representable in the return type of the function. For
backwards compatibility, implementations can return the
number of bytes for strings of up to INT_MAX bytes, and
return INT_MAX for all longer strings."

Also the same on page 908 after line 30373, with puts() changed to fputs().

0000211: inttypes should provide wchar_t

STATUS

For information only; no WG14 action required.

Description

In the move to make all of the standard headers self-contained (as an extension to the C99 behavior of requiring inclusion of prerequisite headers), we overlooked the use of wchar_t in the <inttypes.h> declaration of wcstoimax and wcstoumax.

Agreed Action

At line 8510, change 'type' to 'types'.

At line 8511, insert a new paragraph with CX shading:

wchar_t As described in <stddef.h>

0000249: Add standard support for $'...' in shell

STATUS

For information only; no WG14 action required.

Description

It is annoyingly difficult to set shell variables with certain characters, for example, to set a shell variable to a value that ends with "\n". This is particularly a problem when manipulating IFS, but the problem occurs in many other situations too. One reason is because command substitution consumes all trailing newlines, so constructs like this don't work:

```
IFS=$(printf "\n")
```

It's possible to work around this (e.g., with clever parameter expansions), but the work-arounds are ugly and error-prone. It would be better to support simple, clear scripts when you want to insert characters like newline.

Thus, many shells add the $'...' construct, which is *widely* supported. It is supported by ksh, zsh, bash, and the busybox "sh" command at *least*. This widespread support suggests that this is useful and easy to implement. The fact that even *busybox* supports it, even though they focus on being very small, suggests that this is *important* functionality.

As the korn shell documentation says, "the purpose of $'...' is to solve the problem of entering special characters in scripts [using] ANSI-C rules to translate the string... It would have been cleaner to have all "..." strings handle ANSI-C escapes, but that would not be backwards compatible."
http://kornshell.com/doc/faq.html [^]

I used bash's list of constructs supported in $'...'. I think supporting hex values is a good one. \cX constructs are less necessary, but since they're in bash, may as well include it too (it's trivial to support).

Desired Action

Page 2229, line 72401: At the end of section 2.2, add a new subsection, "2.2.4 Dollar-single-quotes", with this text:

A word beginning with dollar-single-quote ($'), continuing all the way to an unescaped matching single-quote ('), shall preserve the literal value of all characters within, with the exception of

certain backslash escape sequences.
The defined backslash escape sequences are:

\a alert (bell)
\b backspace
\e an escape character
\f form feed
\n new line
\r carriage return
\t horizontal tab
\v vertical tab
\\ backslash
\' single quote
\nnn the eight-bit character whose value is the octal value
nnn (one to three digits)
\xHH the eight-bit character whose value is the hexadecimal
value HH (one or two hex digits)
\cx a control-x character; \cA is 1, while \cZ is 26.

If a backslash is followed by any other sequence, the result is implementation-defined.

The expanded result is treated as if it was single-quoted (as if the dollar sign had not been present); in particular, it is *not* split using the IFS values.

On page 2298, after line 72385, add:
Within the string of characters from an enclosed "$'" to the matching "'", processing shall occur as described in section 2.2.4.

In 0000049 , need to add to the end of the list:

* a <single-quote>

See: http://austingroupbugs.net/view.php?id=49 [^]

Notes

The C standard also includes Universal Character constants, and these should also be considered.

6.4.3 Universal character names
Syntax
1 universal-character-name:
\u hex-quad
\U hex-quad hex-quad
hex-quad:
hexadecimal-digit hexadecimal-digit
hexadecimal-digit hexadecimal-digit
Constraints
2 A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 ($), 0040 (@), or 0060 (‘), nor one in the range D800 through DFFF inclusive.)
Description
3 Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.
Semantics
4 The universal character name \Unnnnnnnn designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is nnnnnnnn.) Similarly, the universal character name \unnnn designates the character whose four-digit short identifier is nnnn (and whose eight-digit short identifier is 0000nnnn).

So far, discussion of this bug has been about which escapes to include.
There are some problems with other aspects of the proposed text that
need to be sorted out. The ones I have spotted are:

Use of "A word beginning" is wrong, as the expansion can be used within a word. E.g. a$'b'c expands to abc.

The sentence beginning "The expanded result is treated as if it was single-quoted" needs to use "shall", and the rest of that sentence should either be removed or reworked.

This addition to 2.2.3 Double-Quotes is not right:

> On page 2298, after line 72385, add:
> Within the string of characters from an enclosed "$'" to the matching "'", processing shall occur as described in section 2.2.4.

since ksh and bash do not expand $'...' within double quotes.
A possible alternative might be to append the following to line 72376:

> The <dollar-sign> shall not retain its special meaning introducing the dollar-single-quotes form of quoting (see [xref to 2.2.4]).

Finally an additional change is needed in 2.3 Token Recognition in list item 4.

NOTE: C99 states

6.11.4 Character escape sequences

1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

This appears to omit the fact that \U is already used by the standard, even though it is in the “extension” space. It is also unclear whether “future standardization” includes POSIX (ISO/IEC 9945), or means “future C standards”. **Nick to request clarification from the C committee.**

0000258: strftime needs an %EB specifier

STATUS

For information only; no WG14 action required.

Description

I would like to propose a new format specifier for strftime function: %EB
It should stand for an alternative form of a full month name in current locale. It is needed for some Slavic languages, for example Russian and Polish, which use nominative case for (example) month/year combination, and genitive case for day/month/year combination. Right now, grammatically correct forms cannot be built with any implementation of corresponding locales with current strftime definition. The standard should provide this option for uniformed and correct locale design.

Agreed Response

Interpretation response

The standard does not speak to this issue, and as such no conformance distinction can be made between alternative implementations based on this. This is being referred to the sponsor.

Rationale:

The locale specification is believed to be incomplete.

Notes to the Editor (not part of this interpretation):

Based on the observation that FreeBSD implemented support for this in 1999, but somewhat differently:
The specifier %B remains the genitive (with day number) and a new
specifier %OB is used for the nominative (no day number), apparently
thought as less common. Although the %O modifier already has a use, this

does not conflict with %B as %B does not contain "digits".

Based on the observation that FreeBSD implemented support for this in 1999, but somewhat differently:
The specifier %B remains the genitive (with day number) and a new
specifier %OB is used for the nominative (no day number), apparently
thought as less common. Although the %O modifier already has a use, this
does not conflict with %B as %B does not contain "digits".

We suggest the following resolution:

Locale Sec 7.3.5.1 LC_TIME p159 after the sentence ending on line 4903 add:
For languages having both a genitive (when used with a day number) and a nominative (no day number) case, this operand shall be used to denote the genitive case.

Locale Sec 7.3.5.1 LC_TIME p159 after line 4903 insert:
alt_mon Define the full month names, corresponding to the %OB conversion specification. The operand shall consist of twelve <semicolon>-separated strings, each surrounded by double-quotes. The first string shall be the full name of the first month of the year (January), the second the full name of the second month, and so on. For languages having both a genitive (when used with a day number) and a nominative (no day number) case, this operand shall be used to denote the nominative case.

Locale Sec 7.3.5.2 LC_TIME p161 after line 4979 insert:
ALTMON_x The alternative full month names (for example, January), where x is a number from 1 to
12.

Locale Sec 7.3.5.3 LC_TIME p163 after line 5109 insert in the table:
alt_mon ALTMON_1 %OB N/A
alt_mon ALTMON_2 %OB N/A
alt_mon ALTMON_3 %OB N/A
alt_mon ALTMON_4 %OB N/A
alt_mon ALTMON_5 %OB N/A
alt_mon ALTMON_6 %OB N/A
alt_mon ALTMON_7 %OB N/A
alt_mon ALTMON_8 %OB N/A
alt_mon ALTMON_9 %OB N/A

alt_mon ALTMON_10 %OB N/A
alt_mon ALTMON_11 %OB N/A
alt_mon ALTMON_12 %OB N/A

Locale Sec 7.4.2 Locale Grammar p171 at the end of line line 5437 add:
| 'alt_mon'

langinfo.h p264 after line 8661 insert:
ALTMON_1 LC_TIME Name of the alternative appropriate first month of the year.
ALTMON_2 LC_TIME Name of the alternative appropriate second month.
ALTMON_3 LC_TIME Name of the alternative appropriate third month.
ALTMON_4 LC_TIME Name of the alternative appropriate fourth month.
ALTMON_5 LC_TIME Name of the alternative appropriate fifth month.
ALTMON_6 LC_TIME Name of the alternative appropriate sixth month.
ALTMON_7 LC_TIME Name of the alternative appropriate seventh month.
ALTMON_8 LC_TIME Name of the alternative appropriate eighth month.
ALTMON_9 LC_TIME Name of the alternative appropriate ninth month.
ALTMON_10 LC_TIME Name of the alternative appropriate tenth month.
ALTMON_11 LC_TIME Name of the alternative appropriate eleventh month.
ALTMON_12 LC_TIME Name of the alternative appropriate twelfth month.

strftime p2010 after line 63652 insert:
%OB Replaced by the locale’s alternative appropriate full month name.

strptime p2028 after line 64194 insert:
%Ob The locale’s appropriate alternative full month name. Either the abbreviated name or the alternative full name may be specified.
%OB Equivalent to %Ob.

date p2576 after line 82930 insert after "era_d_fmt,":
alt_mon,

date p2577 after line 82939 insert:
%OB The locale’s alternative appropriate full month name.

0000327: time_t should be integral

STATUS

For information only; no WG14 action required.

Description

If I'm interpreting history correctly, when time_t was originally standardized, it was allowed to be floating point for two reasons, even though all implementations at the time used an integer:
1. to allow a future implementation to provide a QoI improvement where interfaces like stat() could provide sub-second resolution
2. to allow a future implementation to use IEEE double-precision in order to get 54-bits of integral accuracy instead of 32-bits, before 64-bit integers were common, in order to avoid the year 2038 wraparound limitation

However, both of these aspects have been overcome by history:
1. POSIX 2008 now requires that stat() return struct timespec, so that the fractional portion of a timestamp is represented in tv_nsec; and has added other interfaces like utimensat() that act on struct timespec for full precision of timestamps
2. POSIX 2008 now requires 64-bit integers for _POSIX_V7_ILP32_OFFBIG, which is pretty common for 32-bit hosts these days; and obviously 64-bit hosts support 64-bit integers

Not all conformant systems have converted to a 64-bit time_t yet (in fact, using a 64-bit OS with 32-bit time_t is still all too common), so the standard must still allow 32-bit time_t, and portable code must be aware of this.

However, I am unaware of any compliant system that has actually implemented a floating-point time_t, even though the standard allows it; and the mere fact that the standard allows a weirdnix implementation with floating-point time leads to some awkward portability constraints, that would otherwise be avoidable. For example, the standard is silent on whether (time_t)(uintmax_t)st.st_atime must be equal to st.st_atime. If they need not be equal (because time_t is floating point, and the value contained a fraction), then how does that fraction interact with the st.st_atim.tv_nsec portion of the timestamp?

Additionally, the standard itself has non-normative statements that conflict with the normative requirement that floating-point time_t be allowed, such as line 68373 ("Since the utimbuf structure only contains time_t variables and is not accurate to fractions of a second").

Therefore, in light of recent API additions in POSIX 2008 and in light of existing implementation practice, it makes sense to tighten the standard, and require that time_t be integral. This is tighter than required by C99, but is not the first time that POSIX has imposed additional constraints beyond the C standard (think of NULL, for example).

There are a few remaining APIs that might deserve struct timespec handling, for uniform sub-second time-tracking, but these can be saved for separate bug report(s):
<sys/msg.h> defines msqid_ds using only time_t
<sys/sem.h> defines semid_ds using only time_t
<sys/shm.h> defines shmid_ds using only time_t

difftime() takes time_t but returns double, meaning it cannot represent the difference between all possible 64-bit time_t

Desired Action

At line 13420 (XBD <sys/types.h>) change:

time_t and clock_t shall be integer or real-floating types.

to:

clock_t shall be an integer or real-floating type.
<CX> time_t shall be an integer type.</CX>

At line 18867 (XSH 2.12.1 Defined Types) change:

time_t Integer or real-floating type used for time in seconds, as defined in the ISO C standard.

to:

time_t Integer type used for time in seconds, as defined in the ISO C standard.

At line 115844 (XRAT A.4.15 Seconds Since the Epoch) change:

The data size for time_t is as per the ISO C standard definition, which is implementation-defined.

to:

This standard requires that the time_t type be integral with implementation-defined size, but does not mandate a particular size. The requirement that time_t be integral is an additional constraint beyond the ISO C standard, which allows a real-floating time_t. Implementation practice has shown that much existing code is unprepared to deal with a floating-point time_t, and that use of struct timespec is a more uniform way to provide sub-second time manipulation within applications.