ISO/IEC JTC 1/SC 22/WG14 N 2069

August 20, 2016 v 1

Mandatory C library headers
simplify the transition to a new C standard

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

The transition from one version of the C standard to the next forces C implementors and C programmers to use cascades of non-portable preprocessor conditionals, that are a nightmare for maintenance. These difficulties have an adversary effect for early adoption of a new standard. To be able to deal better with such transitions, C needs tools to identify partial implementations of library features such that new tools can be shipped early and such that C programmers can prepare for them without knowledge of specific versions of implementations.

The transition from one C standard to the next is a difficult exercise for everybody, for compiler and library providers that have to implement new features and for the programmers that want to upgrade their code to these new features. Usually this difficulty is not in the implementation of individual features themselves, but in the coordination of the effort between the different parties.

In particular upgrading a compiler frontend to a new C language standard is often done independently from upgrading different parts of the C library. This is because most modern platforms are not monolithic but very heterogeneous:

— Several compiler frontends are provided by several sources. *E.g* on most POSIX platforms there are versions of the open source compilers `gcc` and `clang`, but also commercial compilers provided by hardware or OS vendors.
— These compiler frontends are used with different C libraries (*e.g* on Linux or Windows systems) or versions thereof.

This proposal tries to improve on that situation and to simplify the life for both ends, C implementors and C programmers.

GOAL 1. *Allow optional C library headers to be incomplete.*

GOAL 2. *Provide feature test macros for completeness of headers.*

GOAL 3. *Provide a version macro for all C library headers.*

GOAL 4. *Distinguish test macros for compiler and library features.*

GOAL 5. *Make all C library headers mandatory.*

C11's approach to query the **__STDC_VERSION__**, and the feature test macros

— **__STDC_HOSTED__**
— **__STDC_NO_ATOMICS__**
— **__STDC_NO_COMPLEX__**
— **__STDC_NO_THREADS__**

is not a satisfactory solution to deal with library versions: they have to be queried *before* any header file has been included and thus cannot easily deal with different versions of the C library. This is in particular not appropriate during a transition phase between C versions, where newly implemented features should get as much coverage as possible.

The solution to these difficulties is usually to impose complicated preprocessor tests that ensure that a certain header can be included and that provide fallbacks if certain features are not yet implemented. These preprocessor tests usually use specific compiler and library knowledge, in particular version numbers, and are in most cases a maintenance nightmare, for C implementors as well as for C programmers.

Depending on the platform definitions, *most* of the standard library headers can be optional. As of C11, the minimal requirement is only to have `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdnoreturn.h>` in a free standing environment. The presence of all other C library headers can be checked by feature test macros, see Table I.

header	C11 feature test macro	new
`<assert.h>`	__STDC_HOSTED__	`__STDC_ASSERT__`
`<complex.h>`	__STDC_NO_COMPLEX__	`__STDC_COMPLEX__`
`<errno.h>`	__STDC_HOSTED__	`__STDC_ERRNO__`
`<fenv.h>`	__STDC_HOSTED__	`__STDC_FENV__`
`<inttypes.h>`	__STDC_HOSTED__	`__STDC_INTTYPES__`
`<locale.h>`	__STDC_HOSTED__	`__STDC_LOCALE__`
`<math.h>`	__STDC_HOSTED__	`__STDC_MATH__`
`<setjmp.h>`	__STDC_HOSTED__	`__STDC_SETJMP__`
`<signal.h>`	__STDC_HOSTED__	`__STDC_SIGNAL__`
`<stdatomic.h>`	__STDC_NO_ATOMICS__	`__STDC_STDATOMIC__`
`<stdio.h>`	__STDC_HOSTED__	`__STDC_STDIO__`
`<stdlib.h>`	__STDC_HOSTED__	`__STDC_STDLIB__`
`<string.h>`	__STDC_HOSTED__	`__STDC_STRING__`
`<tgmath.h>`	__STDC_HOSTED__	`__STDC_TGMATH__`
`<threads.h>`	__STDC_NO_THREADS__	`__STDC_THREADS__`
`<time.h>`	__STDC_HOSTED__	`__STDC_TIME__`
`<uchar.h>`	__STDC_HOSTED__	`__STDC_UCHAR__`
`<wchar.h>`	__STDC_HOSTED__	`__STDC_WCHAR__`
`<wctype.h>`	__STDC_HOSTED__	`__STDC_WCTYPE__`

Table I. Feature test macros for C library headers with optional features

This proposal is intended to invert the situation, namely that the feature test macros can be made dependent of the header file. The "only" requirements for this to be possible, is to mandate the presence of all these (currently 29) headers and to allow them to be mostly empty, eventually.

So in particular, this does not mean that free-standing environments would have to provide all features of the C library, they would just have to provide mostly empty files or have some simple fallback for the include `<>` construct.

As a consequence of our approach, some of the macros that are mandated in the corresponding header files can also be used as feature test macros. E.g *user code* could provide an alternative for assert macros on freestanding environments:

```
#include <assert.h>
#ifndef static_assert
# define static_assert(X, DOC)                          \
  typedef char _Dummy["" DOC "" && sizeof(char[(X)])]
#endif
```

That is the header can be included unconditionally, the presence of the feature can then be tested and an alternative can be provided.

With our approach updating an existing header file to a new C version should be easy. Before implementing additional features that might be requested in most cases simply adding the feature test macro should suffice. *E.g* for the atomics extension:

```
#ifndef __STDC_LIB_VERSION__
# define __STDC_LIB_VERSION__ __STDC_VERSION__
```

```
#endif
#define __STDC_STDATOMIC__ 201112L
#if __STDC_LIB_VERSION__ > __STDC_STDATOMIC__
# define __STDC_LIB_VERSION__ __STDC_STDATOMIC__
#endif
```

This clearly indicates to C programmers that certain feature are not yet available, but that they can rely on features of a previous version of the standard. A free standing environment that does not implement `<setjmp.h>`, say, would just have to add a header

```
#ifndef __STDC_SETJMP__
# define __STDC_SETJMP__ 0L
# undef __STDC_HOSTED__
# define __HOSTED__ 0
#endif
```

or trigger such definitions for any unknown header file that is included with `<>`.

Sections of the C standard to be amended
6.10.8:
Remove **p1** about stating that the macros remain constant throughout compiling the same TU. This has to be elaborated specifically for the different sections.

Modify the first phrase of **p2** from:

```
None of these macro names, nor the identifier defined, shall be the subject
of a #define or a #undef preprocessing directive.
```

by

```
None of these macro names shall be the subject of a #define or a #undef
preprocessing directive other than effected by the inclusion of a standard
header.
The identifier defined shall not be the subject of a #define or a
#undef preprocessing directive.
```

Note: we'd have to think of a better place this constraint for **defined**.
Add a new paragraph.

```
Macros that reflect versions of ISO/IEC 9989 shall expand to the value
of __STDC_VERSION__ of the corresponding version, to 199000L
if the version is ISO/IEC 9899:1990, or to 0L if no version is supported.
```

6.10.8.1, add:

```
The values of the predefined macros __DATE__, __TIME__,
__STDC__ and __STDC_VERSION__ remain constant throughout
the translation unit.
```

Add the following item and footnote to the list in Section 6.10.8.1:

```
__STDC_LIB_VERSION__ The minimal version of ISO/IEC 9899 for
which all included library headers are feature complete.FOOTNOTE

  FOOTNOTE: The intent is that this macro should only differ
  from __STDC_VERSION__ during the transition of an
  implementation to a new version of ISO/IEC 9899.
```

Move the item for __STDC_HOSTED__ from Section 6.10.8.1 to 6.10.8.3. and add the following text at its end.

```
This macro shall be defined whenever one of the standard headers
that are optional for a hosted environment is included. Its
value shall be 1 as long as the included standard headers
provide all mandated features. Its value shall be 0 if any of
the included standard headers misses any of the mandated
features.
```

6.10.8.2, describes language and not library features, therefore they should not change during complilation. Add as new first paragraph

```
The values of the predefined macros listed in the following
subclause are either defined or not before inclusion of all
standard headers and remain unchanged throughout the translation unit.
```

6.10.8.3, add a new first paragraph with example:

```
Each standard header with name of the form <xxxxx.h> that is not
listed in clause 4, p6, defines a macro __STDC_XXXXX__ that expands
to the maximum version number of ISO/IEC 9989 to which this header complies.
This version is greater or equal to __STDC_LIB_VERSION__.

EXAMPLE: In a hosted environment, the header <locale.h> defines the
macro __STDC_LOCALE__ with a value that is at least 199000L.
In a freestanding environment, this macro can also evaluate to the
value 0L to indicate that no conforming implementation of the locale
feature is available. In that case, the macro __STDC_HOSTED__
also is 0 after inclusion of <locale.h>.
```

Then replace the three items

```
__STDC_NO_ATOMICS__ The integer constant 1, intended to
indicate that the implementation does not support atomic
types (including the _Atomic type qualifier) and the
<stdatomic.h> header.
```

```
__STDC_NO_COMPLEX__ The integer constant 1, intended to
indicate that the implementation does not support complex
types or the <complex.h> header.
```

```
__STDC_NO_THREADS__ The integer constant 1, intended to
indicate that the implementation does not support the
<threads.h> header.
```

as follows:

```
__STDC_NO_ATOMICS__, after inclusion of the <stdatomic.h>
header, the integer constant 1, intended to indicate that
the implementation does not support atomic features including
the _Atomic type qualifier.
```

```
__STDC_NO_COMPLEX__, after inclusion of the  <complex.h>
header, the integer constant 1, intended to indicate that
the implementation does not support complex types.
```

```
__STDC_NO_THREADS__, after inclusion of the <threads.h>
header the integer constant 1, intended to indicate that
the implementation does not support the threads
interface.
```

7.1.2, **p2**, remove the footnote (currently number 183). Then add at the end the following paragraph and example after the list of the headers.

```
All these headers shall be present for a conforming implementation. If
a header describes an optional feature that can be tested by a
predefined macro (see Section 6.10.8) the header defines some or all
of the corresponding features and provides the necessary feature test
macros after inclusion.
```

```
Example: In a freestanding environment the header <locale.h>
shall be available, but must not necessarily implement all
features. After the inclusion the macros __STDC_LOCALE__ and
__STDC_HOSTED__ are defined. If not all locale features
of a specific version of ISO/IEC 9989 are implemented by the
header they evaluate to 0L and 0, respectively.
```

Additionally, other of the macros that are specified by library headers can be used as tests for partial features, see Table IV.

This interpretation of the presence of these macros could be enforced with some redactional effort, but we think that it is sufficient to recommend a reasonable usage. We propose the following

Sections of the C standard to be amended

Add to the end of 7.1.2 p2:

RECOMMENDED PRACTICE. If an implementation does not support all features of a specific header or of a new version of it, it should use the macros that are to be provided by the header as test macros for partial features. Therefore an initialization or manipulation macro for a specific type should only be provided if the data types and its operations are also provided. E.g the presence of **ATOMIC_VAR_INIT** should indicate that the **_Atomic** qualifier and operators for atomic objects are provided, and likewise the presence of the macro **ONCE_FLAG_INIT** should indicate the availability of the type **once_flag** and the function **call_once**.

header	macro	feature
<**complex**.h>	**I**, **complex**	**_Complex**
<math.h>	**FP_FAST_FMA**	**fma**
	FP_ILOGB0	**ilogb**
	MATH_ERRNO	**math_errhandling**
<**signal**.h>	**SIG_DFL**	**signal**
<stdatomic.h>	**ATOMIC_VAR_INIT**	**_Atomic**
	ATOMIC_XXXX_LOCK_FREE	type atomic_xxxx
	ATOMIC_FLAG_INIT	**atomic_flag**
<stdio.h>	**BUFSIZ**	**setbuf**
	_IONBF	**setvbuf**
	FOPEN_MAX	**fopen**
	TMP_MAX	**tmpnam**
	SEEK_SET	**fseek**
<stdlib.h>	**EXIT_SUCCESS**	**exit**
	RAND_MAX	**rand**
<threads.h>	**thread_local**	**_Thread_local**
	ONCE_FLAG_INIT	**once_flag**
	TSS_DTOR_ITERATIONS	**tss_t**
<**time**.h>	**TIME_UTC**	**timespec_get**
	CLOCKS_PER_SEC	**clock**

Table IV. Macros to test partial features