Study Group Proposal
WG14 N3171

| | |
|---|---|
| Title: | C Requests for Implementation |
| Author, affiliation: | Alex Celeste, Perforce |
| Date: | 2023-09-24 |
| Proposal category: | Study Group project |
| Target audience: | Compiler/library developers, WG members |

# Abstract

This paper proposes a new mechanism for the evolution and development of extended C library features that provides greater flexibility to implementers and greater involvement by Community members, by establishing a “staging area” for individual library features before full adoption in the C Standard.

The “staging area” is proposed to consist of a Community project outside the management and ownership of ISO, with responsibility for identifying, evolving, and wordsmithing library feature proposals, in the public domain; and a corresponding Committee SG, with the responsibility for selecting finalized feature proposals for inclusion into an ISO TS.

# C Requests for Implementation

Reply-to: Alex Celeste (aceleste@perforce.com)
Document No: N3171
Revises Document No: N/A
Date: 2023-09-24

## Document History

### N3171

- original proposal

## Introduction

Over the course of development of C23, a number of problems with the standardization process were identified. In this proposal we suggest a refinement to the feature development process that aims to ease three specific problems:

- engagement between the Committee and the Community is relatively poor. Users do not in general have a good way of communicating real-world requirements or possible improvements through to the Committee. Users have no straightforward way of bringing original work to the Committee at all, without leveraging personal contacts or networking. The original intent of filtering user requirements up through their vendors has stalled, because…
- implementers are reluctant to experiment with features that are not already approved by the Committee on at least an indicative (N-document) level. This creates a Catch-22 situation where the Committee is reluctant to progress an idea for lack of experimentation, while implementers are unwilling to do the experimentation without an authoritative request, because “{tool} is not your playground”.
- When the Committee does engage in original work, turnaround is extremely slow because of the meeting schedule and the deliberative format. Even technical proposals with widely-agreed merit can take months or years to pass through “design by Committee” because each objection to wording details can effectively require the next revision to go through the same process again.

The basic idea is that by adding a “staging area” for library feature development, we can disconnect the development of library feature proposals from the main standardization process, removing the reliance on N-documents, meeting schedules, and implementation experience, until the proposal has already been reviewed, experimented with by implementers and users, and been wordsmithed into a shape that is more likely to reach consensus to adopt on its first attempt.

This “staging area” may itself serve as a useful final destination for features that are considered useful, but not providing essential core language functionality.

# The SRFI model

This proposal borrows from a feature evolution structure created by the Scheme programming language community, the Scheme Requests for Implementation (SRFI).

The Scheme specification ("Revised Report") is developed by a (non-ISO) Working Group. In order to keep the language focused, the core aims to be relatively small and additional features are separated out. The current version of the language is the R7RS (i.e. the seventh revision), which was defined as a small core language with very few libraries, developed by Scheme WG1; and is in the process of being extended into a larger language with additional, optional features by Scheme WG2.

The main way these optional features have been specified is as SRFIs, which are periodically picked up by WG2 and added to the R7RS. However, because the SRFI project is independent of the Working Groups, implementations are free to pick up whichever of the individual feature specifications they wish to support ahead of time, and register their support for those features against the SRFI number. This provides valuable implementation experience and, if the SRFI is not yet in *final* status, a feedback mechanism to refine the feature before it becomes live. SRFI has been running since the release of the fifth revision (R5RS) and was integral to the development of R6Rs and R7RS.

SRFIs conform to a strict set of requirements, including a rationale, complete specification, and a sample implementation. The specification is required to be enough for an implementer, but the same provides guidance and is considered part of the submission as well. SRFI accepts portable as well as non-portable (and in some cases, incomplete or divergent) sample implementations; they are not "reference implementations" in the sense of necessarily being code intended for implementer reuse, but ideally they *could* be reused.

Every SRFI is hosted as a repo in the SRFI GitHub project. This enables each number to have an independent copyright notice, although the rules require them to be permissive anyway.

In general the approach is that the editors accept all proposals that actually manage to conform to the fairly strict requirements, regardless of whether they think the proposal is a good idea; if it is a bad idea the rationale is that nobody will include it in their implementation. There is a strict time frame for acceptance or rejection to encourage prompt turnaround.

There is a standard template for new SRFI proposals.

## Considerations for adoption by C

C would benefit by copying this library evolution model more or less wholesale.

Because the index of proposals is not normative, there is no requirement for any implementation to pick up all listed features, so features developed in this way are not "invention" that the Committee would be forcing on tool vendors. However, the non-binding request to experiment gives guidance to the likely future directions of WG14 and gives implementers something to point to in justification for experimental work. It may also help to prevent early feature divergence by encouraging collaboration during the refinement process (including the "namespace treadmill" problem).

C would likely benefit from slightly longer time frames than those listed in the SRFI Process, but the aim is to reduce the turnaround for development significantly from the current very slow process.

Eventually, adoption of selected CRFIs by C would follow two major directions: a small number of CRFIs would hopefully prove useful and *fundamental* enough to justify being included in Section 7 (or even to be reworked as language features for Section 6) of the Standard. This is analogous to adoption into a RnRS-small. The majority of remaining *final* CRFIs would be selected by a Study Group and bundled as top-level clauses of a TS, roughly analogous to being picked up for adoption into an RnRS-large. Although each clause would be independent (in the fashion of TR 18037, which explicitly says that the clauses can be imported independently, but that they should be implemented as complete units), as a whole the TS would effectively describe a "Section 8 Extended Library", containing potentially useful features that are not necessarily core to providing the working language.

The TS would be written in the style of the Standard itself and would not import the sample implementations.

CRFIs should ideally focus more strictly on library features and APIs than SRFI, which also contains some entries for new syntax or new fundamental language types. These are always going to be less portable, and are also much more difficult to make useful samples for in a non-Lisp language with limited metaprogramming support.

## What feature goes where?

To determine where a feature proposal should go, all of C is divided into three parts:

- Section 6 "Language": as it exists in the Standard, this describes features that are built directly into the syntax and semantics. These features are all provided directly by the compilation process and cannot be "imported"; they are either present, or they are not. Some optional features (e.g. complex numbers) do exist in the current version of the Standard, or in TSes (e.g. fixed point numbers), or as completely non-ISO extensions (e.g. labels as values); however, fundamentally, this section describes features that are intrinsic to the notation and cannot be imported short of changing the compiler settings.
- Section 7 "Library": this describes a small-ish set of APIs that are required for any useful work to be done with the language (e.g. `malloc`, `fopen`), which provide essenial functionality that cannot be defined within the language itself, either because it relies on a service provided by the platform, or on some detail about the platform which portable C is supposed to abstract. In addition, some APIs that are not strictly necessary are also found here (e.g. `calloc`, `bsearch`).
  We suggest that this section in future aims to stay as small as possible by only including features that expose a platform feature in a portable way, or that abstract secret platform knowledge and thus make it portable.
- Section 8 "Extended Library": this does not exist in the Standard. This hypothetical section is the realm of features that can be built entirely on top of what is provided by Sections 6 and 7, and exposed as APIs (e.g. `memcmp`, `qsort`). This does not mean they would not be more efficient as builtins, but their semantics do not demand a builtin in the same way as the bulk of useful features in Section 7.

By defining the *concept* of a Section 8, the pressure to adopt features into Section 7 and ship them as part of the IS itself is reduced. Features that belong in Section 8 would be adopted by the TS, but not expected to be adopted into the IS.

Some example features and an analysis of where they should go:

- bit-precise integers aim to be directly integrated into the language syntax and promotion rules, and to be as user-friendly as any of the standard arithmetic types. Because they use the same infix and literal syntax as any C17-native type, and because they support implicit conversions and modified integer promotions, they need to be directly integrated at a syntactic and type system level and therefore cannot reasonably be provided outside of Section 6, at least in any way that reasonably reflects the authors' intent.
  Essentially the same applies for decimal floating point numbers.
- checked integers and supplemental checked integers do *not* aim to provide literal or infix syntax (this was discussed, but not pursued) and instead rely entirely on opaque, strong types and function-call syntax. The feature can be implemented entirely in portable C in a single header, in a way that conforms exactly to everything specified in the document. Although the implementation will be more efficient if it has the option to call GCC `__builtin`s, there is no requirement for it to do so in order to produce *correct* results.
  This makes this feature a strong candidate for inclusion in Section 8, because it does not rely on any non-portable platform functionality being present – even though it will take advantage of it, if it is.
  This feature is also a strong use case for how the evolution process would be improved by moving review and wordsmithing to a separate group that is not dependent on Committee time. Despite widespread approval of the *concept*, checked integers took a very long time to be adopted because the Committee kept sending the proposal back to the authors for small changes, and almost failed to make it into C23 as a result of process limitations.
- `memset_explicit` relies on a behaviour that cannot easily be specified in the Standard because what it actually does is outside of what the language is able to define. The important part of the description relies on non-normative text.
  However, this means that the feature *cannot* usefully be implemented in portable, user-written code – it does rely on deep platform integration and on some underlying mechanism to bypass the optimizer being present; the fact this isn't specified in normative text is purely a limitation of the document. Therefore, this feature has to go into Section 7 because a useful reference implementation is more or less impossible to provide (some argument about whether `volatile` can be used, but it would not be enough to reflect the intent).
- `memalignment` exposes secret platform knowledge in a portable way. The functionality itself is relatively easy to write in non-portable C that has knowledge of the underlying address structure (i.e. it relies on casts to `uintptr_t` telling the user something intelligible), but the purpose of the API is to avoid the user needing to access that knowledge directly. Therefore, the ultimate destination of this feature had to be Section 7, *but* it would have been a good candidate for development as a CRFI because assuming knowledge of common platforms with simple address spaces (like x86_64) would be acceptable for the purposes of providing a sample implementation; the "sample" provided in the N-document completely illustrates the semantics of the API, it just depends on the simplest option being available.

- stdbit.h does not expose secret platform knowledge or features, and even assumes that *some* of the API will not be implemented as natively as other parts that map directly to an intrinsic.
  However, although every part of this API can be implemented in portable C and provide correct results, the main purpose of this feature is to take advantage of the fact that the intrinsics exist. It isn't useless in the absence of intrinsics, but their existence does make up a very large part of the rationale.
  Therefore, this is a good candidate for Section 7 *even though* it could also, if necessary, have shipped in Section 8 – the value to the user in this API is not in the arithmetic it implements but in the promise that the arithmetic will *always* be implemented natively if a native opcode exists.

- linmath.h is a small (single-header) library for vector and matrix math that has never been proposed for adoption into the Standard. It is completely portable and relies on the optimizer for performance, trusting that the simple patterns used will be picked up and vectorized automatically.
  This is an excellent candidate for a CRFI because the semantics are the main point rather than eking performance out of the system – it aims to abstract matrix and vector operations under a readable, function-call API (like the one used for checked integers), not to necessarily provide obscure operations or highly optimized throughput. Therefore, the main implementation serves perfectly as a sample (and reference) implementation of the interface. This would target Section 8, but might not even need to be adopted into the TS in order to provide utility. Implementers could choose to recognize the APIs and replace them with intrinsics, but do not need to do *anything* in order to support the feature as it is intended.

- `defer` is the most extreme case of what can potentially be argued to work as a library feature. `defer` is semantically a control structure, but because it *happens* to take the form of keywords followed by either expressions or braced blocks, it is possible to implement I almost entirely using clever macros.
  Therefore, this feature works as a CRFI even though it would ideally target Section 6, because users can get stared with writing code for simple cases using the reference implementation, which would serve well as a sample implementation. The feature cannot be implemented *completely* with macros, but only some functionality is missing and the sample implementation does still fulfil the essential requirement of demonstrating how it is intended to work (so long as it is given a sample *program* that avoid certain edge cases).
  The reference implementation also puts great effort into tuning for some implementation extensions, which is not required at all for a sample implementation but does definitely show how specific vendors might start to integrate it.

- a hypothetical `stdalgorithm.h` would be a good fit for a CRFI because most of what it would provide would again be demonstrable by a single-header example, even hough ideally it would have optimizer support.
  This feature would probably target Section 8, because even though it would want the optimizer to be aware of the various operators, they would not depend on it for correctness.

- a hypothetical `stdtypetraits.h` would be a good fit for a CRFI that relies to some extent on builtins and extensions. Some traits (e.g. `IsArray`) can be expressed in portable C, but some (e.g. `IsUnion`) cannot, and would require a `__builtin` to be available.

The sample implementation could either provide only the portable APIs, demonstrating intent, and specify the rest as interfaces; or it could choose to rely on a specific implementation like GCC, and simply demonstrate intent in the context of that implementation's builtin queries.
This feature could target either Section 7 or Section 8 depending on whether it intended to prioritize only the portable queries, but would probably be most useful if it targeted Section 7 so it could assume that all functionality is available.
This feature would probably require a *lot* of workshopping to determine the shape of the API (the same is true of an algorithm library, but more so for type queries), and therefore would definitely benefit from the separated, faster review process so that the Committee would not need to send it back for revision multiple times.

- lambdas or statement expressions could not really work as a CRFI, for the same reason as bit-precise integers: introducing a new expression syntax and new literal object type cannot really be meaningfully abstracted by macros or library features in any way that even remotely preserves the intent of the feature as it would be proposed.
  These would target Section 6, but unlike `defer`, would really need to be developed from the outset as first-class language feature proposals to go directly to the WG for discussion and approval, because there is no useful intermediate action that can be taken.

Finally, a dedicated CRFI is needed to define the interface for determining whether any (other) CRFI or extension is implemented, to allow individual CRFIs to depend on the presence of other features. This would probably be assigned CRFI number zero, corresponding approximately to SRFI 7 `requires` (SRFI 0 is more fundamental and describes a module system, which is out of scope for the equivalent C feature requests as it would have to target Section 6). This would exclusively target Section 8, as any feature that is selected for adoption into the IS does not need to be queried for existence using this mechanism.

## Copyright and Study Group ownership

For the CRFIs to be useful on an individual level, they need to be freely accessible to implementers and freely visible to users in a public place *before* their crystallization into the TS. Additionally, sample code must be unencumbered by copyright in order to be useful because implementers and users may want to either make use of it directly or create directly derived works. Users and group members are not necessarily generally able to assign copyright to ISO either, and in any case doing so would be inappropriate for proposals which are, by nature, *unfinished* and not yet adopted into the IS or TS; users need to be able to review and refine and propose changes, as well as make use of draft features in order to provide feedback before adoption.

The suggested working licence for CRFIs in discussion phase and before adoption by the TS is CC0 (public domain). SRFI prefers a permissive MIT-style licence, though this still imposes a minimal user requirement for attribution.

Work which is complex enough not to be suitable for dedication to the public domain may be self-selecting as too complex for a CRFI proposal.

To protect this the charter for any created Study Group needs to include, in the establishing N-document, similar wording to WG9 Ada's N367, which outlines in resolution 37-7 that new work is not produced by WG9 and WG9's role is to *approve* work created by an external body.

# Proposal

We propose to create the new mechanism in two parts, imitating a combination of WG9's relationship with MITRE and Scheme WG2's relationship with SRFI.

- First, a Community project is established, explicitly outside of the ownership of ISO and contributed to on the independent initiative of its members. A placeholder project is established at GitHub, which provides essential features for users to raise issues and make changes to sample implementations that would be missing from a "paper-oriented" service, although the project could live elsewhere (GitHub is used by SRFI).
  Membership of this project should ideally include individuals who are also members of WG14, and therefore able to give useful guidance about the direction of the group; but the group itself does not *represent* WG14. Ideally the project will also see engagement from individuals associated with implementations and library projects or in their capacity as interested users. Any individual able to meet the structure requirements of a CRFI and able to design a proposal with *any* technical merit (at all) would be invited and welcome to participate directly **without the need for a champion**.
  This project is responsible for creating proposals, assigning the CRFI numbers, discussion and reviewing proposal content, producing sample code, and progressing the state of proposals from *draft* to *final*, *withdrawn*, or *rejected*.
  Implementers wanting to build experience should be able to look at this project directly for specifications requesting feedback, and provide comments and tweaks here.

  The Community project *is* CRFI, "C Requests for Implementation", itself.

- Second, a Study Group is established within WG14. In order to protect the copyright of the Community submissions, the role of this SG is tightly constrained by its charter: it is purely responsible for producing a TS document, "C Extended Library", and to do so by selecting individual CRFIs (that have been progressed to final status) from the Community project to accept as clauses within the TS. Apart from editing the TS document, this SG produces **no original content** on its own time except up/down votes on adoption of each CRFI, with no discussion.
  Membership of this SG is restricted to existing members of the WG by ISO rules. Ideally all or most members of the SG should also be members of the Community project, so that they are familiar with the material.
  It is assumed that CRFI (the Community project) should already have gathered all necessary feedback by the time an individual proposal reaches *final* and probably from the same people who would vote on its inclusion into the TS, and therefore there is no need for feedback from the SG.
  In practice, because the *only* role of this SG is to select *final*-status CRFIs and to edit them into a document suitable for publication as a TS, it can arguably consist of only one person if necessary as nothing "interesting" should be happening in the SG's jurisdiction.
  The SG needs to exist so that it can have a charter that clarifies the copyright status of the work.

  The Study Group is "Library Evolution Study Group", distinguishing itself from CRFI.

### Questions for WG14

Does WG14 agree to create this Study Group, pending membership and content appearing within the Community project?

Is anyone from WG14 interested in explicitly joining the CRFI Community project as a project owner and proposal editor or reviewer?

Is anyone representing an implementation interested in explicitly joining the CRFI Community project to either submit feature specifications, or pick up features for experimental implementation?

Is anyone from WG14 interested in joining the SG to be involved in the TS selection and editing process? (welcome but not necessary)

## References


- Scheme Requests for Implementation
- SRFI template, SRFI 0, SRFI 7
- SRFI at GitHub
- R7RS-small
- R7RS-large
- n2683 Towards Integer Safety
- n2868 Supplemental Integer Safety
- n2897 `memset_explicit`
- n3022 Modern Bit Utilities
- n2974 Queryable pointer alignment
- n2763 Adding a Fundamental Type for N-bit
- n1657 Decimal floating-point arithmetic
- Placeholder GitHub organization and project
- linmath.h
- `defer` reference implementation
- Embedded C (TR 18037)
- WG9 (Ada) Meeting 37 minutes
- Creative Commons Zero licence