Project Connected Home over IP Software

Best Practices, Coding Conventions, and Style

Revision 1
2020-04-14

Status: Draft / Active

Table of Contents

Typographic and Syntactic Conventions

The following syntactic conventions are used throughout this document:

shall
: is used to indicate a mandatory rule or guideline that must be adhered to without exception to claim compliance with this specification.

should
: is used to indicate a rule or guideline that serves as a strong preference to suggested practice and is to be followed in the absence of a compelling reason to do otherwise.

may
: is used to indicate a rule or guideline that serves as a reference to suggested practice.

1. Introduction

There are likely as many unique combinations of software engineering and development standards, conventions, and practices as there organizations that do such work. This document pulls together those that Project Connected Home over IP believes best for our organization, its efforts, and products that consume those efforts, with a particular emphasis on embedded systems with C or C++ language development and runtime environments.

This document and requirements should be considered canonical for all Project Connected Home over IP shared infrastructure software, including both RTOS-based and non-RTOS-based projects on both tightly- and loosely-constrained system platforms.

The document is broadly categorized at the highest level into:

- Best Practices and Conventions
- Format and Style

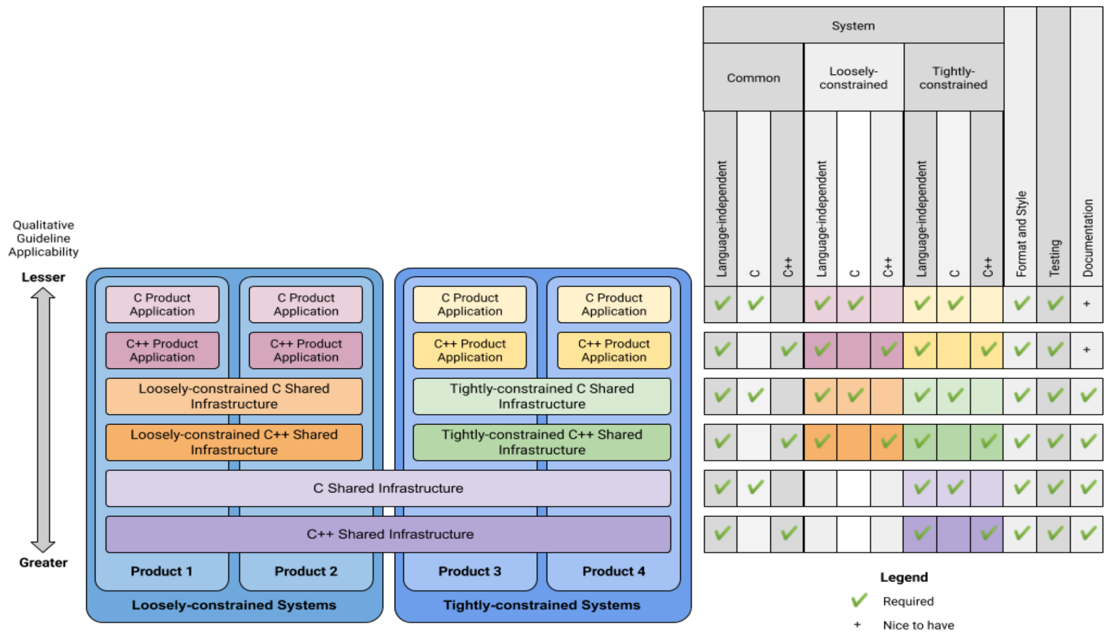
And, within conventions, further sub-categorized into those that apply to:

- Tightly-constrained
- Loosely-constrained

system platforms. Applicability to tightly-constrained systems also generally applies to shared infrastructure software that is used on both tightly- and loosely-constrained systems.

Figure 1 below attempts to illustrate both qualitative and quantitative applicability of these guidelines to Project Connected Home over IP software.

Generally, product-specific applications have the greatest flexibility and latitude in applying these guidelines to their software. Whereas, shared infrastructure bears the least flexibility and bears the greatest adherence to these guidelines.


Qualitative Guideline Applicability
Lesser
Greater
C Product Application
C++ Product Application
Loosely-constrained C Shared Infrastructure
Loosely-constrained C++ Shared Infrastructure
Tightly-constrained C Shared Infrastructure
Tightly-constrained C++ Shared Infrastructure
C Shared Infrastructure
C++ Shared Infrastructure
Product 1
Product 2
Product 3
Product 4
Loosely-constrained Systems
Tightly-constrained Systems
Legend
Required
Nice to have

	System											
	Common			Loosely-constrained			Tightly-constrained					
	Language-independent	C	C++	Language-independent	C	C++	Language-independent	C	C++	Format and Style	Testing	Documentation
C Product Application	✓	✓		✓	✓		✓	✓		✓	✓	+
C++ Product Application	✓		✓	✓		✓	✓		✓	✓	✓	+
Loosely-constrained C Shared Infrastructure / Tightly-constrained C Shared Infrastructure	✓	✓		✓	✓		✓	✓		✓	✓	✓
Loosely-constrained C++ Shared Infrastructure / Tightly-constrained C++ Shared Infrastructure	✓		✓	✓		✓	✓		✓	✓	✓	✓
C Shared Infrastructure	✓	✓					✓	✓		✓	✓	✓
C++ Shared Infrastructure	✓		✓				✓		✓	✓	✓	✓

Figure 1. Graphical summary of the qualitative and quantitative applicability to Project CHIP software.

1.1. Summary

Requirement	Description
2	Standards
2.1	Project CHIP embedded software development uses and enforces the ISO9899:1999 (aka ISO C99, C99) C language standard as the minimum. Wherever possible, particularly in non-product-specific, shared-infrastructure software, toolchain-specific (e.g GCC/GNU) extensions or the use of later standards shall be avoided or shall be leveraged through toolchain-compatibility preprocessor macros.
2.2	Project CHIP embedded software development uses the ISO14882:2011 (aka ISO C++11) language standard as a baseline for source code compatibility. Conformance with other standards, e.g., ISO14882:1998 (aka ISO C++98), may be additionally required in cases where wider portability is necessary, but in all cases, ISO C++11 is the baseline requirement.

	Wherever possible, particularly in non-product-specific, shared-infrastructure software, toolchain-specific (e.g GCC/GNU) extensions or the use of later standards shall be avoided or shall be leveraged through toolchain-compatibility preprocessor macros.
3	Conventions and Best Practices
3.1	Common
3.1.1	When in Rome, do as the Romans do.
3.1.2	Language-independent
3.1.2.1	The use of code in headers and, more specifically, the use of the non-local scope inline functions should be avoided.
3.1.2.2	Inline functions should be used judiciously.
3.1.2.3	There should be one return statement per free function or method at the end of the free function or method.
3.1.2.4	There should be no calls to the functions `setjmp` or `longjmp`.
3.1.2.5	There should be no calls to the C/C++ keyword `goto`.
3.1.2.6	C Preprocessor
3.1.2.6.1	Use of the C preprocessor should be limited to file inclusion and simple macros.
3.1.2.6.2	Macros shall not be defined within a function or a block and should be defined at the top of a file.
3.1.2.6.3	All `#else`, `#elif`, and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.
3.1.2.6.4	All `#endif` directives equal to or greater than 20 lines away from the `#if` or `#ifdef` directive to which they are related shall be decorated by language comment indicating the conditional they are associated with.
3.1.2.6.5	include Directive
3.1.2.6.5.1	Preprocessor `#include` directives in a file shall only be preceded by other preprocessor directives or comments.
3.1.2.6.5.2	Preprocessor `#include` directives shall use brace ("<") and (">") style for all public headers, including C and C++ standard library, or other first- and third-party public library headers. Preprocessor `#include` directives should use double quote ("") and ("") style for all private or relative headers.
3.1.2.6.5.3	Preprocessor `#include` directives should be grouped, ordered, or sorted as follows: ● This compilation unit's corresponding header, if any. ● C++ Standard Library headers ● C Standard Library headers ● Third-party library headers ● First-party library headers

	• Private or local headers • Alphanumeric order within each subgroup
3.1.2.6.6	The preprocessor shall not be used to redefine reserved language keywords.
3.1.2.6.7	Unused code shall not be disabled by commenting it out with C- or C++-style comments or with preprocessor `#if 0 … #endif` semantics.
3.1.2.6.8	Use of the preprocessor token concatenation operator '`##`' should be avoided.
3.1.2.6.9	The `undef` preprocessor directive should be avoided and shall never be used to undefine a symbol from a foreign module.
3.1.2.7	Data objects shall be declared at the smallest possible level of scope. No declaration in an inner scope shall hide or shadow a declaration in an outer scope. Compiler flags shall be set to flag and enforce this.
3.1.2.8	There shall be no direct or indirect use of unbounded recursive function calls.
3.1.2.9	The only functions that should not return some sort of error status are those that cannot possibly fail in either today's implementation or tomorrow's.
3.1.2.10	Wherever possible and appropriate, particularly around the management of resources, APIs should be symmetric. For example, if there is a free function or object method that allocates a resource, then there should be one that deallocates it. If there is a free function or object method that opens a file or network stream, then there should be one that closes it. Further, the names of these free functions or objects methods should be consistently named such that the names are consistent with the operation being performed and are truly opposites: • **Good:** open / close • **Bad:** open / finish • **Bad:** open In the first *bad* example, it is likely unclear to the caller whether or not `finish` will do more or less than the opposite of `open`. Moreover, in the second *bad* example, things are even worse for the developer, for either error handling or successful operation closure, there is no API whatsoever to resolve the `open` call.
3.1.2.11	Standard, scalar data types defined in *stdint.h* (C) or *cstdint* (C++) should be used for basic signed and unsigned integer types, especially when size and serialization to non-volatile storage or across a network is concerned. Examples of these are: `uint8_t`, `int8_t`, etc.
3.1.2.12	Read-only methods, global variables, stack variables, or data members are read-only should be qualified using the C or C++ `const` qualifier. Pointers or references to read-only objects or storage, including but not limited to function parameters, should be qualified using the C or C++ `const` qualifier.
3.1.2.13	

	All C, C++, Objective C, and Objective C++ headers shall use preprocessor header include guards. The terminating `endif` preprocessor directive shall have a comment, C or C++ depending on the header type, containing the preprocessor symbol introduced by the `ifndef` directive starting the guard. The symbol used for the guard should be the file name, converted to all uppercase, with any spaces (" ") or dots (".") converted to underscores ("_").
3.1.2.14	Function and Method Prototypes
3.1.2.14.1	All external free functions shall have explicit forward declarations accessible via a public header file.
3.1.2.14.2	All void functions or methods shall explicitly declare and specify the void type keyword.
3.1.3	Language-dependent
3.1.3.1	C
3.1.3.1.1	All header files intended to have C symbol linkage shall use "extern C" linkage wrappers.
3.1.3.2	C++
3.1.3.2.1	Unlike C, C++ offers an alternate way to alias data over and above a pointer, the reference, indicated by the `&` symbol. Where appropriate, the reference should be preferred to the pointer.
3.1.3.2.2	Size- and call frequency-based considerations should be made when passing scalars as to whether they should be passed by value or by constant reference; however, pass-by-value should generally be preferred.
3.1.3.2.3	The creation of empty or useless destructors should be avoided. Empty or useless destructors should be removed.
3.1.3.2.4	When you declare C++ free functions and object methods, you should avoid or minimize using default parameters.
3.1.3.2.5	When you declare C++ virtual object methods, you shall avoid using default parameters.
3.1.3.2.6	There shall be no use of global, static or otherwise, object construction. The use of scoped static object construction should be avoided.
3.1.3.2.7	Wherever possible and practical, C++ style casts should be used and preferred to the C style cast equivalent.
3.1.3.2.8	The C++ `using namespace` statement should not be used outside of object scope inside header files.
3.2	Tightly-constrained Systems and Shared Infrastructure
3.2.1	Heap-based resource allocation should be avoided.
3.2.2	There shall be no direct or indirect use of recursive function calls.

3.2.3	The use of virtual functions should be avoided.
3.2.4	The use of the C++ Standard Library shall be avoided.
3.2.5	The use of the C++ Standard Template Library (STL) should be avoided or minimized.
3.2.6	The use of the C++ templates should be avoided or minimized.
3.2.7	Code shall not use exceptions.
3.2.7.2	In the rare case that Proect CHIP code does interact with external third-party or standard library code that does throw exceptions, exceptions shall be caught locally, translated into error status, and passed up the stack.
3.2.8	Code shall not use C++ runtime type information (RTTI), including facilities that rely upon it, such as `dynamic_cast` and `typeid`.
3.3	Loosely-constrained Systems
4	Format and Style
4.1	File names should match the names and types of what is described in the file. If a file contains many declarations and definitions, the author should choose the one that predominantly describes or that makes the most sense. File contents and names should be limited in the scope of what they contain. It may also be possible that there is too much stuff in one file and you need to break it up into multiple files. File names may be extended and the implementation split across multiple such files by using a dash (“-”) in the file name to indicate the subcomponent or subset of functionality. File names should be all lower case. File names should be prefixed with ‘chip’. File names should follow the pattern *chip<component>[-<subcomponent>].{c,cpp,h,hpp,m,mm}*.
4.1.1	File extensions shall be indicative and appropriate for the type and usage of the source or header file.
4.2	Names should be descriptive but not overly so and they should give some idea of scope and should be selected such that wrong code looks wrong. Names shall not give any idea of type, such as is done with System Hungarian notation.
4.2.2	Case
4.2.2.1	C preprocessor symbols should be all uppercase.
4.2.2.2	All Project CHIP names in the C language shall be in *snake case*.
4.2.2.3	All Project CHIP class, namespace, structure, method, function, enumeration, and type names in the C++ language shall be in *upper camel case*.

	All Project CHIP instantiated names of instances of classes, namespaces, structures, methods, functions, enumerations, and types as well as method and function parameters in the C++ language shall be in *lower camel case*.
4.2.2.3.3	All Project CHIP names exported with C binding from the C++ language shall follow the conventions for names in the C++ language.
4.2.3	Symbol Qualification
4.2.3.1	All Project CHIP C public data types and free functions should have ‘chip’ prepended to their name.
4.2.3.2	All Project CHIP C++ or Objective C++ code should be in the ‘chip’ top-level namespace.
4.2.3.2.2	All Project CHIP names exported with C binding or in the global namespace in the C++ language should have ‘chip’ prepended to their name.
4.2.4	Scope
4.2.4.1	All global data shall have a `g` prepended to the name to denote global scope.
4.2.4.2	All static data shall have a `s` prepended to the name to denote static scope.
4.2.4.3	All class or structure data members shall have a `m` prepended to the name to denote member scope.
4.2.4.4	All free function or method parameters should have an `a` prepended to the name to denote function parameter scope. All free function or method parameters may alternatively have: • `in` prepended to the name to denote input, read-only • `out` prepended to the name to denote output, write-only • `io` prepended to the name to denote input/output, read/write function parameters.
4.2.4.5	All variables that do not have such prefixes shall be assumed to be function local scope.
4.2.5	Data Types
4.2.5.1	Types should be descriptive but not overly so and they should give some idea of use.
4.2.5.2	Snake case type names should have ‘`_t`’ appended to their names.
4.2.6	Function and method names should be descriptive but not overly so and they should give some idea of use.
4.2.6.2	Functions or methods that return or allocate resources or objects should have method names indicative of their resource management contract.
4.3	In compound expressions with multiple sub-expressions the intended order of evaluation shall be made explicit with parentheses.
4.4	There should be no more than one statement or variable declaration per line.
4.5	Developers shall make use of white space in their code.

4.5.2	Indentation shall be 4 space characters.
4.5.3	Conditionals shall always appear on a separate line from the code to execute as a result of the condition.
4.5.4	All scoped (i.e. stack) variable declarations should be placed together at the top of the enclosing scope in which they are used. There shall be an empty line after all such variable declarations. The names of all variable declarations should be left aligned.
4.5.5	All data member declarations should be placed together. The names of all data member declarations should be left aligned. The data member declarations for C++ classes should be placed at the end or tail of the class.
4.5.6	Braces should go on their own lines. Statements should never be on the same line following a closing brace.
4.5.7	There should be a single space after language-reserved keywords (for, while, if, etc).
4.6	Comments
	All C, C++, Objective C, Objective C++, Perl, Python, and Java code should use Doxygen to: ● Detail what the various source and header files are and how they fit into the broader context. ● Detail what the various C++ / Objective C++ namespaces are. ● Detail what the constants, C preprocessor definitions, and enumerations are. ● Detail what the globals are and how they are to be used. ● Detail what the free function and object / class methods are and how they are to be used, what their parameters are, and what their return values are. ● Detail any other important technical information or theory of operation unique and relevant to the stack that is not otherwise captured in architecture, design, or protocol documentation.
4.6.1	Every C, C++, Objective C, Objective C++, Perl, Python, Shell, and Java source file should, at minimum, have a standard, boilerplate Project CHIP file header that also describes what the file is and how, if applicable, it fits into the broader implementation.
4.6.2	Every public, and ideally private, free function and class method should likewise have a prologue comment that: ● Briefly describes what it is and what it does. ● Describes in detail, optionally, what it is and what it does. ● Describes the purpose, function, and influence of each parameter as well as whether it is an input, an output, or both. ● Describes the return value, if present, and the expected range or constraints of it.
4.6.3	Other
4.6.3.1	● **Do** use the '@' Doxygen markup style rather than the '\' markup style.

	● **Do** consider consulting the "Project CHIP Copy Style Guide" if you feel uncomfortable or unclear on your own writing style. ● **Do** also consider consulting tips on Plain Language for additional style and tone input. ● **Do** use consistent terminology and lingo. ● **Do** properly paragraph justify and wrap your documentation. ○ See your editor's documentation on how to do this (e.g. M-q in Emacs).
4.6.3.2	● **Do not** forget to document your files, enumerations, constants, classes, objects, namespaces, functions, and methods. ● **Do not** include the file name in any Doxygen file comments or directives. ○ Your editor knows the file name, source code control knows the file name, and you know the file name. ○ When it changes on the file system, having to change it in the file comments is simply an added burden. ● **Do not** include your name in any Doxygen comments or directives. ○ Source code control knows who you are and what file revisions you own. ○ We do not want our external partners knowing who you are and calling or e-mailing you directly for support. ● **Do not** include the modification date the file was last changed in Doxygen comments or directives, except for the copyright header. ○ Source code control knows when the file was last changed and the date other revisions were made. ● **Do not** include subjective or opinionated commentary or expose proprietary and confidential information not relevant to the code or APIs. ○ This content **will be** published to and for consumption by members, the CHIP community, and the general public.
5	No third-party software shall be used or included without prior legal, managerial, and technical authorization and approval.
6	All code should be designed for test and co-developed with tests. All code unit tests should be implemented using nlunit-test. All tested code should be coverable with code coverage.
7	Documentation
8	Building and Packaging

Table 1.1. Summary of best practices and coding conventions.

2. Standards

Project CHIP embedded software development adopts the minimum C and C++ standards listed in Table 2.1 below.

Language	Minimum Standard	Aliases
C	ISO9899:1999	ISO C99, C99
C++	ISO14882:2011	ISO C++11, C++11

Table 2.1. C and C++ language minimum standards adopted by Project CHIP software.

Product-specific software may elect to use later standards to the extent their software is not broadly shared inside or outside Project CHIP.

2.1. C

Project CHIP embedded software development uses and enforces the ISO9899:1999 (aka ISO C99, C99) C language standard as the minimum.

Wherever possible, particularly in non-product-specific, shared-infrastructure software, toolchain-specific (e.g GCC/GNU) extensions or the use of later standards shall be avoided or shall be leveraged through toolchain-compatibility preprocessor macros.

2.1.1. Motivation and Rationale

At the time of this writing, the C99 standard has been out for over 20 years. Project CHIP and the source code it has produced have only existed for the last five of those 20 years.

This is beyond more than adequate time for this standard to be pervasive throughout any toolchain vendor’s C compiler and saves team members from worrying about ISO9899:1990 (aka ISO C90, C90) portability issues that have long-since been solved by C99.

2.2. C++

Project CHIP embedded software development uses the ISO14882:2011 (aka ISO C++11) language standard as a baseline for source code compatibility. Conformance with other standards, e.g., ISO14882:1998 (aka ISO C++98), may be additionally required in cases where wider portability is necessary, but in all cases, ISO C++11 is the baseline requirement.

Wherever possible, particularly in non-product-specific, shared-infrastructure software, toolchain-specific (e.g GCC/GNU) extensions or the use of later standards shall be avoided or shall be leveraged through toolchain-compatibility preprocessor macros.

2.2.1. Motivation and Rationale

At the time of this writing, the C++11 standard has been out for over seven years in one form or another and has been twice supplanted, once by C++14 and again by C++17. Project CHIP and the source code it has produced are nearly concurrent with C++11.

This is beyond more than adequate time for this standard to be pervasive throughout any toolchain vendor’s C++ compiler and saves team members from worrying about portability issues that have long-since been solved by C++11.

By contrast, ISO14882:2014 (aka ISO C++14, C++14) and ISO14882:2017 (aka ISO C++17, C++17), are still insufficiently broad and pervasive in their toolchain support to warrant the introduction of dependencies on these standards across all software.

Note, that while C++11 is the C++ language bar, per Figure 1, embrace of C++11 language- and library-specific features should be approached thoughtfully and carefully, depending on the deployment context. A loosely-constrained embedded Linux or Darwin application may want a broad embrace of C++11 language and library features whereas a tightly-constrained piece of shared infrastructure may want to eschew C++11 entirely or conditionally depend on language-specific features, where appropriate.

That said, suitable portability mnemonics, for example, via the C preprocessor should be used where possible and appropriate to maximize code portability, particularly for shared embedded product software. An example of such a portability mnemonic is shown in Listing 2.1 below.

```
#ifdef __cplusplus
# if __cplusplus >= 201103L
# define __chipFINAL final
# else
# define __chipFINAL
# endif
#else
#define __chipFINAL
#endif
```

Listing 2.1. Using the C preprocessor to provide a portability mnemonic for the C++11 and later *final* keyword.

3. Conventions and Best Practices

3.1. Common

The following sections summarize those best practices that are independent of particular nuances of either the C or C++ languages.

3.1.1. When in Rome

The most important convention and practice in the Project CHIP embedded software is "*When in Rome…*", per the quote below.

> "*If you should be in Rome, live in the Roman manner; if you should be elsewhere, live as they do there.*"
>
> -St. Ambrose

3.1.1.1. Motivation and Rationale

At this stage in the work group's and the team's life cycle, it is rare the project or subsystem that is entirely new and built from scratch. More often than not, development will involve extending, enhancing, and fixing existing code in existing projects.

When in this situation, it is mandatory you observe how things are done in this context and do the best that you can to follow the prevailing conventions present. Not doing so can lead to readability and maintenance problems down the line and will likely earn you the disapprobation of the code's *owner* or other team members.

Your extensions or fixes to existing code should be **indistinguishable**, stylistically, from the original code such that the only way to ascertain ownership and responsibility is to use the source code control system's change attribution (aka *blame*) feature.

If you find the conventions so foreign or otherwise confusing, it may be best to let whoever *owns* the file make the necessary changes or seek the counsel of others in the group to find out what the right thing to do is. Never just start changing code wholesale for personal reasons without consulting others first.

3.1.2. Language-independent

3.1.2.1. Avoid the Use of Inlines

The use of code in headers and, more specifically, the use of the non-local scope inline functions should be avoided.

3.1.2.1.1. Motivation and Rationale

Without proper care and oversight and a continuous attention to quantitative measurement, excessive inlining can lead to excessive code growth.

Care must be taken and measurements must made to assess the impact of the:

1. number of times a function is called
2. size of the call site setup for each call

From there, relative comparisons should be made between the size of the function instantiated and shared once through calls versus inlining that same function at each call site.

3.1.2.1.2. Exceptions

Simple setters and getters such as those shown in Listing 3.1 are fine since the compiler can efficiently optimize these and make their overhead as low as a direct data member access.

```
inline int SetFoo(int aFoo)
{
    int status = SUCCESS;

    if (mFoo == aFoo)
        status = -EALREADY;
    else
        mFoo = aFoo;

    return status;
}
```

Listing 3.1. Using the inline keyword for a simple setter free function.

As always, make quantitative size assessments using the `size` command line tool to assess trade-offs between using an inline or not.

3.1.2.2. Use Inlines Judiciously

Inline functions should be used judiciously.

3.1.2.2.1. Motivation and Rationale

See “*3.1.2.1. Avoid the Use of Inlines*” above.

The judicious use of inlines can make an overall program smaller and more efficient. Make sure you can quantitatively measure and prove that is the case when you use inlines for small functions, such as simple data member setters or getters.

3.1.2.3. Return Statements

There should be one return statement per free function or method at the end of the free function or method.

3.1.2.3.1. Motivation and Rationale

Use of return at places other than the end of functions can lead to all sorts of problems. A function has a single entry point. This makes sense and makes it easy to know where execution begins (what variable state looks like, what parameters are passed in and how they are validated, etc.).

For the same reasons it also make sense to have a single return point. With each additional return point that is introduced into a function you increase the number of places that must be examined and, potentially, updated each time changes are made to that function. In code with multiple return points, failure to properly handle each and every return point when the function is changed is a common way that bugs are introduced. Of these bugs, resource leaks are one of the most common. System state may also become inconsistent (and this can lead to some very nasty problems to find).

Very often a function starts out simple and having multiple returns seems harmless enough. That may indeed be true in some cases (e.g. `if (...) return 0; else return 1;`). What later often happens, however, is that the function gets much more complicated and rather than eliminate the multiple return points because it's no longer such a simple piece of code, more return points get added because that seems easier to do than fix things the right way.

If you go through a few iterations of this, you wind up with needless copy and pasting of code at each return point. Naturally, this does not always happen, but it does happen often enough to make use of multiple return points a very risky and unnecessary proposition.

Additionally, with C++ in particular, the compiler must determine how to unwind the stack frame and destroy any scoped objects for each particular `return` site. While the compiler should, in theory, be intelligent about eliminating duplication among the unwind and destruction code it generates, in practice, many are not leading to nearly duplicate code generated at each `return` site. Consequently, coding for a single return site may also make your code smaller.

3.1.2.4. Non-local Goto

There should be no calls to the functions `setjmp` or `longjmp`.

3.1.2.4.1. Motivation and Rationale

With very few exceptions, it is possible to write any piece of code without using non-local gotos. It will be both more readable and maintainable without them.

3.1.2.4.2. Exceptions

Using these constructs to implement threading interfaces, such as *POSIX Threads*, is not uncommon. Generally-speaking, however, no one should be writing and implementing such interfaces.

3.1.2.5. Local Goto

There should be no calls to the C/C++ keyword `goto`.

3.1.2.5.1. Motivation and Rationale

With very few exceptions, it is possible to write any piece of code without using non-local gotos. It will be both more readable and maintainable without them.

Should you find that you're tempted to add a goto to solve some coding problem, first consider restructuring what is there, factoring some of the code into separate functions and/or using a different algorithm. Most of the time you'll find that the goto is no longer needed.

3.1.2.5.2. Exceptions

The use of local gotos for the purposes of common error handling blocks and single points of function return (see *3.1.2.3. Return Statements*) at the bottom of a function is the first allowable exception to this.

The implicit use of local gotos when used in the context of an approved declarative assertion macro library, such as *nlassert*, is the second allowable exception to this.

3.1.2.6. C Preprocessor

3.1.2.6.1. Scope

Use of the C preprocessor should be limited to file inclusion and simple macros.

3.1.2.6.1.1. Motivation and Rationale

Broad and extensive use of the C preprocessor, like C++ templates, can make code difficult to debug and to search through. From a debugging perspective, unlike C++ templates, all the more so because C preprocessor macros are not strongly typed in the way that inline or template functions are.

3.1.2.6.2. define Directive Scope

Macros shall not be defined within a function or a block and should be defined at the top of a file.

3.1.2.6.2.1. Motivation and Rationale

Including preprocessor macros within a function or block can lead the author or subsequent readers to believe the symbol defined is limited to that scope when, in fact, it is not.

Further, such inclusion makes the code more difficult to read.

3.1.2.6.3. endif Directive Scope

All `#else`, `#elif`, and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

3.1.2.6.3.1. Motivation and Rationale

Debugging and trying to intuit the behavior of preprocessor macros across several different file scopes is incredibly difficult.

3.1.2.6.4. endif Directive Distance

All `#endif` directives equal to or greater than 20 lines away from the `#if` or `#ifdef` directive to which they are related shall be decorated by language comment indicating the conditional they are associated with.

3.1.2.6.4.1. Motivation and Rationale

While some code editors have built-in support for correlating blocks of preprocessor conditional directives, many do not. This means that it may be difficult for code readers and reviewers to visually and mentally correlate long conditional blocks. Providing such comments helps them perform that correlation.

3.1.2.6.5. include Directive

3.1.2.6.5.1. Placement in File

Preprocessor `#include` directives in a file shall only be preceded by other preprocessor directives or comments.

3.1.2.6.5.1.1. Motivation and Rationale

The typical expectation is that such directives occur at the top of files. Enforcing that they be there reinforces that convention and follows the rule of least astonishment.

When you are trying to debug or visually analyze code, it is typically a large and unpleasant surprise that there is an `#include` directive buried deep in the file.

3.1.2.6.5.2. Style

Preprocessor `#include` directives shall use brace (“<”) and (“>”) style for all public headers, including C and C++ standard library, or other first- and third-party public library headers.

Preprocessor `#include` directives should use double quote (““‘) and (““‘) style for all private or relative headers.

3.1.2.6.5.2.1. Motivation and Rationale

First, using separate styles for public versus private headers makes it very easy for the code reader to quickly and easily identify which headers are public and which are private.

Second, the brace style uses broader and different search rules relative to the narrower search rules used for double quote style.

3.1.2.6.5.3. Ordering

Preprocessor `#include` directives should be grouped, ordered, or sorted as follows:

1. This compilation unit's corresponding header, if any
2. C++ Standard Library headers
3. C Standard Library headers
4. Third-party library headers
5. First-party library headers
6. Private or local headers

7. Alphanumeric order within each subgroup

Assuming a subsystem with the following files:

- Private Header and Source Files
  - chipfoo/chipfoobar.h
  - chipfoo/chipfoobar.c
  - chipfoo/chipbaz.h
  - chipfoo/chipbaz.c
  - chipfoo/chipbar.h
- Public Header Files
  - chipfoo/include/chipfoo/chipfoo.h

An example for the *chipfoo/chipfoobar.c* source compilation unit is shown in Listing 3.2.

```
...

#include “chipfoobar.h”

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include <acme/tools.h>

#include <chip/chiputilities.h>

#include <foo/chipfoo.h>

#include “chipbar.h”
#include “chipbaz.h”

...
```

Listing 3.2. Conformant example of include directive ordering.

3.1.2.6.5.3.1. Motivation and Rationale

This convention, first and foremost, with the preferred ordering, if the compilation unit’s corresponding header omits any necessary includes, the build of the compilation unit will break. Thus, this rule ensures that build breakages due to headers that themselves insufficiently specify their own include directive dependencies show up first for the people working on these files, not for unrelated developers working on other packages and components.

Second, this rule establishes a convention in the presence of otherwise random or arbitrary header organization.

Beyond that, it ensures that include directives occur mostly in dependency order with standards and system libraries acting as the base of that dependency graph and taking primacy in symbol definition.

Finally, the convention gives developers an easy, at-a-glance view of where to focus in when looking for a particular header.

3.1.2.6.6. Redefining Language Keywords

The preprocessor shall not be used to redefine reserved language keywords.

3.1.2.6.6.1. Motivation and Rationale

Redefining reserved language keywords leads to surprising and difficult to debug behavior for code reviewers and maintainers.

3.1.2.6.7. Commenting Out or Disabling Code

Unused code shall not be disabled by commenting it out with C- or C++-style comments or with preprocessor `#if 0` … `#endif` semantics.

3.1.2.6.7.1. Motivation and Rationale

Code should either be actively maintained and “in” the source base for a purpose or removed entirely. Code that is disabled in this way is generally sloppy and does not convey a sense of certainty and direction in the code.

Anyone who is interested in the history of a particular source code file should use the source code control system to browse it.

Code that is debug- or test-only should be moved to a conditionally compiled test source file or conditionalized with an appropriate WITH_DEBUG, WANT_DEBUG, WITH_TESTS, WANT_TESTS, or some similar such preprocessor mnemonic that can be asserted from the build system.

Please see “*7.3.2. Functions and Methods*” for more detail on commenting functions and methods.

3.1.2.6.8. Token Concatenation

Use of the preprocessor token concatenation operator '##' should be avoided.

3.1.2.6.8.1. Motivation and Rationale

While this is a very powerful tool of the C preprocessor and can eliminate a lot of otherwise duplicative typing, it makes code very difficult to debug as it makes symbols difficult, if not impossible, to search for in non-preprocessed code.

3.1.2.6.9. undef Directive

The `undef` preprocessor directive should be avoided and shall never be used to undefine a symbol from a foreign module.

3.1.2.6.9.1. Motivation and Rationale

Through the use of the `undef` directive, it becomes possible to have multiple definitions of macros with the same name in different file scopes with different behavior. Consequently, depending on the order of header inclusion, debugging and figuring out which symbol is in effect and what behavior will result can be both tedious and difficult.

3.1.2.6.9.2. Exceptions

Frequently undef is used to limit the scope of the definition of a local preprocessor symbol within a module, as shown in Listing 3.3, ensuring it does not propagate beyond that scope. This use of undef is acceptable and encouraged.

```
...

#if SOMETHING
#define __MY_INLINE inline
#else
#define __MY_INLINE __inline__
#endif

__MY_INLINE foo(int aBar);
__MY_INLINE bar(int aFoo);

#undef __MY_INLINE

...
```

Listing 3.3. Using the undef preprocessor directive to control symbol scope.

3.1.2.7. **Object** Scope

Data objects shall be declared at the smallest possible level of scope.

No declaration in an inner scope shall hide or shadow a declaration in an outer scope. Compiler flags shall be set to flag and enforce this.

3.1.2.7.1. Motivation and Rationale

The larger the scope of a data object, the more exercise and thought that must be given to managing access to and the lifetime of the data object and the more difficult it will be for later evaluators and maintainers of the code to reason about the object.

Likewise, while the C and C++ language specifications are very clear about scope precedence, having hidden or shadowed data declarations can make it very difficult for later evaluators and maintainers of the code—even the original author distanced by time—to reason about the correct behavior of the code.

3.1.2.8. Unbounded Recursion

There shall be no direct or indirect use of unbounded recursive function calls.

3.1.2.8.1. Motivation and Rationale

On our embedded systems there is no paging backing store and, for deeply embedded systems, memory resources are constrained and there is no dynamic memory allocation. Consequently, thread stacks are practically limited and fixed in size and, as a result, unbounded recursion can lead to rapid exhaustion in stack resources or the unnecessary expansion of those resources leading to non-real time system behavior.

Either reconstruct a recursive algorithm to be non-recursive, thereby keeping stack usage constant for that algorithm relative to its recursive peer or place depth bounds on the recursion such that maximum resource usage can be predicted and accounted for in system resource budgeting and provisioning.

3.1.2.9. Error Handling

The only functions that should not return some sort of error status are those that cannot possibly fail in either today's implementation or tomorrow's.

3.1.2.9.1. Motivation and Rationale

Murphy, per the quote below, perhaps expresses the expectations about errors and faults best:

> “*Anything that can go wrong, will go wrong.*”
>
> -Murphy

In light of this, plan for errors and faults.

Once a function has been released to the world (i.e. checked in), it will be used. If you first use a `void` return type for a free function or method and then later decide that it should return status of some type, all code that calls that function needs to be revisited. In addition, the people that write code that calls that function need to be informed and remember that all of a sudden they need to start checking a return status.

Think carefully about error handling not only when generating new APIs but also when implementing code within the bodies of APIs and interfaces. Look at and understand the error schemes used by other parts of the code base before just rolling your own.

Look at return status from functions you call and think about what to do. Real thought should go into error handling. A mental response along the lines of, “I don't know what to do when I get this error, so I'll throw it away." is generally **unacceptable**.

If there is nothing reasonable that you can do with an error, then the most likely situation is that some other object has called your function and you should cleanup and communicate (i.e. pass up or return) the error status to the caller. An example of bubbling up an error message is a failed resource request call, as shown in Listing 3.4 below.

```
int bar(foo **p)
{
    int status = 0;

    ...

    if (p == NULL)
    {
        status = -EINVAL;
    }
    else
    {
        ...


        p = foo_malloc();

        if (p == NULL)
        {
            status = -ENOMEM;
        }
        else
```

```
            {
                ...
            }
        }

    return status;
}
```

Listing 3.4. Correctly and successfully handling interface and allocation errors by passing return status up to the caller.

Typically, an application will call some other module to allocate a resource, which may call through to something else to do something equivalent to `open` or `allocate`. At the lowest level, where the `open` actually fails, it probably is not appropriate to display an error dialog or print a message (other than an assert/debug type message). In fact it is probably not appropriate to do **anything** at any level below the caller that started the whole process except exit gracefully and pass the error to the caller.

In general, modules used by an application should not take any action beyond defensive, preventing "bad things from happening" due to errors. It should be up to the application to take positive action to recover from an error.

Do not use macros that can be compiled out as a method of handling errors. Since they can be compiled out they don't always do anything. Macros for error flagging can be useful to log unexpected error conditions at run time but should never take the place of explicit error handling.

Project CHIP’s technologies sit at a unique confluence of consumer electronics and the mission- and application-criticality typically associated with medical devices or telecommunications and networking equipment. Consequently, it is important that the software running on them be resilient and robust in the face of errors and faults.

3.1.2.10. Symmetric APIs

Wherever possible and appropriate, particularly around the management of resources, APIs should be symmetric.

For example, if there is a free function or object method that allocates a resource, then there should be one that deallocates it. If there is a free function or object method that opens a file or network stream, then there should be one that closes it.

Further, the names of these free functions or objects methods should be consistently named such that the names are consistent with the operation being performed and are truly opposites:

- **Good:** open / close
- **Bad:** open / finish
- **Bad:** open

In the first *bad* example, it is likely unclear to the caller whether or not `finish` will do more or less than the opposite of `open`. Moreover, in the second *bad* example, things are even worse for the developer, for either error handling or successful operation closure, there is no API whatsoever to resolve the `open` call.

3.1.2.10.1. Motivation and Rationale

Having symmetric APIs pays two dividends. The first is that it makes it visually and logically easier for humans reading and auditing the API and its callers to make sense of it and to identify places where leaks or other breakdowns in the API contract are occurring.

The second is that it provides developers a means by which to perform both exceptional error handling clean up as well as successful operation closure, leaving system resources in a consistent and reusable state.

3.1.2.10.2. Exceptions

There are isolated cases in which the system operation semantics are such that some resources are singletons and are initialized on system start-up and then are never disposed of and cannot be reinitialized until the system restarts.

This may be particularly true for constrained systems in which the resources needed to implement symmetric APIs aren’t justified because they are unneeded. Ideally, the API designer includes a design in which the API is symmetric and can be fully tested as such but includes compile-time directives in which the deallocation or deinitialization half of the API may be optionally compiled out.

3.1.2.11. Use C *stdint.h* or C++ *cstdint* for Plain Old Data Types

Standard, scalar data types defined in *stdint.h* (C) or *cstdint* (C++) should be used for basic signed and unsigned integer types, especially when size and serialization to non-volatile storage or across a network is concerned.

Examples of these are: `uint8_t`, `int8_t`, etc.

3.1.2.11.1. Motivation and Rationale

These types have been effectively standardized since C99 and should be available on every platform and provide more neutral portability than OS-specific types such as `u8`, `UInt8`, etc.

Moreover, because these are pervasive, you do not need to spend any time and energy as a developer and engineer creating more such types on your own—the compiler vendors have already done the hard work for you.

Additionally, using traditional scalar types such as `int`, `short`, or `long` have portability issues where data width is concerned because these types are sized differently on different processor architectures and ABIs for those architectures. For example, a long is 32-bits on some architectures and 64-bits on others.

3.1.2.12. Constant Qualifiers

Read-only methods, global variables, stack variables, or data members are read-only should be qualified using the C or C++ `const` qualifier.

Pointers or references to read-only objects or storage, including but not limited to function parameters, should be qualified using the C or C++ `const` qualifier.

Examples are shown in Listing 3.5 through Listing 3.7 below.

```
extern int Bar(const uint8_t *aFirst);
extern int Foo(int aFirst, int aSecond, const int &aThird);
```

Listing 3.5. Use of the const qualifier with read-only pass-by-pointer and -reference parameters.

```
class Foo
{
public:
    Foo(void);
    ~Foo(void);

    int GetBar(void) const;

private:
    int mBar;
};
```

Listing 3.6. Use of the const qualifier with read-only object methods.

```
int Bar(const uint8_t *aFirst)
{
    const size_t length = sizeof (struct chipFoo);
    ...

    ...
}
```

Listing 3.7. Use of the const qualifier with read-only variables.

3.1.2.12.1. Motivation and Rationale

Use of the `const` keyword helps both the compiler and the reader of the code. It helps the compiler and reader alike by informing them of the contractual intent that no modifications to the variable, parameter, or object state will or should be made. The compiler, in particular, can leverage this intent insight by optimizing the code appropriately.

3.1.2.13. Header Include Guard

All C, C++, Objective C, and Objective C++ headers shall use preprocessor header include guards.

The terminating `endif` preprocessor directive shall have a comment, C or C++ depending on the header type, containing the preprocessor symbol introduced by the `ifndef` directive starting the guard.

The symbol used for the guard should be the file name, converted to all uppercase, with any spaces (“ “) or dots (“.”) converted to underscores (“_”).

An example of this is shown in Listing 3.8 below.

```
#ifndef CHIPEXAMPLE_HPP
#define CHIPEXAMPLE_HPP
...
#endif // CHIPEXAMPLE_HPP
```

Listing 3.8. Example use of a header include guard for the file *chipexample.hpp*.

3.1.2.13.1. Motivation and Rationale

This prevents redundant header inclusion on compilers that cannot detect this internally and stems hard-to-debug symbol redefinition errors. Also, see “*3.1.2.6.4. endif Directive Distance*”.

3.1.2.14. Function and Method Prototypes

3.1.2.14.1. Forward Declarations

All external free functions shall have explicit forward declarations accessible via a public header file.

3.1.2.14.1.1. Motivation and Rationale

This is a useful hint to the compiler to support ensuring that it can check and verify that the implementation of a function matches its public declaration. In addition, this a simple courtesy to other developers. A user of your public function should be able to include an appropriate header that contains your public, forward declaration and prototype rather than creating the prototype in their own source.

3.1.2.14.2. Void Declarations

All void functions or methods shall explicitly declare and specify the `void` type keyword.

3.1.2.14.2.1. Motivation and Rationale

The void type is an intrinsic type just as is `char`, `short`, and `long`. If you declared a function with a single char parameter, you would declare it as:

```
int foo(char);
```

and invoke it as:

```
foo(‘c’);
```

similarly, when declaring a void parameter, the type should be declared explicitly:

```
int bar(void);
```

where it would be invoked as:

```
bar();
```

This style makes it easy to visually identify and differentiate, at a glance, a declaration from an invocation.

3.1.3. Language-dependent

3.1.3.1. C

3.1.3.1.1. C / C++ Linkage Wrappers

All header files intended to have C symbol linkage shall use “extern C” linkage wrappers as shown in Listing 3.9.

```
...

#ifndef __cplusplus
extern “C” {
#endif

/* C Symbol Definitions Are Placed Within This Brace Scope */

#ifndef __cplusplus
}
#endif

...
```

Listing 3.9. Example of using “extern C” linkage wrappers in C header files.

3.1.3.2. C++

In Project CHIP, the main idea with C++ is to think of C++ as if it were C plus classes, C+ if you will.

The C++ language provides us stricter static type checking and better access control. However, as you will read below, some of the other C++ features could be a double-edged sword, felling even the most seasoned and experienced of programmers.

If you are working in the context of a tightly constrained system or on shared infrastructure code to be used across both tightly- and loosely-constrained system, the use of C++ should be limited to this range.

3.1.3.2.1. Prefer Passing Parameters by Reference to Pointer

Unlike C, C++ offers an alternate way to alias data over and above a pointer, the reference, indicated by the `&` symbol.

Where appropriate, the reference should be preferred to the pointer.

3.1.3.2.1.1. Motivations and Rationale

The semantics of references are stronger than their weaker pointer cousins, mandating a reference must always alias a real object whereas a pointer can alias just about anything, including NULL. Because of this, callees dealing with references need not be burdened with performing NULL pointer checks.

While NULL pointers can be great for indicating optionality of a parameter, if there is no valid case in which anything other than a real object can be passed to your interface, stipulate a reference rather than a pointer.

Consider Listing 3.10 below. The overhead imposed by the checking required of the pointer-based version of `Subtract` imposes 57% to 200% overhead relative to its reference-based version when compiled using G++ 4.8.2 for *x86_64-unknown-linux-gnu*.

```
#include <errno.h>

struct Arguments
{
    int mMinuend;
    int mSubtrahend;
    int mDifference;
};

extern int Subtract(Arguments *aArguments);
extern int Subtract(Arguments &aArguments);

int Subtract(Arguments *aArguments)
{
    int status = 0;

    if (aArguments == 0)
    {
        status = -EINVAL;
    }
    else
    {
        aArguments->mDifference =
            aArguments->mMinuend - aArguments->mSubtrahend;
    }

    return status;
}

int Subtract(Arguments &aArguments)
{
    aArguments.mDifference =
        aArguments.mMinuend - aArguments.mSubtrahend;

    return 0;
}
```

Listing 3.10. Comparison of pass-by-pointer vs. pass-by-reference semantics.

3.1.3.2.2. Passing Base Scalars

Size- and call frequency-based considerations should be made when passing scalars as to whether they should be passed by value or by constant reference; however, pass-by-value should generally be preferred.

3.1.3.2.3.1. Motivation and Rationale

Depending on the size of the scalar, compiler, optimization level, instruction set architecture (ISA), and application binary interface (ABI), both the callee and caller overhead for scalar (i.e. non structure) types can vary depending on whether the values are passed by value or by constant reference. Data are shown below in Table 3.1 and Table 3.2 for armv7-none-eabi and x86_64-unknown-linux-gnu, respectively.

Architecture	**armv7-none-eabi**					
Pass By	**Value**			**Constant Reference**		
Type	**Caller**	**Callee**	**Total**	**Caller**	**Callee**	**Total**
bool	26	22	48	30	18	48
uint8_t	26	22	48	30	18	48
uint16_t	30	22	52	32	18	50
uint32_t	28	16	44	28	18	46
uint64_t	38	22	60	34	22	56
float	28	16	44	32	18	50
double	48	22	70	40	22	62

Table 3.1. Comparison of caller and callee overhead of pass-by-value versus pass-by-constant reference for scalar types for the armv7-none-eabi binary interface.

Architecture	**x86_64-unknown-linux-gnu**					
Pass By	**Value**			**Constant Reference**		

Type	Caller	Callee	Total	Caller	Callee	Total
bool	10	4	14	21	6	27
uint8_t	10	5	15	21	7	28
uint16_t	10	5	15	26	8	34
uint32_t	10	4	14	24	6	30
uint64_t	12	5	17	25	8	33
float	13	6	19	24	10	34
double	13	6	19	30	10	40

Table 3.2. Comparison of caller and callee overhead of pass-by-value versus pass-by-constant reference for scalar types for the x86_64-unknown-linux-gnu binary interface.

Using the ARM ABI as a representative example, while the callee side may be smaller for pass-by-constant-reference, the caller side is larger. Consequently, for more than one invocation of a function or method, the economics of pass-by-value will often prevail.

3.1.3.2.3. Eliminate Unnecessary Destructors

The creation of empty or useless destructors should be avoided.

Empty or useless destructors should be removed.

3.1.3.2.3.1. Motivation and Rationale

Destructors, whether virtual or not, have an impact on code space. For a project with a lot of classes, all the useless destructors add up.

When the destructor is virtual, all the worse, because the linker cannot dead strip them. Only you, as the developer or engineer, have the insight and context to know when such destructors may be eliminated.

3.1.3.2.4. Default Parameters

When you declare C++ free functions and object methods, you should avoid or minimize using default parameters.

3.1.3.2.4.1. Motivation and Rationale

The code in Listing 3.11 provides a good example of what is happening behind the scenes with the compiler.

```
#include <stdio.h>

extern void foo(int aParameter, int aOptionalParameter = 1234, void *
anOptionalPointer = NULL);

void foo(int aParameter, int anOptionalParameter, void * anOptionalPointer)
{
        printf("Received %d %d %p\n", aParameter, anOptionalParameter,
anOptionalPointer);
}

int main(void)
{
    foo(42);
}
```

Listing 3.11. The use of C++ default parameters in a free function.

The program prints the expected result but there is subtle a piece of magic happening courtesy of the compiler that has a cost associated to it: it wastes valuable code space.

As shown in Listing 3.12, the compiler has to pass the default values `1234` and `NULL` to the function `foo` even if the caller has not specified them, accepting the defaults. While at first glance, this does not seem like a big deal, if `foo` is called hundreds of times, the waste adds up to kilobytes.

The cost of preparing these 2 extra parameters could be as low as 4 bytes on architectures such as ARM; but, more frequently, it is 8 bytes or more.

```
000000000040055c <main>:
  40055c:   55                      push   %rbp
  40055d:   48 89 e5                mov    %rsp,%rbp
  400560:   ba 00 00 00 00          mov    $0x0,%edx         # Pass NULL
  400565:   be d2 04 00 00          mov    $0x4d2,%esi       # Pass 1234
  40056a:   bf 2a 00 00 00          mov    $0x2a,%edi        # Pass 42
  40056f:   e8 b9 ff ff ff          callq  40052d <foo(int, int, void*)>
  400574:   b8 00 00 00 00          mov    $0x0,%eax
  400579:   5d                      pop    %rbp
  40057a:   c3                      retq
  40057b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

Listing 3.12. The x86_64 assembly for the main routine from Listing 3.11.

The preferred way to address this, courtesy of C++ overloading, is shown in Listing 3.13.

```
extern void foo(int aParameter);
extern void foo(int aParameter, int aOptionalParameter);
extern void foo(int aParameter,
    int aOptionalParameter,
    void * aOptionalPointer);

void foo(int inParameter)
{
    foo(inParameter, 1234);

    // Alternatively, if you are worried about performance and want to
    // avoid the secondary, implicit call overhead.

    foo(inParameter, 1234, NULL);
}

void foo(int aParameter, int anOptionalParameter)
{
    foo(aParameter, anOptionalParameter, NULL);
}

void foo(int aParameter, int anOptionalParameter, void * anOptionalPointer)
{
    printf("Received %d %d %p\n",
           aParameter,
           anOptionalParameter,
           anOptionalPointer);
}

int main(void)
{
    foo(42);
}
```

Listing 3.13. Minimizing the overhead of C++ default parameters by using function overloading.

This time around, the result is identical, but the compiler calls the function `foo` without pushing three (one explicit and two implicit, default) parameters like it did in the example in Listing 3.12.

3.1.3.2.4.2. Exceptions

If you have leaf inlines which do not call other non-inline functions, then the compiler can optimize the parameters without having to perform a formal call site with default argument passing, as shown in Listing 3.14.

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

static inline int foo(int aParameter, int aOptionalParameter = 1234, void *
anOptionalPointer = NULL)
{
    const uint32_t magic = 0x34ac91f0;
    int            iterator;
    int            result = 0;

    for (iterator = aParameter; iterator < anOptionalParameter; iterator++)
    {
        result = result + (iterator ^ magic);

        if (anOptionalPointer != NULL)
        {
            result += *static_cast<int *>(anOptionalPointer);
        }
    }

    return result;
}

int main(void)
{
    foo(42);

    return 0;
}
```

Listing 3.14. Exception to minimizing the overhead of C++ default parameters when leaf inline functions are involved.

However, following the original guideline is still **strongly** recommended as subsequent authors of the code that may, absent a comment in the code about it, enhance or extend the API to call non-leaf or non-inline functions introducing the undesired overhead.

3.1.3.2.5. Default Parameters with Virtual Methods

When you declare C++ virtual object methods, you shall avoid using default parameters.

3.1.3.2.5.1. Motivation and Rationale

If different default parameters are used from superclass to subclass or if the superclass uses them but the subclass does not, this can be confusing to the reader and extremely difficult to debug.

Inline forwarding functions or the use of the technique described in “*Avoid or Minimize Default Parameters*” should be used.

3.1.3.2.6. Global and Scoped Static Construction

There shall be no use of global, static or otherwise, object construction.

The use of scoped static object construction should be avoided.

Examples of this are shown in Listing 3.15 below.

```
class Bar
{
public:
    Bar(int aValue) { mValue = aValue; }
    int GetValue(void) const { return mValue; }

private:
    int mValue;
};

// Global Construction
const Bar gClass(15);

Bar & GetSharedBarInstance(void)
{
    // Scoped Static Construction
    static Bar sBar(42);

    return sBar;
}
```

Listing 3.15. Global and scoped static object construction.

3.1.3.2.6.1. Motivation and Rationale

Whether in embedded systems or not, there are a host of reasons for which this is problematic:

- The C++ standard does not define a means by which the construction of global objects are ordered.
    - Consequently, it happens in a platform-specific and arbitrary order.
    - There are no standard ways of indicating dependencies and forcing an order without linker script or C++ runtime start-up customization that make both very platform-, compiler-, and application-dependent.
- Scope static objects must be constructed once and only once, when the scope is first executed.
    - This is guaranteed using some runtime semantics which not only have large overhead but is also difficult to orchestrate on deeply-embedded systems.
        - This entails an extra piece of in-RAM state to effect a once variable for each object as well as code to set and check it.
        - This variable setting and checking must also be thread safe, so the compiler also generates a lock and lock management code as well.
- For both global and scoped static construction, the C++ runtime tries, per the standard to call the destructors when `main` entry point exits.
    - It does that by calling `atexit()` which, in turn, uses the heap to dynamically allocate the necessary closures for these destructors.
    - Unfortunately, many of Project CHIP's deeply-embedded runtimes do not support heap-based allocation.
    - Whether or not `atexit` and `malloc` are called, there are nearly 100 bytes (on ARM architectures) of unneeded stub function calls per construction, for the thread-safe once variable management.

3.1.3.2.7. C++-style Casts

Wherever possible and practical, C++ style casts should be used and preferred to the C style cast equivalent.

3.1.3.2.7.1. Motivation and Rationale

One of the advantages of using C++ over C is its stronger type checking capabilities. This extends to the C++ style casts.

The C style cast of:

```
(type)(rvalue)
```

is a blunt instrument in that it tells the compiler, "Trust me. I know what I am doing; treat this *rvalue* as this *type*. It'll be fine. Really.", even if it is technically incorrect or illogical. By contrast, the C++ casts:

- const_cast
- static_cast
- reinterpret_cast

provide varying degrees of power and are strongly-typed, preventing you—or at least making it more difficult—from making casts that are technically incorrect or illogical.

3.1.3.2.8. Avoid using namespace Statements in Headers

3.1.3.2.8.1. Motivation and Rationale

By doing this, you are effectively forcing every other module that includes the header to also be using the namespace. This causes namespace pollution and generally defeats the purposes of namespaces. Fully-qualified symbols should be used instead.

3.2. Tightly-constrained Systems and Shared Infrastructure

Applicability to tightly-constrained systems also generally applies to shared infrastructure software that is used on both tightly- and loosely-constrained systems.

3.2.1. Avoid Heap-based Resource Allocation

Heap-based resource allocation should be avoided.

3.2.1.1. Motivation and Rationale

As emphasized throughout this document, the software produced by Project CHIP is consumed both inside and outside Project CHIP, across a variety of platforms. The capabilities of these platforms are broad, spanning soft real-time, deeply-embedded systems based on FreeRTOS to richer, softly-embedded systems based on non-RTOS platforms such as Linux or Darwin. While the latter are apt to have fully-functional heaps, the former explicitly do not.

Consequently, when planning new or extending existing projects, consider the platforms to which the project is targeted. If the platforms include those deeply-embedded platforms absent functioning heaps, then heap-based resource allocation is absolutely forbidden. If not, consideration should be made to the cost / benefit trade-offs of heap-based allocation and, if possible, it should be avoided.

3.2.1.2. Alternatives

In either case, recommended resource allocation alternatives are:

- In Place Allocation and Initialization
- Pool-based Allocators
- Platform-defined and -assigned Allocators

3.2.1.2.1. Use In Place Allocation and Initialization

Regardless of whether the source code and runtime are C or C++, the first step is creating storage for the object being allocated and initialized. For simple plain-old-data (POD) data structures, this can be done by just allocating the structure at an appropriate scope. Alternatively, *raw* storage can be allocated and then cast. However, great care must be taken with the latter approach to ensure that natural machine alignments and language strict-aliasing rules are observed. With the simple data structure declaration, the compiler does this on your behalf. With the raw approach, you must do this.

Once the storage has been allocated, then use symmetric initializers and deinitializers such as those, for example, for *pthread_attr_t*. An example is shown below in Listing 3.16.

```
#include <pthread.h>
#include <stdint.h>

...

#if USE_STRUCT_STORAGE
// Allocate the structure directly.
static pthread_attr_t sThreadAttributes;

#elif USE_RAW_STORAGE
// Allocate the structure using "raw" storage.
#define chipDEFINE_ALIGNED_VAR(name, size, align_type) \
    align_type name[(((size) + (sizeof(align_type) - 1)) /
sizeof(align_type))]

static chipDEFINE_ALIGNED_VAR(sThreadAttributes, sizeof (pthread_attr_t),
uint64_t);

#endif // USE_STRUCT_STORAGE

int foobar(void)
{
    int                 retval;
    int                 status;
    pthread_t           thread;
    pthread_attr_t *    attrs = (pthread_attr_t *)&sThreadAttributes;

    // Now "construct" or initialize the storage.
    retval = pthread_attr_init(attrs);

    if (retval == 0)
    {
        retval = pthread_create(&thread, attrs, NULL, NULL);
        if (retval == 0)
        {
            status = pthread_join(thread, NULL);

            if (status != 0)
            {
                retval = status;
            }

            status = pthread_attr_destroy(attrs);

            if (status != 0)
            {
                retval = status;
            }
        }
    }

    return retval;
}
```

Listing 3.16. Using in place allocation and initialization in C or C++.

For non-scalar types and objects such as C++ classes, this gets slightly trickier since C++ constructors and destructors must be accounted for and invoked. Fortunately, C++ has *placement* new which handles this. Listing 3.17 below modifies Listing 3.16 above using C++ placement new to ensure the class is properly constructed before initialization and destructed after deinitialization.

```
#include <new>

#include <pthread.h>
#include <stdint.h>

...

class ThreadAttributes
{
public:
    ThreadAttributes(void) {};
    ~ThreadAttributes(void) {};

    operator pthread_attr_t *(void) { return &mAttributes; }

private:
    pthread_attr_t mAttributes;
};

// Allocate the structure using "raw" storage.
#define chipDEFINE_ALIGNED_VAR(name, size, align_type) \
    align_type name[(((size) + (sizeof (align_type) - 1)) / sizeof
(align_type))]

static chipDEFINE_ALIGNED_VAR(sThreadAttributes, sizeof (ThreadAttributes),
uint64_t);

int foobar(void)
{
    int                retval = -1;
    int                status;
    pthread_t          thread;
    ThreadAttributes * ta;
    pthread_attr_t *   attrs;

    ta = new (&sThreadAttributes) ThreadAttributes;

    if (ta != NULL)
    {
        attrs = static_cast<pthread_attr_t *>(*ta);

        // Now "construct" or initialize the storage.
        retval = pthread_attr_init(attrs);

        if (retval == 0)
        {
            retval = pthread_create(&thread, attrs, NULL, NULL);

            if (retval == 0)
            {
                status = pthread_join(thread, NULL);

                if (status != 0)
                {
                    retval = status;
                }

                status = pthread_attr_destroy(attrs);

                if (status != 0)
                {
                    retval = status;
                }
            }
        }

        ta->~ThreadAttributes();
    }

    return retval;
}
```

Listing 3.17. Using C++ placement new for in place allocation and initialization.

3.2.1.2.2. Use Pool-based Allocators

In place allocation allows the successful allocation, initialization, deinitialization, and deallocation of a single object allocated from preallocated storage. However, if the desire exists for a fixed, configurable pool of objects where 0 to *n* of such objects can be allocated at any one time, a pool allocator for that specific object type must be created.

As shown in Listing 3.18 below, a pool allocator for a Foo class of CHIP_FOO_COUNT objects is effected, assuming the existence of another helper class, StaticAllocatorBitmap, which uses a bitmap to track the storage of objects from a static array of storage.

```
#include <stdint.h>

class Foo
```

```
{
public:
    Foo(void);
    Foo(const Foo &inFoo);
    ~Foo(void);
};

static DEFINE_ALIGNED_VAR(sFooAllocatorBuffer,
sizeof (StaticAllocatorBitmap), uint32_t);
static StaticAllocatorBitmap *sFooAllocator;

static void CreateFooAllocator(void *inStorage,
                               const StaticAllocatorBitmap::size_type
&inStorageSize,
                               const StaticAllocatorBitmap::size_type
&inElementCount,
                               StaticAllocatorBitmap::InitializeFunction
inInitialize,
                               StaticAllocatorBitmap::DestroyFunction
inDestroy)
{
    sFooAllocator = new (sFooAllocatorBuffer)
        StaticAllocatorBitmap(inStorage,

                              inStorageSize,
                              inElementCount,
                              inInitialize,
                              inDestroy);

}

static StaticAllocatorBitmap &GetFooAllocator(void)
{
    return *sFooAllocator;
}

static void *FooInitialize(AllocatorBase &inAllocator, void *inObject)
{
    memset(inObject, 0, sizeof(Foo));

    return inObject;
}

static void FooDestroy(AllocatorBase &inAllocator, void *inObject)
{
    return;
}

int Init(void)
{
    static const size_t sFooCount = CHIP_FOO_COUNT;
    static chipAllocatorStaticBitmapStorageDefine(sFooStorage, Foo,
sFooCount, uint32_t, sizeof (void *));
     int retval = 0;

    CreateFooAllocator(sFooStorage,
                       sizeof (sFooStorage),
                       sFooCount,
                       FooInitialize,
                       FooDestroy);

    return retval;
}

Foo * FooAllocate(void)
{
    Foo *foo;

    foo = static_cast<Foo *>(GetFooAllocator().allocate());

    return foo;
}

void FooDeallocate(Foo *inFoo)
{
    GetFooAllocator().deallocate(inFoo);
}
```

Listing 3.18. Using pool-based allocators.

3.2.1.2.3. Use Platform-defined and -assigned Allocators

This is a variation on both in place allocation and pool-based allocation in that it completely delegates resource allocation to the system integrator and the platform on which the particular software subsystem is running.

The advantage of this approach is that it allows the platform to decide how resource allocation will be handled and allows the package to scale independently of platform resource allocation.

The package may define default implementations for a few types of platform allocation strategies, such as heap-based allocators and pool-based allocators.

There are a range of granularities for achieving this type of delegation, depending on the desired size of the API surface, as shown in Listing 3.19 through Listing 3.22 below.

```
chipPlatformInitFooAllocator();
chipPlatformInitBarAllocator();
…
foo = chipPlatformGetFooAllocator().allocate();
…
chipPlatformGetFooAllocator().deallocate(foo);
…
bar = chipPlatformGetBarAllocator().allocate();
…
chipPlatformGetBarAllocator().deallocate(bar);
```

Listing 3.19. Using a common allocator method pattern with unique allocators per object, accessed from a unique singleton access per allocator.

```
chipPlatformInitAllocator(CHIP_FOO_T);
chipPlatformInitAllocator(CHIP_BAR_T);
…
foo = chipPlatformGetAllocator(CHIP_FOO_T).allocate();
…
chipPlatformGetAllocator(CHIP_FOO_T).deallocate(foo);
```

```
…
bar = chipPlatformGetAllocator(CHIP_BAR_T).allocate();
…
chipPlatformGetAllocator(CHIP_BAR_T).deallocate(bar);
```

Listing 3.20. Using a common allocator method pattern with unique allocators per object, accessed from a common singleton access with type per allocator.

```
chipPlatformInitFooAllocator();
chipPlatformInitBarAllocator();
…
foo = chipPlatformFooAllocate();
…
chipPlatformFooDeallocate(foo);
…
bar = chipPlatformBarAllocate();
…
chipPlatformBarDeallocate(bar);
```

Listing 3.21. Using unique allocators per object.

```
chipPlatformInitAllocator(CHIP_FOO_T);
chipPlatformInitAllocator(CHIP_BAR_T);
…
foo = chipPlatformAllocate(CHIP_FOO_T);
…
chipPlatformDeallocate(CHIP_FOO_T, foo);
…
bar = chipPlatformAllocate(CHIP_BAR_T);
…
chipPlatformBarDeallocate(CHIP_BAR_T, bar);
```

Listing 3.22. Using a common allocator pattern with unique allocators per object, accessed from a common interface with type per allocator.

3.2.2. Recursion

There shall be no direct or indirect use of recursive function calls.

3.2.2.1. Motivation and Rationale

On deeply embedded systems, particularly those with constrained resources and no dynamic memory allocation, thread stacks are both limited and fixed in size. Consequently, unbounded or even shallow recursion can lead to rapid exhaustion in stack resources and the unnecessary expansion of those resources.

Typically, a recursive algorithm (while clever) can be restructured to be non-recursive, thereby keeping stack usage constant for that algorithm relative to its recursive peer.

3.2.3. Avoid the Use of Virtual Functions

The use of virtual functions should be avoided.

3.2.3.1. Motivation and Rationale

While virtual functions definitely solve a very real design problem and serve a very real purpose, their use is frequently misunderstood and, consequently, they are frequently overused when a more efficient solution could be employed. In addition, their use and inclusion exacts very real overhead in the system. That overhead comes in two forms:

- The addition of the first virtual function to an object adds an implicit pointer to the object for a virtual function dispatch table, or *vtable*, growing the object by 4 or 8 bytes, depending on the processor architecture.
- All virtual method calls are dereferenced or indirected through this *vtable*, implying that the overhead to call a virtual function is higher than for a non-virtual function.

Virtual functions are most effective and applicable where there are multiple **runtime** instances of an object, each of which effects a common base interface but may have a very different underlying implementations, implementations which need to be differentiated at **runtime**.

If there will only ever be one concrete implementation of an object in the system at runtime, then other techniques such as a static symbol dependency can be used.

If at the beginning of the project you thought it was a good idea to keep the design as flexible as possible with virtual methods sprinkled around your objects, but the need for them did not materialize, revisit and eliminate the unnecessary virtual methods.

3.2.4. C++ Standard Library

The use of the C++ Standard Library shall be avoided.

3.2.4.1. Motivation and Rationale

The C++ Standard Library, including the string and stream libraries, are large and provide limited utility over their C Standard Library counterparts with all of the overhead of exceptions and runtime type information that we further want to avoid.

Use the C Standard Library instead.

3.2.5. C++ Standard Template Library (STL)

The use of the C++ Standard Template Library (STL) should be avoided or minimized.

3.2.5.1. Motivation and Rationale

Much of the C++ Standard Template Library relies on dynamic memory allocation and exceptions, the latter of which are forbidden by this document.

In addition, while there are at this point in time few if any bugs in the STL itself, code that uses STL can be difficult to debug.

Finally, in the hands of the non-expert, injudicious use of STL can quickly lead to unexpected code growth as instantiations of templates with different types can lead to many unique but otherwise identical template instantiations.

3.2.6. C++ Templates

The use of the C++ templates should be avoided or minimized.

3.2.6.1. Motivation and Rationale

Code that uses C++ templates can be difficult to debug. However, they are far preferable to using the C preprocessor, when and where absolutely necessary.

In addition, in the hands of the non-expert, injudicious use of STL can quickly lead to unexpected code growth as instantiations of templates with different types can lead to many unique but otherwise identical template instantiations.

3.2.7. Exceptions

Code shall not use exceptions.

3.2.7.1. Motivation and Rationale

Exceptions add extra and substantial overhead to the code to ensure that, when an exception is thrown, all the appropriate error handling and object destruction occurs across whatever call frames the exception might possibly traverse before it is caught.

Per this document, best practice and style is to use return error codes or status and to propagate them iteratively back up the call stack chain. Error handling and object destruction still needs to occur; however, it is explicitly generated by the developer rather than automatically and pessimistically by the compiler and is therefore smaller.

3.2.7.2. Exception Conversion

In the rare case that Project CHIP code does interact with external third-party or standard library code that does throw exceptions, exceptions shall be caught locally, translated into error status, and passed up the stack, as prescribed in *3.1.2.9. Error Handling*.

An example of this is shown below in Listing 3.23.

```
int chipFooAllocate(chipFoo *& aFoo)
{
        int retval = 0;

        try
        {
                aFoo = sFooInstace().allocate();
        }

        catch (std::bad_alloc &anException)
        {
                retval = -ENOMEM;
        }

        return retval;
}
```

Listing 3.23. Local termination of exceptions and propagation as error status.

3.2.8. Runtime Type Information (RTTI)

Code shall not use C++ runtime type information (RTTI), including facilities that rely upon it, such as `dynamic_cast` and `typeid`.

3.2.8.1. Motivation and Rationale

Runtime type information adds extra and substantial overhead to the code to ensure that any generic base pointer can be generically introspected to find out what type it is.

Rather than incurring this overhead for every object in the system, object-specific introspection should be used when and where necessary to provide comparable but application-specific functionality.

3.3. Loosely-constrained Systems

There are no conventions and best practices guidelines specific to loosely-constrained systems at this time.

4. Format and Style

4.1. File Names

File names should match the names and types of what is described in the file.

If a file contains many declarations and definitions, the author should choose the one that predominantly describes or that makes the most sense.

File contents and names should be limited in the scope of what they contain. It may also be possible that there is too much stuff in one file and you need to break it up into multiple files.

File names may be extended and the implementation split across multiple such files by using a dash (“-”) in the file name to indicate the subcomponent or subset of functionality.

File names should be all lower case.

File names should be prefixed with ‘chip’.

File names should follow the pattern *chip<component>[-<subcomponent>].{c,cpp,h,hpp,m,mm}*.

Examples are provided in Table 4.1 and in the list below.

Type	**Language**	**File Name**
chip_hash_table	C	chiphashtable.[ch]
chipHashTable	C or C++	chiphashtable.{c,cpp,h,hpp}
chip::HashTable	C++	chiphashtable.{cpp,hpp}

Table 4.1. File names for specific data types and languages.

- *chipfoo-types.h*
    - Basic public types for the foo subsystem.
- *chipfoo-private.[ch]*
    - Private types and interfaces for the foo subsystem.
- *chipfoo-test.[ch]*
    - Test types and interfaces for the foo subsystem.

4.1.1. Extensions

File extensions shall be indicative and appropriate for the type and usage of the source or header file, as directed by Table 4.2 below.

Extension	Type	Usage
.h	Header	C-only or C and C++ with appropriate __cplusplus guards.
.hpp	Header	C++-only.
.c	Source	C-only.
.cpp	Source	C++-only.
.m	Source	Objective C-only.
.mm	Source	Objective C++-only.

Table 4.2. File extensions for specific file types and usages.

4.2. Naming

Names should be descriptive but not overly so and they should give some idea of scope and should be selected such that *wrong code looks wrong*.

Names shall not give any idea of type, such as is done with *System Hungarian* notation.

4.2.1. Motivation and Rationale

Short, vague variable names—outside of basic loop iterators, no matter how convenient they might be to type, are just a bad idea because their use and context is often ambiguous.

The days of FORTRAN and BASIC's requirements for tiny variable names are long gone, so let us take advantage of that.

4.2.2. Case

4.2.2.1. C Preprocessor

C preprocessor symbols should be all uppercase.

4.2.2.1.1. Motivation and Rationale

This convention makes it clear that such symbols are typically preprocessor macros.

4.2.2.2. C

All Project CHIP names in the C language shall be in *snake case*.

4.2.2.2.1. Motivation and Rationale

This is previously-agreed-upon long-standing convention.

4.2.2.3. C++

All Project CHIP class, namespace, structure, method, function, enumeration, and type names in the C++ language shall be in *upper camel case*.

All Project CHIP instantiated names of instances of classes, namespaces, structures, methods, functions, enumerations, and types as well as method and function parameters in the C++ language shall be in *lower camel case*.

Examples are shown in Listing 4.1.

```
namespace chip
{

namespace ExampleNamespace
{

class ExampleClass
{
public:
    enum ExampleEnumeration {
        kFirstEnumeration,
        kSecondEnumeration
    };

    typedef int Status;

public:
    ExampleClass(void);
    ~ExampleClass(void);

    Status ExampleMethod(int aParameter);
    Status SetEnumeration(ExampleEnumeration aEnumeration)
    {
        Status retval = 0;

        if (mEnumeration == aEnumeration)
        {
            retval = -EALREADY;
        }
        else
        {
            mEnumeration = aEnumeration;
        }

        return retval;
    }

    ExampleEnumeration GetEnumeration(void) const;

private:
    ExampleEnumeration mEnumeration;
};

ExampleClass::ExampleEnumeration GetEnumeration(ExampleClass &aClass);

}; // namespace ExampleNamespace

}; // namespace chip
```

Listing 4.1. Example usage of lower and upper camel case in the C++ language.

4.2.2.3.1. Motivation and Rationale

This is previously-agreed-upon long-standing convention.

4.2.2.3.2. Exceptions

The top level Project CHIP namespace ‘chip’.

4.2.2.3.3. C in C++

All Project CHIP names exported with C binding from the C++ language shall follow the conventions for names in the C++ language, per 4.2.2.3.

4.2.2.3.3.1. Motivation and Rationale

This follows the module or subsystem convention (ostensibly C++ here) in preference to the language convention.

4.2.3. Symbol Qualification

4.2.3.1. C

All Project CHIP C public data types and free functions should have ‘chip’ prepended to their name.

Examples are shown in Listing 4.2.

```
enum chip_example_enumeration
{
    first_enumeration,
    second_enumeration
};

typedef chip_example_enumeration chip_example_enumeration;

struct chip_example_struct
{
    chip_example_enumeration   enumeration;
    void *                     data;
};

extern int chip_function(chip_example_enumeration enumeration);

extern chip_example_enumeration chip_global_example_enumeration;
```

Listing 4.2. Example usage of ‘chip’ symbol qualification in the C language.

4.2.3.1.1. Motivation and Rationale

The ‘chip’ qualifier prevents symbol collisions with standard, system, platform, or vendor symbols and further makes it clear which code is owned and managed by Project CHIP and which is not.

4.2.3.2. C++

All Project CHIP C++ or Objective C++ code should be in the ‘chip’ top-level namespace.

An example is shown in Listing 4.3 below.

```
namespace chip
{

...

}; // namespace chip
```

Listing 4.3. Project CHIP ‘chip’ top-level C++ namespace.

4.2.3.2.1. Motivation and Rationale

This prevents symbol collisions with standard, system, platform, or vendor symbols and further makes it clear which code is owned and managed by Project CHIP and which is not.

Code in the ‘chip’ namespace is, by extension, exempt from further qualifying its names and symbols with ‘chip’.

4.2.3.2.2. C in C++

All Project CHIP names exported with C binding or in the global namespace in the C++ language should have ‘chip’ prepended to their name.

4.2.3.2.2.1. Motivation and Rationale

The ‘chip’ qualifier prevents symbol collisions with standard, system, platform, or vendor symbols and further makes it clear which code is owned and managed by Project CHIP and which is not.

4.2.4. Scope

4.2.4.1. Global Data

All global data shall have a `g` prepended to the name to denote global scope.

4.2.4.2. Static Data

All static data shall have a `s` prepended to the name to denote static scope.

4.2.4.3. Object Data

All class or structure data members shall have a `m` prepended to the name to denote member scope.

4.2.4.4. Function and Method Parameters

All free function or method parameters should have an `a` prepended to the name to denote function parameter scope.

All free function or method parameters may alternatively have:

- `in` prepended to the name to denote input, read-only
- `out` prepended to the name to denote output, write-only
- `io` prepended to the name to denote input/output, read/write

function parameters.

4.2.4.4.1. Motivation and Rationale

At minimum, qualifying parameter names with `a` (for "argument") helps to avoid shadow variable conflicts with local, stack variables and gives a clear indication to the reader, particularly for longer functions, which values being manipulated are local, stack variables and which are function or method arguments.

In addition, the alternative `in`, `out`, and `io` provides further self-documentation to the code for the benefit of the reader and helps to align the code with the Doxygen `@param[in|out|io]` documentation directives.

4.2.4.5. Other

All variables that do not have such prefixes shall be assumed to be function local scope.

4.2.5. Data Types

4.2.5.1. Descriptiveness

Types should be descriptive but not overly so and they should give some idea of use.

4.2.5.1.1. Motivation and Rationale

Any developer working on a new and sufficiently complex technology has a large amount of concepts and nomenclature to keep straight in his or her head. While he or she can always consult the documentation for a newly-encountered data type, self-documenting data types alleviate the developer of some of that mental load.

4.2.5.2. **Snake Case** Decoration

Snake case type names should have ‘`_t`’ appended to their names.

4.2.5.2.1. Motivation and Rationale

The ‘`_t`’ qualifier makes it visually clear which symbols are types and which are instantiated type names.

4.2.6. Functions and Methods

Function and method names should be descriptive but not overly so and they should give some idea of use.

4.2.6.1. Motivation and Rationale

Any developer working on a new and sufficiently complex technology has a large amount of concepts and nomenclature to keep straight in his or her head. While he or she can always consult the documentation for a newly-encountered function or method name, self-documenting function or method names alleviate the developer of some of that mental load.

4.2.6.2. Resource-managing Function and Method Names

Functions or methods that return or allocate resources or objects should have method names indicative of their resource management contract.

Recommendations are shown in Table 4.3 and examples are shown in Listing 4.4.

Name Should Contain	Return Semantics	Object Ownership Transferred
Get	Returns a weak reference to an object.	No
Copy	Returns a copy of an object.	Yes
Retain	Returns a strong reference to an object.	Yes

Table 4.3. Recommended naming conventions for resource-managing functions or methods.

```
class Foo;

const Foo * GetFoo(void);      // Get an immutable weak reference to Foo.
Foo *       GetFoo(void);      // Get a mutable weak reference to Foo.
const Foo & GetFoo(void);      // Get an immutable weak reference to Foo.
Foo &       GetFoo(void);      // Get a mutable weak reference to Foo.
Foo         CopyFoo(void);     // Copy Foo by value.
Foo *       CopyFoo(void);     // Copy Foo by pointer.
Foo *       RetainFoo(Foo *); // Get a mutable strong reference to Foo.
```

Listing 4.4. Examples of naming conventions for resource-managing functions or methods.

4.2.6.2.1. Motivation and Rationale

Resource management is an area of software development rife with the potential for bugs, resource leaks in particular. By choosing function and method names indicative of the resource management contract, this increases the likelihood that the developer will take the correct and appropriate actions to manage the lifetime of the object returned.

4.3. Parenthesis and Compound Logical Expressions

In compound expressions with multiple sub-expressions the intended order of evaluation shall be made explicit with parentheses.

4.3.1. Motivation and Rationale

Even though the C and C++ language specifications are very explicit about the order of evaluation, not every developer or engineer is equally clear and knowledgeable. Making the order explicit makes intent clear and prevents later code evaluators and maintainers from having to be C and C++ specification experts to intuit the correct intent and operation of the code.

4.4. Compound Expressions and Statements

There should be no more than one statement or variable declaration per line.

4.4.1. Motivation and Rationale

See “*4.5. White Space*” below. Following a one expression or statement per line makes the code easier to read by making it clearer exactly what is happening on each line. When multiple expressions or statements are made per line, it can be visually difficult to see those expressions following the first.

4.4.2. Exceptions

A single exception is the C / C++ `for` keyword loop control, where the three controlling statements (initialization, loop bound, and increment) may be placed on a single line.

4.5. White Space

Developers shall make use of white space in their code.

4.5.1. Motivation and Rationale

Proper use of white space can make it much easier to quickly scan through a piece of code to find something that you are looking for without the need to read each and every character.

Further, adding white space makes code less dense which makes it harder for characters to get accidentally "lost" or missed when reading expressions.

4.5.2. Indentation

Indentation shall be 4 space characters.

4.5.2.1. Motivation and Rationale

The interpreted width of tabs is variable, depending on the editor in use. The use of spaces ensures consistent formatting of code, no matter what editor is used to view or edit the code.

4.5.3. Conditionals

Conditionals shall always appear on a separate line from the code to execute as a result of the condition.

4.5.3.1. Motivation and Rationale

As with all white space style issues, this improves readability by ensuring the code to execute is clearly distinguished and visible from the conditional. Consider the improved readability of `do_something` between the second of the two examples in Listing 4.5.

```
// Non-conformant example
if (is_something()) { do_something(); }

// Conformant example
if (is_something())
{
    do_something();
}
```

Listing 4.5. Non-conformant and conformant examples of improving conditional readability with white space.

4.5.4. Scoped Variable Declarations

All scoped (i.e. stack) variable declarations should be placed together at the top of the enclosing scope in which they are used.

There shall be an empty line after all such variable declarations.

The names of all variable declarations should be left aligned.

An example of these three guidelines is shown in Listing 4.6.

```
void foo(void)
{
    static const bar   = { 1, 2, 3, 4 };
    const        magic = 0x9c3a5b1f;
    int          iterator;
    int          status

    /* Actual Body of the Function Starts Here */

      ...
```

```
        iterator = 0;

        while (/* some condition */)
        {
            const foobar_t *foobar = chipGetFoobarElement(i);

            ...

            iterator++;
        }

        ...
}
```

Listing 4.6. Grouping of, use of a new line after, and left alignment of the names of scoped variable declarations.

4.5.4.1. Motivation and Rationale

This improves readability by allowing the code auditor quickly and easily see, at a single glance, all of the stack or automatic variables *in play* for the current scope in the following block and to further easily discern the types of those variables from their names.

Finally, for C++ code, this follows Resource Acquisition Is Initialization (RAII) practices

4.5.5. Data Member Declarations

All data member declarations should be placed together.

The names of all data member declarations should be left aligned.

The data member declarations for C++ classes should be placed at the end or tail of the class.

An example of these guidelines are shown in Listing 4.6 and Listing 4.7.

```
struct chipFoo
{
    uint8_t *    mData;
    int          mBar;
    const void * mPointer;
    int          mParameter;
    int          mResult;
}
```

Listing 4.6. Grouping and left alignment of the names of data member declarations for structs.

```
class chipBar
{
public:
    chipBar(void);

    int GetBar(void) const;
    const void *GetPointer(void) const;
    int GetParameter(void) const;
    int GetResult(void) const;

private:
    int          mBar;
    const void * mPointer;
    int          mParameter;
    int          mResult;
}
```

Listing 4.7. Grouping, placement, and left alignment of the names of data member declarations for classes.

4.5.6. Braces

Braces should go on their own lines.

Statements should never be on the same line following a closing brace.

4.5.6.1. Motivation and Rationale

Braces are like code white space (see "*4.5. White Space*" above), not just a syntactical requirement of the compiler.

4.5.7. Keywords

There should be a single space after language-reserved keywords (for, while, if, etc).

4.5.7.1. Motivation and Rationale

As with all white space style issues, this improves readability by ensuring the keyword is clearly distinguished and visible from its arguments. Consider the improved readability of `for` between the second of the two examples in Listing 4.8.

```
// Non-conformant example
for(frame = 0; frame < limit; frame++)
```

```
// Conformant example
for (frame = 0; frame < limit; frame++)
```

Listing 4.8. Non-conformant and conformant examples of improving keyword readability with white space.

4.6. Comments

Due to Project CHIP's infrastructure nature, it will be consumed by other teams, both inside and outside Project CHIP. Therefore it is critical that how they work, how they behave, and how they are interfaced with are clearly documented.

In support of this effort Project CHIP uses Doxygen to markup (or markdown) all C, C++, Objective C, Objective C++, Perl, Python, and Java code to:

- Detail what the various source and header files are and how they fit into the broader context.
- Detail what the various C++ / Objective C++ namespaces are.
- Detail what the constants, C preprocessor definitions, and enumerations are.
- Detail what the globals are and how they are to be used.
- Detail what the free function and object / class methods are and how they are to be used, what their parameters are, and what their return values are.
- Detail any other important technical information or theory of operation unique and relevant to the stack that is not otherwise captured in architecture, design, or protocol documentation.

4.6.1. File

Every C, C++, Objective C, Objective C++, Perl, Python, Shell, and Java source file should, at minimum, have a standard, boilerplate Project CHIP file header that also describes what the file is and how, if applicable, it fits into the broader implementation.

Canonical examples for C, C++, Objective C, and Objective C++ and Python, Perl, and shell are shown in Listing 7.1 and Listing 7.2 below.

```
/*
 *    Copyright (c) <Create year>[-<Last modified year>] Project CHIP
Authors.
 */

/**
 *    @file
 *        <Brief description>
 *
 *      [<Detailed description>]
 */
```

Listing 7.1. Standard, boilerplate Project CHIP file header for C, C++, Objective C, and Objective C++..

```
#
#    Copyright (c) <Create year>[-<Last modified year>] Project CHIP
Authors.
#

##
#    @file
#        <Brief description>
#
#      [<Detailed description>]
#
```

Listing 7.2. Standard, boilerplate Project CHIP file header for Perl, Python, shell, and make.

where:

- *<Create year>* is the year the file was created.
- *<Last modified year>* is, optionally, the year the file was last modified if it is different from <Create year>.
- *<Brief description>* is a succinct description of what the file is.
- *<Detailed description>* is, optionally, a more in-depth description of what the file is and how it fits into the broader context.

For header files, a good prologue for *<Brief description>* is "This file defines...", describing what is being defined or declared. Likewise, for source files, a good prologue for *<Brief description>* is "This file implements...", describing what is being implemented. Usually, copy-and-pasting the brief description from the header to the source and changing the prologue from "*defines*" to "*implements*" is sufficient.

The *<Detailed description>*, if present, could be a link to one or more of the architecture, design, or protocol specifications or some more in depth but still succinct information about where the file and what it defines or implements fit into the broader design or implementation.

4.6.1.1. Motivation and Rationale

The motivation and rationale for this is not from a legal perspective and as a consequence is not in opposition to guidance from legal. However, when Project CHIP provides a substantial amount of our code as reference code and as an SDK to third-parties, this makes it very clear—and consistently so—what code belongs to and is authored by Project CHIP and what is not.

4.6.2. Functions and Methods

Every public, and ideally private, free function and class method should likewise have a prologue comment that:

- Briefly describes what it is and what it does.

- Describes in detail, optionally, what it is and what it does.
- Describes the purpose, function, and influence of each parameter as well as whether it is an input, an output, or both.
- Describes the return value, if present, and the expected range or constraints of it.

An example is shown in Listing 27 below for C, C++, Objective C, and Objective C++. Adapt as appropriate for Perl, Python and Shell.

```
/**
 * Parse and attempt to convert a string to a 64-bit unsigned integer,
 * applying the appropriate interpretation based on the base parameter.
 *
 * @param[in]  str     A pointer to a NULL-terminated C string representing
 *                     the integer to parse.
 * @param[out] output  A reference to storage for a 64-bit unsigned integer
 *                     to which the parsed value will be stored on success.
 * @param[in]  base    The base according to which the string should be
 *                     interpreted and parsed. If 0 or 16, the string may
 *                     be hexadecimal and prefixed with "0x". Otherwise, a 0
 *                     is implied as 10 unless a leading 0 is encountered in
 *                     which 8 is implied.
 *
 * @return true on success; otherwise, false on failure.
 */
```

Listing 27. Standard Doxygen-compatible free function or method comment for C, C++, Objective C, and Objective C++.

In addition, developers should well document the bodies of their functions and methods, describing the overall flow, design intent, error handling nuances, historical bugs encountered and resolved, and so forth. While these types of comments do not typically become part of the external documentation, they are invaluable to future maintainers of the code.

4.6.3. Other

4.6.3.1. Dos

- **Do** use the '@' Doxygen markup style rather than the '\' markup style.
- **Do** consider consulting the “Project CHIP Copy Style Guide” if you feel uncomfortable or unclear on your own writing style.
- **Do** also consider consulting tips on Plain Language for additional style and tone input.
- **Do** use consistent terminology and lingo.
- **Do** properly paragraph justify and wrap your documentation.
    - See your editor's documentation on how to do this (e.g. M-q in Emacs).

4.6.3.2. Don'ts

- **Do not** forget to document your files, enumerations, constants, classes, objects, namespaces, functions, and methods.
- **Do not** include the file name in any Doxygen file comments or directives.
    - Your editor knows the file name, source code control knows the file name, and you know the file name.
    - When it changes on the file system, having to change it in the file comments is simply an added burden.
- **Do not** include your name in any Doxygen comments or directives.
    - Source code control knows who you are and what file revisions you own.
    - We do not want our external partners knowing who you are and calling or e-mailing you directly for support.
- **Do not** include the modification date the file was last changed in Doxygen comments or directives, except for the copyright header.
    - Source code control knows when the file was last changed and the date other revisions were made.
- **Do not** include subjective or opinionated commentary or expose proprietary and confidential information not relevant to the code or APIs.
    - This content **will be** published to and for consumption by our third-party partners.

5. Third-party Software

No third-party software shall be used or included without prior legal, managerial, and technical authorization and approval.

5.1. Motivation and Rationale

Third-party software, both commercial and open-source can, in many cases, provide a tremendous amount of benefit and leverage. In fact, nearly, all Project CHIP products to date have a substantial amount of third-party software in their core platform.

However, third-party software comes with contractual, licensing, and technical liabilities that must be understood and vetted by appropriate legal, managerial, and technical leadership before being integrated with internal software.

A failure to understand those liabilities can expose Project CHIP to substantial risk including, but not limited to, the loss of intellectual property.

6. Testing

All code should be designed for test and co-developed with tests.

All code unit tests should be implemented using nlunit-test.

All tested code should be *coverable* with code coverage.

6.1. Motivation and Rationale

It is recognized and acknowledged that not every piece of code or subsystem, library, thread or process can be easily unit tested.

However, unit tests—and tests in general—are a *Good Thing*™; and will give you the confidence to make changes and prove to both yourself and other colleagues that you have things *right*, in so far as your tests cover *right*.

Unit tests will be combined with other methods of verification to fully cover the various test requirements in the system and for the module you are developing or supporting. The purpose of the unit test is to isolate your module's methods and ensure that they have the proper output, affect state properly, and handle errors appropriately. This ensures each of the building blocks of the system functions correctly. These tests can also be used for regression testing so that any code changes that affect a module's output will be automatically flagged.

Code coverage is important because without it, it is impossible to know how effective your tests are at exercising your code, ensuring as much of it is as tested as desired. Further, without code coverage, it is possible without knowing it to add test upon test to your module that do not actually improve the amount of code exercised and tested but rather just further wear a well-worn path through your code.

7. Documentation

All new and revisionary work should attempt to create the following documents or specifications:

- Functional Requirements
- Architecture and Design
- Protocol

7.1. Requirements, Architecture, and Design

7.1.1. Functional Requirements

- Specifies what it is we are building, who the stakeholders are, how it fits into a larger system, what the broad functional behaviors are, what the functional requirements are, and what the priorities of those requirements are.

7.1.2. Architecture and Design

- Specifies an architecture and design that meets the functional requirements.

- This covers enough detail to guide and influence an implementation but does not actually specify the implementation.

7.1.3. Protocol

- If applicable, this may be a subset of the architecture and design specification.
- This provides enough detail to completely specify packet formats, packet field definitions, protocol exchanges, schema definitions, data tags, status enumerations, and so forth.
- Basically, this should be sufficient for anyone to write a protocol implementation against or to develop a network analysis or debugging tool against.

In short, all of these documents—the architecture, design, and protocol in particular—should be sufficient to:

- Inform partners, influencers, managers, and new employees about how the protocol, its implementation, and its various components work.
- Enable support and tool vendors to develop and maintain network analysis and debugging tools.
- Enable integration and quality assurance personnel to develop and execute test suites.
- Enable technical writers to produce additional, externally-facing documentation in support of the above.

7.2. Stack and API

Equally as important as the requirements, architecture and design documents and specifications that lead up an actual implementation, is the documentation for the API implementation itself.

In support of this effort Project CHIP uses Doxygen to markup (or markdown) all C, C++, Objective C, Objective C++, Perl, Python, and Java code to:

- Detail what the various source and header files are and how they fit into the broader architecture and design.
- Detail what the various C++ / Objective C++ namespaces are.
- Detail what the constants, C preprocessor definitions, and enumerations are.
- Detail what the globals are and how they are to be used.
- Detail what the free function and object / class methods are and how they are to be used, what their parameters are, and what their return values are.
- Detail any other important technical information or theory of operation unique and relevant to the stack that is not otherwise captured in architecture, design, or protocol documentation.

7.2.1. Motivation and Rationale

As mentioned previously, Project CHIP's technology and infrastructure are business-critical infrastructure that undergirds all work group member products as well as those of their current and future product ecosystem partners. As a result, it is exceedingly important that it be documented as well as possible so that those inside and outside of Project CHIP can understand it, integrate it, support it, debug it, and develop against it.

8. Building and Packaging

Projects should either use standalone makefiles if they are simple or *nlbuild-autotools* if they are more complex, particularly when automatic support for unit testing and documentation is sought.

8.1. Motivation and Rationale

Having project- and technology-neutral build infrastructure is important because embedded communications software technologies are consumed by and used in a variety of internal projects as well as by external partners. In addition, they need to be used and testable in a standalone fashion.

9. Recommended Reading

While the following references and reading are not part of the formal best practices, coding conventions, and style cannon, they are informative and useful guides for improving the style and quality of the code you write:

1. Jet Propulsion Laboratory. *JPL Institutional Coding Standard for the C Programming Language.* Version 1.0. March 3, 2009.
2. Jet Propulsion Laboratory. *The Power of Ten – Rules for Developing Safety Critical Code*. December 2014.
3. Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs.* Third Edition. 2005.
4. Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* 1996.
5. Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems.* March 2013.
6. Motor Industry Software Reliability Association. *Guidelines for the Use of the C++ Language in Critical Systems*. June 2008.

Revision History

Revision	Date	Modified By	Description
1	2020-04-14	Grant Erickson	Initial revision.