|  |  |
| --- | --- |
| Document number: | *Nnnnn=yy-nnnn* |
| Date: | 2017-09-04 |
| Project: | Programming Language C++, Library Working Group |
| Reply-to: | Ben Craig <ben dot craig at gmail dot com> |

**I. Introduction**

C++ defines a “freestanding” implementation that is intended to be used on systems with limited operating system support. The standard has a small set of headers and functionality that are required to be present in a freestanding implementation.

Some features are problematic in system programming. Exceptions, RTTI, heap management, global state, thread-local storage, and floating point support are all problematic.

This proposal seeks to expand the list of headers and functionality to include all parts of the hosted C++ implementation that don’t use problematic features, while simultaneously attempting to remove headers and functionality from the freestanding requirements that do require the problematic features.

This is intended as an early paper to seek general direction and opinions.

**II. Motivation**

Systems programmers want to sort things. They want to use move semantics. They may even want to bundle the arguments of a variadic template function into a tuple. These are all reasonable things to do on a system with no operating system and kilobytes of storage. The C++ standard even has reasonable specifications for these operations in the context of a tiny, OS-less system. However, systems programmers must currently rely on either hand-rolled code or implementer extensions in order to get these facilities.

Systems programmers don’t have a guide as to what C++ library facilities will work without trying them. The standard says atomic\_load will work; memcpy will probably work; but will stable\_sort? Standardizing the subset of implementable C++ that is usable in a freestanding environment would provide clarity here, and help to educate systems programmers.

There were some presentations at recent CppCon where others made a more full featured C++ work in the Linux and Windows kernels [Quinn2016] [Baker2017]. In both of these cases, C++ was being used in a sandboxed / pseudo-virtualized way. C++ code (with exceptions and RTTI) was being run in the kernel context, but very few calls were made between the C++ code and the operating system kernel. The C++ code was in a guest operating system. This proposal should make it reasonable to have C++ code interact closely with the host operating system, perhaps in the context of a driver.

**III. Scope**

The current scope of this proposal is limited to the standard library available to systems and embedded programming.

This paper is currently concerned with the divisions of headers and library functions as they were in C++17. I have not seen the paper that proposes which parts of the standard library go into which [sub-]modules, much less a paper that says which [sub-]modules will be freestanding. A later paper may attempt to do that.

I could see the scope increasing to the standard library availability on GPUs. It also occurs to me that the list of standard library functions suitable for a freestanding implementation is extremely similar (maybe identical?) to the list of functions that could be marked constexpr, and conditionally noexcept. This paper will not attempt to address those items though.

**IV. Impact on the standard**

Rather than list all of the facilities available to a freestanding implementation in one place, as is currently done in [compliance], the standard would tag each header, class, or function that is available in a freestanding implementation. I expect this to be a large number of small edits, but the edits wouldn’t have difficult to understand ramifications throughout the standard.

There is precedent for this kind of tagging in other specification documents. The ECMAScript Language Specification has optional support for ECMA-402 (internationalization). The impact of ECMA-402 is called out explicitly in several places. POSIX tags functions as supported in base, XSI, or in one of many option groups.

**V. Design decisions**

Even more so than for a hosted implementation, systems and embedded programmers do not want to pay for what they don’t use. As a consequence, I am avoiding all features that require global storage, even if that storage is immutable.

* Exceptions either require external jump tables or extra bookkeeping instructions.
* RTTI requires extra data in vtables and extra classes that are difficult to optimize away.
* The heap is a big set of global state. In addition, heap exhaustion is typically expressed via exception. Some embedded systems don’t even have a heap.
* Thread-local storage requires extra code in the operating system for support. In addition, if one thread uses thread local storage, that cost is imposed on other threads.
* Many embedded systems don’t have floating point hardware, and software emulated floating point can drag in large runtimes that are difficult to optimize away.
* Some functions require global or thread-local storage. These functions have been omitted or removed from the freestanding library. Examples are the locale aware functions, many of the “exit” functions, and the C random number functions.

There are a few classes that mostly work in a systems programming environment, but have a few member functions or specializations that are troublesome. In these situations, the member function or specialization may be omitted while keeping the rest.

* array::at()’s whole reason of existing is to provide a throwing operator[]. array will be supported, but without the at() member function.
* error\_code and error\_condition will be supported in general, but the non-member operator<<, and the member message() functions will not, as they require the usage of classes that won’t be supported in a freestanding environment.
* complex with integer types could still be useful, but the floating point specializations will not be required.

For the <algorithms> header, we could support the parallel algorithms, but we would only be able to support sequential execution. That doesn’t seem worthwhile enough to keep, so the execution policy overloads will be omitted.

As a QOI issue, algorithms that can operate faster when provided with an extra buffer need to take care not to cause an exception to be thrown when memory is exhausted. Implementations should support a “return nullptr;” implementation of nothrow new.

This proposal doesn’t remove problematic features from the language… but it does make it so that the freestanding standard library doesn’t require those features. Users that disable the problematic features (as is existing practice) will still have portable portions of the standard library at their disposal.

**VI. Technical Specifications**

**VII. Acknowledgements**

Similar work was done in the C++11 timeframe by Lawrence Crowl and Alberto Ganesh Barbati in N3256.

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3256.html>

CppCon talks on getting C++ support in various unusual environments:

[Baker2017] CppCon 2017: Billy Baker “Almost Unlimited Modern C++ in Kernel-Mode Applications”

[Quinn2016] CppCon 2016: Rian Quinn “Making C++ and the STL Work in the Linux / Windows Kernels"