Document Number: P0568R0
Date: 2017-02-06
Authors: Michael Wong
Project: Programming Language C++, SG14
Reply to: Michael Wong <michael@codeplay.com>

Towards Better Embedded programming support for C++ and an update on the status of SG14, two years later.

Background

While C has been the main Embedded Programming language of choice, a few brave souls have tried to champion C++ as a more ideal language for Embedded Programming. At CPPCON 2016, two of the keynotes [Saks] [Turner] were about Embedded programming and the possibly perceived problem of C++'s acceptance in the Embedded domain. Some of the issues discussed was a lack of perceived framing caused by the perception that all of C++ needs to be learned before it can be useful, despite evidence that in many cases, the equivalent C++ code can be just as fast, or even faster than the same C code. The thought was brought up that we should emphasize the type-safety aspect of C++, and prove the zero-cost claim with stringent research. This is the same question that has plagued SG14 as well.

A critical question was asked at 1:04 of the Grill the Committee Panel [grill] at CPPCON 2016 where the answer revealed some of the diverse opinion of the level of support for embedded programming in C++.

This panel and questions actually unintentionally nails the problem.

So what has C++ done for Embedded Programming

The most recent C++ effort to support Embedded Programming came from a pseudo Study Group formed by Daniel Gutson. This group never attained official status because of the uncertainty of the chair attending C++ Standard meetings regularly. The group had reasonable size and its own reflector. There was much commonality of interest of this pseudo-SG with that of SG14 which was formed in 2015 [SG14]. Specifically, both groups were interested in improving C++ on constrained resource devices and were interested in some form of alternate light-weight exception handling.

Daniel Gutson agreed to fold this pseudo-Embedded SG into SG14 at the start of 2016. The much larger size of SG14 also enabled sharing of interest and support that a proposal had a much larger backing beyond just supporting embedded programming.

SG14 status two years later

SG14 was designed to support the demands of low-latency in the Games Development, financial, and simulation environment. It specifically was formed to represent C++ members who could not attend C++ standard meetings to represent their interest.

Reflector

- https://groups.google.com/a/isocpp.org/forum/?fromgroups=#!forum/sg14

Code and proposal Staging area

- https://github.com/WG21SG14/SG1

Along with these F2F meetings, SG14 has settled on to monthly telecons in 2017 usually on the second Wednesday of the month at 2 -4 pm Eastern time. This invite can be found on the SG14 reflector.

Joining SG14 discussion on the reflector or attending the F2F meeting is voluntary and requires no invites. Current members include people from Intel, Wargaming, EA, Blizzard, Ubisoft, Flight Safety, Creative Assembly, Codeplay, Autointern, Sony, Credit Suisse, Optiver, Facebook, IBM, and many companies.

The working of the group was to hear these interest (especially as many cannot attend C++ Standard meetings), at conferences that such members would attend, such as CPPCON, Games Developers Con, and any other conferences of special interest to the members, and then have C++ Standard members represent these proposals at C++ standard meetings.
Since then a large number of SG14 meetings have been held:

- CPPCON 2015
- GDC 2016 hosted by Google JF Bastien
- STAC London 2016 held at Credit Suisse hosted by Neil Horlock
- STAC Chicago 2016 held at Neurensic hosted by Tom Rodgers and Nevin Liber
- Post-Oulu 2016 in Amsterdam held at Optiver hosted by Carl Cook
- CPPCON 2016
- Heterogeneous Computing for C++ Summit held at Lawrence Berkeley National Lab 2016 hosted by Bryce Lelbeck
- Meeting C++ 2016

All these meetings continue to be very well attended (CPPCON and GDC average 60 while Financial-focused location usually had 10). Some of these (CPPCON/GDC) were official C++ meetings with minutes and formal proposal presentations, while others tend to be brainstorm on what can be done and were recorded on the SG14 reflector. While 2015 focused on the needs of Games Developers as they were one of the initiating groups, 2016 had a distinct focus on the needs of the financial community as evidenced by the attendance at co-located companies with STAC conferences, a financial technology conference. 2017 will move to focus on the needs of Embedded programming with the kickoff of this document, and Embo++[Embo] held in Bochum in February where we will hold the next SG14 F2F meeting.

Embo++ is designed to serve the intersection of the needs of embedded programming and C++. The aim of the conference is how to make C++ better for Embedded Programming. It is not about what it takes to convert C Embedded programmers to C++ though that is one of the talks. That is a far larger issue and likely involves deeply ingrained perception, or framing of C++ from C Embedded programmers that can only be overcome with time and effort. One such perception issue is that C++ is large and complex, and it is necessary to learn all of C++, or that we need to not use certain parts of C++ such as anything involved with virtual functions or template programming. Both are false in any serious analysis.

Indeed it seems there are two specific uses of C++ in embedded programming. One is simple struct/enum translation of embedded C code that achieves and often can surpass the same C code performance without the need to learn all of C++ [Saks]. The second is the avoidance of complex C++ features such as template programming [Saks]. There was an Embedded C++ specification made almost ten years ago which removed many of these features. This disables much of the power of C++. At the time, this may have been reasonable given the state of compiler implementation of many of these features. We feel that this is not the right approach today. One of the aims of Embo++ is to show these template programming uses can be extremely powerful in embedded domain.

In a discussion held early 2017 in an SG14 meeting for Embedded SIG, members of the embedded programming domain discussed some of the issues of modern C++ and what can be done. This work will likely continue with other meetings in 2017. This paper represent much of that ongoing discussion and act to ignite a call for others to join SG14 to support and contribute to that discussion with the aim of generating proposals for C++, by demonstrating precisely what cannot be done in C++ now for Embedded programming, preferably using specific code examples in the proposal. Even if there is no solution offered, the community can still brainstorm.

If we were to survey C++17 and what it offers, some initial conclusion would be that

1. A light-weight Exception handling is one of the key to improving C++ for Embedded domain
2. C++11/14/17 Constexpr is an excellent tool but it is unfinished in that when it asserts it throws an exception
3. Standard vector is still too heavy weight because vector of ints can zero initializes everything. The guidance to use unique_ptr helps, but it can still be slower than new and delete if you don't use noexcept. It really depends on the size of the data you work with. For example, it they are Control Area Network (CAN) packets which are 0-8 bytes long, then even in the 8 byte case, it would still have 3 pointers added on top of that which more than double the size of the object that will go on the heap. The answer here may to fully control the padding. We do have a proposal for avoiding padding within structures.

It seems that some form of Lint or Coding guideline is needed at a minimum. The Guideline Support Library [GSL] was designed as such a dynamic Lint tool, which can be adaptable to real-time experiences as well as C++ standard revisions. They are based on The C++ Core Guidelines [CPPCore] which are a set of tried-and-true guidelines, rules, and best practices about coding in C++. Currently, CPPCore has C style programming for C++ with five rules, and some memory management rules. Some in the group have felt there may be a need for a section on Embedded Programming. Others have felt there may be a need for an Embedded (or close to zero-cost as possible) namespace within the C++ Standard Library.

Many people may have come and tried embedded programming in C++ and may have left. One of the main issues is interrupt handling but it is not clear where discussions of interrupt handling and their solution fit. Some have tried to do this in SG13 (HMI) as an interface/event handling topic, but it seems to be more appropriate in SG14 in the Embedded Special Interest Group (SIG).

SG14 Embedded SIG discussion

What follows is the main salient point of the discussion at the SG14 Embedded SIG where the groups started exploring the difference in style of programming in embedded domain, and start to identify how it differs from current Standard C++. This paper offers a view of a possible solution domain. A follow-on paper will detail the specific use-cases.

It was noted that the style of programming in Embedded is different then what is presumed in the C++ Standard.

In Embedded Programming, the programming style changes from thread-based to an event-based run to completion style. This is used because it is deterministic, where for example; you cannot have the phase-shift in the scheduler change the timing.

The memory management style is also different. More often it uses memory pools where you are looking for a slice off the free-list rather than from a pool where it can be fragmented.

There are also potential issues with STL containers. Since we don't know what is the slice size of a std::deque, depending on the Std library implementation which can vary, it means it can be in one of the pool, in between pools, or bigger than all the pools, which can cause waste of memory. All this speaks to have greater support for control over allocations. I believe there is a recent proposal which allows memory to grow by pool size which we endorse though I have yet to track it down.

Next the group discussed how locking is different in Embedded programming. Threading mutex blocks, but interrupt priority means the highest priority one cannot block, leading to a one-way lock. Since C++ Std::atomics can default to a lock, then it is not useful in an interrupt model.

Volatile is an issue. Is it a memory fence? GCC does not stop reordering so it is not a memory fence. Others make it a full fence, and reload from main memory. Programmers in the C++ world know that every volatile needs to be examined if you were to switch compilers.

However, the Embedded domain almost never switch compilers, but volatile is still a problem as it accumulates compiler peculiarities and makes the code even more non-portable. There was mention that one possible guideline is to use fences, or signal fences. But the question of whether fences give enough semantics to support, for example reading a register and do nothing with the result as is required in interrupt driven programming. One solution offered is to wrap all uses in library, so that any change can be done all in one place. Another possible guideline is that the user should never use volatile. It should only be used in the library.

It was observed that nearly all of Embedded issues (may be not interrupt) are also Games Development issues. Most game systems (especially consoles) use embedded technology, such as OpenGL ES. These are constrained systems that cannot upgrade the ram, or use virtual memory.

In games, being early does not really help but being late is terrible. The cliché example is embedded technology is a life-saving airbag, where being late is deadly. In VR, high-latency means people throws-up. Neither are desirable results.

This paper is meant to show that C++ is interested in taking proposals to improve embedded devices. This will have a long-term impact of ultimately making C++ more suitable in this domain.

Of existing C++ proposals, the embedded domain welcomes contract programming and static assert. They are extremely useful. They make special function register safer to interact because if you change anything in SFR, the entire data has to be reread. One member even wrote a DSL using expression templates to enforce the rules: don't corrupt reserve bits, write to read-only memory. These can increase programmer productivity.

Dan Saks [Saks] mentioned that people have this notion that C++ expensive or that you need to learn all of C++ to use it well. He also urged that we spend some time demonstrating scientifically the benefit of type safety and that C++ zero-cost abstractions are truly zero-cost.

In the past, it was hard to find all the advice on how best to use C++. Today, ISOCPP.org has become the de-facto central gathering place for the most up-to-date news on C++ usage. But that problem remains for C++ in an embedded domain guideline. It would be good to have it all in GSL the best practices. But even if that were possible, it may be we are putting standardization before the horse.

Has C++ defended its position within the embedded community very well? Has type safety been proven to improve efficiency, or is zero abstraction proven in a scientific manner? Some may say that arguing this may be pointless as it won't change people to use C++ even if we prove these cases. Will Games

programmers use C++ even if that is proven? What Dan Saks said was “if you are arguing, you are losing.”

What the real problem may be is that there are not enough libraries for embedded programming. As long as there are no libraries for embedded programming in C++, then we have a hole and no way of learning from that experience to reach a set of guideline, let alone standardization.

It was mentioned that Xilinx an FPGA company, uses metaprogramming though they may not be a full fledged C++, often without use heap allocation, but it is used side by side with Verilog or VHDL, They have no problem diving into metaprogramming for microcontrollers.

It seems that it is appropriate to create something similar to a Boost repository for embedded domain, which does not heavily use exceptions, or heap management. It is unclear whether Boost is the right place as they seem to do the exact opposite, but the Boost model for embedded library is the right model. It is also easier if we have a public library infrastructure, then guidelines will be easier to write. The group unanimously agrees that we need to have a Boost-like repository for embedded domain. It would also motivate embedded compiler developers to move their implementation faster to conformance. The tool chains must be fully be there to support modern usage.

We could also ask to just use Boost, but it is not clear whether Boost will accept this style of programming. Boost’s complex dependency is a problem as it pulls in several mega-byte of code for dependencies and that is not acceptable, as embedded engineers are used to third party libraries having bug so they often need to fully understand the code.

Finally the group discussed that Embedded should be called Bare-metal. Embedded is huge field and some devices have a heap where latency is not a problem while others have no heap

Embedded C++ Standard

Any discussion of C++ with Embedded programming needs to consider a previous attempt at a standard for embedded programming, Embedded C++ (EC++) [EC++] which is a dialect of the C++ programming language for embedded systems. It was defined by an industry group led by major Japanese central processing unit (CPU) manufacturers, including NEC, Hitachi, Fujitsu, and Toshiba, to address the shortcomings of C++ for embedded applications. It removes certain C++ features:

- Multiple inheritance
- Virtual base classes
- Run-time type information (typeid)
- New style casts (static_cast, dynamic_cast, reinterpret_cast and const_cast)
- The mutable storage class specifier
- Namespaces
- Exceptions
- Templates

Today, this specification is woefully out-of-date and seems to be still based on C++98. The group felt that removing template was the biggest problem. It may have still been useful had they not taken it out. Taking template out of C++ may have been due to the lack of efficient compiler support at the time, which was over twenty years ago. Today, it definitely does not make any sense and it disables all the truly useful of C++ features. This point to that having a living document that can be updated based on specific C++ Standard revisions and popular implementation support makes a great deal more sense for this field,

where they may indeed be using the same compiler for the next 10 years or using the latest clang or gcc compilers. All agreed that EC++ is not the right way to go for SG14.

At the end, the group formed a plan of possible action for removing forward with further deliberations.

1. Write a paper for Kona outlining our discussion and approach. This paper P0568.
2. Meet once every quarter to discuss GSL additions for embedded
3. Write papers and talks to address specific issues for C++, CPP CON, or EMBO
4. More scientific research on type safety and zero abstraction verification
5. Advance the idea of a public Boost-like repository for Embedded C++ programming
6. Urge Tools chain improvement and add emphasis on safety critical

State of the Art: C++ Solutions So far

Previous proposals that were examined for Embedded Domain are:
N3986

	Adding Standard support to avoid padding within structures
N3990	Adding Standard Circular Shift operators for computer integers
N4049	0-overhead-principle violations in exception handling
N4234	0-overhead-principle violations in exception handling - part 2
N4226	Apply the [[noreturn]] attribute to main as a hint to eliminate global object destructor calls

Of these, it was noted that N4226 was the only one that made it into C++17, and it is incredibly useful for Embedded programming as it eliminates global destructor call for main.

C++ solutions in future: SG14

Some current proposals that support this aim already exist in the proposal queue. These are ring span, clump/small vectors. These were features were led by the gaming community in SG14 but is also blessed by the Embedded SIG. Other examples include the following which was mentioned on the call but is not an exhaustive listing:

- fixed point
- Set SSO size else it locks in standard library
- In place allocators

- More control over allocation but does not use exceptions
- New feature of atomic queue for embedded

Acknowledgement

Thanks to the discussion in SG14 and those who joined the Embedded SIG call: Odin Holmes, Ronan Keryell, Paul Bendixen, John Szwast, John McFarlane, Lee Howes, shay, Mateusz Pusz, Marco Foco, Billy Baker, Brett Searles, and David Hollman.

Reference:

[SG14]: Report on SG14, a year later and future

Directions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0365r0.pdf

[Grill]: https://www.youtube.com/watch?v=CPgxw1EzC54&list=PLHTh1InhhwT7J5jl4vAhO1WvGHUUFgUQH&index=1

[Embo] **Embedded C++ Conference** https://www.embo.io/

[Saks] Extern "C": Talking to C Programmers about C++: https://www.youtube.com/watch?v=D7Sd8A6_fYU

[Turner] Rich code For Tiny Computers: https://www.youtube.com/watch?v=zBkNBP00wJE

[GSL] Guideline Support Library: http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[CPPCore] The C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines

[EC++] The Embedded C++: http://www.caravan.net/ec2plus/