EDK II C Coding Standards Specification

TABLE OF CONTENTS

EDK II C Coding Standards Specification
1 Introduction
- 1.1 Abstract
- 1.2 Rationale
- 1.3 Scope
- 1.4 References
- 1.5 Glossary

2 Guiding Principles
- 2.1 Software Design
- 2.2 Principles for Software Maintenance
- 2.3 Additional Recommendations

3 Quick Reference
- 3.1 Naming
- 3.2 Formatting
- 3.3 Files: General Rules
- 3.4 Documentation

4 Naming Conventions
- 4.1 General Naming Rules
- 4.2 File Names
- 4.3 Identifiers
- 4.4 Global & Module Variables
- 4.5 Name Space Rules

5 Source Files
- 5.1 General Rules
- 5.2 Spacing
- 5.3 Include Files
- 5.4 Code File Structure
- 5.5 Preprocessor Directives
- 5.6 Declarations and Types
- 5.7 C Programming

5.8 Error Handling and ASSERT
6 Documenting Software
6.1 Documentation Concepts
6.2 Comments
6.3 What NOT to Comment
6.4 What You Must Comment
6.5 Types of Comments
6.6 Introducing Doxygen
6.7 How Doxygen Works
6.8 Special Documentation Blocks
6.9 Putting Documentation after Members
6.10 Special Commands
APPENDIX A Common Examples
APPENDIX B Reserved Identifiers
APPENDIX C Optimization and Performance
Tables
Table 1 Common Opposites
Table 2 EFI Supported Abbreviations
Table 3 EFI Supported Acronyms
Table 4 Reserved Keywords
Table 5 Permissible Escape Sequences (ISO/IEC 9899:1990 6.1.3.4)
Table 6 EFI Data Types (slightly modified from UEFI 2.3.1)
Table 7 Modifiers for Common EFI Data Types (reference the UEFI Specification and Beyond Bios)
Table 8 EFI Constants
Table 9 Parameter Modifiers
Table 10 Predicate Expression Examples
Table 11 HTML Character Entities
Table 12 HTML Commands

EDK II C Coding Standards Specification

DRAFT FOR REVIEW

09/18/2019 10:20:55

Acknowledgements

Revision History

Revision	Revision History	Date

0.0.1	First swag.	6/23/00
0.0.2	Included feedback from team.	8/3/00
0.3	Add comments.	8/10/00
0.3001	Pre-vacation update, need to sync with new numbering process.	9/11/00
0.31	Incorporated Sync 1 learnings.	12/12/00
0.32	Completed TAT ARs.	6/8/01
0.33	Added goto rules.	8/16/01
0.34	Updated to match driver and Runtime Lib.	11/15/01
0.9	Updated to Intel(R) Platform Innovation Framework for EFI. Added checklist appendix.	1/8/04
0.91	Completed editing and formatting pass.	3/3/04
0.92	Updated the structure declaration rules: Added section 4.8 and modified the checklist in Appendix A.	4/8/04
0.93	Added some minor clarifications in section 3.1, 4.5, 7.1, and 10.	9/14/04
0.94	Revised to accommodate Doxygen style commenting standards	3/1/06
0.50	Change to new numbering scheme. Incorporate Review Comments. Editing and formatting.	4/21/06
0.51	Changed to EDK II.	7/13/06
0.52	Update rules to clarify areas of misinterpretation. Add copyright formatting rules.	2/09/2010
0.60	Re-organize document and update to current rules.	2/15/2010
0.70	Release for Review	3/1/2010
0.95	Review comments incorporated, Release to Tech Pubs for Finalization	3/10/2010
1.00	First full release	3/15/2010
1.01	Restructure into book format.	12/08/2011
1.02	Incorporate suggestions and trackers	3/19/2012
	Release For Review	4/2/2012
	Release	4/16/2012
1.03	Update and incorporate requests and bug fixes. Remove "Intel Confidential" classification.	9/11/2014
1.50	Release for Review	9/26/2014

1.80	Incorporate US Review Comments	10/10/2014
1.85	Incorporate PRC Review Comments	10/24/2014
	Release for Extended US & PRC Review	10/28/2014
2.0	Release	11/14/2014
2.1	DRAFT for REFORMAT	10/30/2015
2.2	Convert to Gitbook	June 2017
	#425 [CCS] clarify line breaking and indentation requirements for multi-line function calls	
	#1656 Update all Wiki pages for the BSD+Patent license change with SPDX identifiers	
	#607 Document code comment requirements for spurious variable assignments	

1 INTRODUCTION

1.1 Abstract

This specification establishes a set of rules to:

- Establish uniformity of style.
- Set minimum information content requirements.
- Allow all programmers to easily understand the code.
- Facilitate support tasks by establishing uniform conventions.
- Ease documentation efforts by embedding the design documentation in the code.
- Reduce customer and new employee learning curves by providing accurate code documentation and uniform style.

These rules apply to all code developed for inclusion in the EDK II code base, and are intended as an enabling philosophy. All changes or additions from this point on shall conform to this specification. Pre-existing code does not need to be updated for the sole purpose of conforming to this specification. As conforming updates are made, the developer may update other content within the modified file to bring it within compliance with this specification. Code originally developed for other environments that has been ported to, or modified for, the EDK II environment, is not obligated to conform. However, any new code added to the ported code must conform.

This specification addresses the chronic problem of providing accurate documentation of the code base by embedding the documentation within the code. While this does not guarantee that the documentation will be kept up to date, it significantly increases the chances. A document generation system, Doxygen, then produce formatted documentation by extracting information from specially formatted comment blocks and the syntactic elements of the code.

This specification presents protocol standards that will ensure that the contractual relationship between APIs and their callers is clear and well maintained.

This specification describes standard practices designed to eliminate or mitigate pitfalls inherent in the C language.

In recognition that a coding standard of this size can be a bit daunting, a concise reference to the standard's key elements is available in "Quick Reference".

1.2 Rationale

Software engineering is much more than writing code that will work one time. Software engineering entails writing code that:

1. Meets project requirements.
2. Is testable.
3. Can be maintained by the author or by others that have varying degrees of experience and familiarity with the code.
4. Minimizes the learning curve for programmers new to the product. (Our customers)

We use the C programming language because of its simplicity, flexibility, and wide support. On the downside in that each developer's C code could (conceivably) be constructed in totally different and inconsistent ways. This lack of uniformity makes understanding and maintaining the code very difficult.

Uniformity is the key theme of these rules. You may disagree with some of our decisions. Nevertheless, we ask that you commit to conforming to standards of this specification. Also, there are pitfalls inherent in the C language that this style guide may help you to avoid. The goal of this document is making you, and those who follow you, more productive.

Some of the strict rules for protocol and driver construction may seem overly onerous. Don't panic - there is a method to our madness - we intend to construct wizards to aid in the construction of protocol include files and device driver templates. The resulting consistency will help prevent name collision and require much less rote memory (or code surfing) to remember the names of protocol declarations or GUID definitions.

Good software engineers think about maintaining a consistent and uniform coding style. Having a set of rules to follow allows them to spend time solving actual problems and less time thinking about style. Junior software engineers may not see the benefits, but now is a good time for them to start thinking about what it will take to maintain their code. Junior software engineers will also benefit by being able to understand other people's code much more easily because the code will be written in a consistent manner. Consistency and uniformity enable productivity.

In conclusion, it's uniformity, uniformity, uniformity. With that said, this document is not intended to be dogma. However, violating a rule is contingent on valid reasons for the violation, and the approval of the various stakeholders involved. It is not something to venture into lightly and is not recommended.

1.3 Scope

This specification describes stylistic conventions and requirements as they apply to writing C code for inclusion within the EDK II code base. Its bulk is intended to provide rationale and disambiguating detail for each rule and requirement. The rules are also presented in summary form for quick reference. Rationale and detail for each of the rules is then presented in subsequent sections.

The majority of code produced for EDK II must support multiple compilers and be able to be retargeted to multiple processor and system architectures. This requires coding practices to conform to the "lowest common denominator" of the supported tools, processor, and system architectures.

The C dialect we use is ISO/IEC 9899:199409, or C95, with some elements from C99 If you are not familiar with this dialect, refer to Harbison and Steele's *C: A Reference Manual*. This is the language dialect held in common with all of the compilers supported by EDK II.

Note: There are some significant differences between ANSI C (C89) and the dialect recognized by compilers supported by EDK II (C95). These differences primarily revolve around use of the external storage class and the elimination of implicit types and storage classes.

The use of compiler-specific language extensions is very strongly discouraged.

Topics covered in this coding standard include:

- File structure and formats
- C language rules and guidelines
- Naming Conventions
- Documentation
- Commenting rules
- Doxygen

These guidelines represent an attempt make you aware of your actions, because those actions affect the future readers and maintainers of the code you produce.

Pre-existing code ported to the EDK II environment does not have to conform to this specification. New code which is added to, or which accompanies, the ported code must conform.

This specification considers the core firmware, applications, and individual UEFI drivers as distinct and separate units. This is because UEFI drivers, like applications, are not linked to the main body of code and do not expose their internal namespaces to other components of the firmware system.

1.4 References

- MISRA-C: *2004 Guidelines for the use of the C language in critical systems*, Tyler Doering http://www.misra-c.com, *Guidelines for the Use of the C Language in Critical Systems*, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.
- *Universal Principles of Design* by William Lidwell, Kritina Holden, and Jill Butler. ISBN 159253-007-9.
- *C: A Reference Manual* by Samuel P. Harbison III and Guy L. Steele Jr. ISBN 0-13-089592x.
- *Enough Rope to Shoot Yourself in the Foot* by Allen I. Holub. ISBN 0-07-029689-8.
- *Code Complete* by Steven C. McConnell. ISBN 1-55615-484-4.
- *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. ISBN 0-13110362-8.
- ISO/IEC 9899: 1990, Programming Languages - C. This specifies ANSI C.
- ISO/IEC 9899: 199409, Programming Languages - C. This specifies C95.
- ISO/IEC 9899: 1999; Cor-3, Programming Languages - C. This specifies C99.
- Writing Solid Code by Steve Maguire. ISBN 1-55615-551-4.
- EFI Application Toolkit Project Engineering Conventions. 10/4/1999.
- ISO/IEC 6592: 2000, Information Technology - Guidelines for the Documentation of Computer-based Application Systems.
- ISO/IEC 18019: 2004, Software and System Engineering - Guidelines for the Design and Preparation of User Documentation for Application Software.
- *Indian Hill C Style and Coding Standard* by L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington, Bell Labs; Henry Spencer, Zoology Computer Systems, University of Toronto; David Keppel, EECS, UC Berkeley CS&E, University of Washington; Mark Brader, SoftQuad Incorporated, Toronto. 6/25/1990.

- *Doxygen manual*, http://www.doxygen.org/manual.html, Version 1.4.6-NO
- *Doxygen Primer* by Daryl McDaniel, IBM 2002; Updated for Intel Corporation 1/2006
- *UEFI Specification*, http://www.uefi.org
- *Beyond Bios:Developing with the Unified Extensible Firmware Interface*, Second Edition, Zimmer, Michael Rothman, Suresh Marisetty Copyright @2010 Intel Corporation ISBN 13 978-1-934053-29-4

1.5 Glossary

ANSI

American National Standards Institute

C

The generic name for the C Programming Language. Originally finalized as an ANSI standard in 1989 ('C89') and updated in 1999 ('C99'). Subsequently adopted by the ISO (ISO/IEC 9899:1990), which has replaced the ANSI standard, even in the US.

CVS

Concurrent Versioning System

EFI

Extensible Firmware Interface

ISO

International Standards Organization

SVN

The Subversion revision management system.

Tab Stop

EDK II uses space characters instead of Horizontal Tab characters. If the left-most column is column 1, then every odd numbered column is a Tab Stop. Code elements are aligned at Tab Stops.

2 GUIDING PRINCIPLES

This chapter discusses the principles of high-quality software. It is likely that your design will not be able to achieve all of these goals; if it does, your design is very good indeed. Note that some of these design goals contradict other design goals. Contradiction is part of the challenge of doing a good design.

Maintainability of software is of such importance that those principles are expanded separately in "Principles for Software Maintenance".

2.1 Software Design

The following is an alphabetical list of software design principles:

Accessibility

This entails designing objects and environments to be usable, with no modification, by the greatest number of people as possible, including people with varying educational and social backgrounds, as well as those with motor or sensory challenges.

Alignment

Elements within a design should be aligned with one or more other elements. Alignment of related or like elements within a design reduces the perception of disorder and promotes understanding.

Chunking.

Chunking groups units of information into a small number of units (maximum of four plus or minus one) to help the efficient processing of information by shortterm memory, as well as to accommodate its limits.

Confirmation.

This is a technique used for critical actions, inputs, or commands. Confirmations are primarily used to prevent unintended actions. Minimize errors in critical or irreversible operations with confirmations. If you overuse confirmations, expect that they will be ignored Avoid overusing confirmations to ensure that they remain unexpected and uncommon; otherwise, they may be ignored. Use a two-step operation for hardware confirmations and dialogs for software confirmations.

Consistency.

Express similar parts or concepts in similar ways to make a system to improve usability and learnability. Apply consistency to design and coding style, as well as user interfaces. Do not apply consistency to the point of compromising clarity or usability.

Maintainability

Design for the maintenance programmer or sustaining engineer for maintainability, or ease of maintenance. The design of the system should be self-explanatory.

Extensibility

Extensibility entails enhancing a system without violating the underlying structure. The most likely changes should cause the system the least trauma. For example, you know the BIOS is responsible for booting the system, so adding a new type of boot device should not cause trauma to the system. Be careful about the assumptions you make.

Forgiveness

Design to help users avoid errors and reduce the negative consequences of errors any errors made. Recommended methods for achieving design forgiveness include affordances, reversibility of actions, and safety nets. Effectively designing for forgiveness results in a design needing minimal confirmations, warnings, and help.

High fan-in

Sharing a high number of routines that call a given routine produces high fan-in. Sharing entails designing a system to make good use of utility routines at the lower levels.

Horror Vacui

This is a Latin phrase for "fear of emptiness", which is the desire to fill empty spaces with information or objects. Research shows that as *horror vacui* increases, perceived value decreases. In programming, lines consisting solely of comment characters, with no actual comment, are good examples of *horror vacui.*

Intellectual manageability

Intellectual manageability is a primary goal in any system. It is essential to the overall system integrity and affects how easily programmers can initially build a system as well as maintain it later.

Interference Effects

When two or more perceptual (or cognitive) processes are in conflict, the competing mental processes slow down mental processing or make mental processing less accurate. Examples of this include violations of convention (a red **OK** light), information conflicts (a color name, **GREEN**, in a different color), and incorrect use of opposites.

Leanness

Design the system so that it has no extra parts, i.e. "lean". If you add extra code, remember that it needs to be developed, reviewed, tested, maintained, understood, and taken into account when the code is modified. Also, future versions of the code may have to be backward compatible with the extra code.

Low complexity

Low complexity is part of intellectual manageability.

Minimal connectedness

Design so that you keep connections among subprograms to a minimum. This minimizes work during integration, testing, and maintenance. Use industry standards whenever possible. Make sure you are not reinventing something that already exists.

Minimize code size

Unlike many software engineering projects, EDK II firmware is targeted to be stored in a device that represents a tangible cost to the system. Thus, the minimization of code size is important to reduce the overall cost of a system.

Portability

Design the system so that it is easily moved to another environment. With EDK II, making sure the code will run on IA32, X64 and Intel(R) Itanium(R) processors (IPF) is an example of portability.

Reusability

Design the system so that pieces of it can be reused in other systems. In EDK II, reusability also means designing code so that it can be used in various classes of platforms from embedded systems to massively parallel computers.

Standard techniques

Greater reliance on unique or exotic pieces makes a system harder to understand, and more intimidating for someone trying to understand it the first time. Using standardized, common approaches should be to give the whole system a familiar feeling. This standardization is one of the primary goals of this document.

Stratified design

In order to view the system at any single level and get a consistent view of it, you should attempt to keep the levels of decomposition layered. The OSI multi-layernetworking model is an example of stratified design.

2.2 Principles for Software Maintenance

2.2.1 Understand the problem before you fix it.

The best way to ruin a code base is to attempt to fix problems without a clear understanding of the problem itself. If your fix is out of context, the next fix in the routine will be four times more complicated. Triangulate the error with cases that should and should not cause the error. Keep at it until you understand the error.

2.2.2 Understand the program, not just the problem.

Understanding the context in which a problem occurs will increase your likelihood of solving the problem completely.

2.2.3 Fix the symptom AND the underlying problem.

Fix the symptom, but your focus should be on fixing the underlying problem. Without thoroughly understanding the problem, you will not fix the code. You will also feel very uncomfortable when a peer reviews your fix before you check it into the source base.

2.2.4 Change the code only for good reason.

Do not change code at random until it seems to work; that isn't effective. If you do this you are not learning anything and are just goofing around. You should have confidence that a change will work before making a change. Being wrong about a change should be rare and cause personal reevaluation.

2.2.5 Do not debug by superstition.

Don't blame every problem on the computer, bad data, or the effects of a full moon. You wrote the program; take responsibility for it.

2.2.6 Don't blame everyone else's code.

It is human nature to trust the code that you wrote and understand, and to distrust all other code. You must resist this tendency and root cause the problem systematically.

2.2.7 Don't use source control to debug problems.

You don't debug by randomly trying previous versions of code. You can use previous versions as a single test of your triangulated test cases. Bugs are not always introduced by changes. Bugs can lie dormant in a code base for long periods of time. By blindly rolling back changes, you could just be hiding a bug, versus fixing one.

2.2.8 Check your fix.

Run the triangulated test cases against your code. Have another set of eyes look at your change, preferably someone who is experienced in the code.

2.2.9 Look for similar errors.

Errors tend to occur in groups, so check to make sure a similar mistake was not made in other parts of the code.

2.2.10 Fix the code AND the comments.

When you have correct code but incorrect comments, you will confuse the next person in the code. Remember, programming is a team sport.

2.2.11 Fix the comments AND the documentation.

The next person to make a large change to code will thank you and might even not camp out in your cube for a week. This has been simplified in EDK II by embedding the documentation within the comments.

2.3 Additional Recommendations

- If you have trouble debugging without violating these rules, please ask for help.
- Learn to debug by debugging with someone who is good at it. There is (approximately) a 20-to-1 difference in the time it takes an experienced programmer to find the same set of errors as an inexperienced programmer.
- Try to not focus only on the bug, but on the process and techniques the experienced programmer uses to find bugs.
- When working with a more experienced debugger, your goal should be to improve your debugging skills.
- If you rely too heavily on the debugging skills of others, your own expertise suffers.

3 QUICK REFERENCE

3.1 Naming

- Names must clearly represent the purpose of the named object. See "General Naming Rules".
- Do not use names that are very similar to existing concepts, such as 'event'. See "General Naming Rules".
- Do not use the names of symbols declared in standard header files as internal symbols. See "Function and Data Names".
- Overloading function or type names is not allowed. "Name Space Rules".
- Use the correct opposites when naming. See "Common Opposites in Variable Names".
- Use standard abbreviations only. See "Abbreviation Usage".
- Use industry standard acronyms only. See "Acronym Usage".
- Any nonstandard abbreviation or acronym must be defined in the file header of any file using the abbreviation or acronym. See "Abbreviation Usage" and "Glossary".
- There is no limit to name lengths. A length of 10 to 30 characters is ", recommended. See "File Names" & "Identifiers that are always reserved".
- File names must not start with numbers. See "File Names".
- Each include file name must be unique. See "Include Files".
- File, function, variable, enumeration, and data structure elements must have names like the following: `EachWordIsDistinctEvenAcronymsLikeAcpi`. See "Identifiers".
- Don't capitalize all letters in acronyms. `MyPCIAddress` is hard to read, compared to `MyPciAddress`, especially if acronyms are mixed with numbers, like `My870PCIBus0BDF`. See "Acronym Usage".
- Acronyms in comments and documentation shall follow English rules and be capitalized. See "Acronym Usage".
- Functional macros, `#defines`, and `typedefs` must have names like: `EACH_WORD_IS_DISTINCT_EVEN_ACRONYMS_LIKE_ACPI` See "Type and Macro Names".
- Hungarian naming is not allowed. See "Hungarian Prefixes".

- Global data names must be prefaced with a ' `g` '. Example: `gMyGuid` . See "Global & Module Variables".
- Module global data names must be prefaced with an ' `m` '. Example: `mMyGuid` . See "Global & Module Variables".

3.2 Formatting

3.2.1 Formatting: General Rules

- Tab characters are not allowed. See "General Rules".
- All indentation (tabs) is two spaces. See "General Rules".

3.2.2 Formatting: Vertical spacing

- Use blank lines and comments to group blocks of related code. See "Vertical Spacing".
- Never put more than one statement per line. See "Vertical Spacing".
- Never put the code and conditional on one line. See "Vertical Spacing
- Never put more than one declaration per line. `UINT8 MyData1, MyData2;` is illegal. See "Vertical Spacing".
- Always put open braces ' `{` ' on their own line for functions or multi-line predicate expressions. All other uses put the brace following the conditional. See "Vertical Spacing".
- Always put close braces ' `}` ' on their own line, indented to match the first line of the construct. Exceptions are `else` , `else if` , and `do-while` code. See "Vertical Spacing".

3.2.3 Formatting: Horizontal spacing

- Always put space before and after binary operators. See "Horizontal Spacing".
- Never put space between unary operators and the operand. See "Horizontal Spacing".
- Always put space after ' `,` ', or ' `;` ' if more code follows. See "Horizontal Spacing".
- Always put space before a ' `(` ' except for ' `((` '. See "Horizontal Spacing".
- Always put space before a ' `{` ' if it is not on its own line. See "Horizontal Spacing".
- Never put spaces around ' `.` ' or ' `->` ' operators. See "Horizontal Spacing".

- Never put a space between array operands and '`[`'. See "Horizontal Spacing".
- Always Line up continued lines with the element being continued. See "Horizontal Spacing".

3.2.4 Formatting: Predicate Expressions

- Always use parentheses rather than relying on "Horizontal Spacing" (above) operator precedence. "Predicate Expressions".
- Booleans do not need to be compared with `TRUE` or `FALSE`. See "Predicate Expressions".
- Pointers must be explicitly compared to `NULL`. See "Predicate Expressions".
- Numbers must be explicitly compared to another number. See "Predicate Expressions".

3.3 Files: General Rules

- Do not use tabs, only use Spaces. "General Rules".
- Unless explicitly stated otherwise, spaces (white space) may be one or more space characters long. See "General Rules".
- Files may only contain the ASCII characters 0x0A, 0x0D, and 0x20 through 0x7E, inclusive. See "General Rules".
- Do not produce lines that exceed 120 columns in your source files. See "General Rules".
- New files shall not use `#pragma` except for `#pragma pack(#)`. See "General Rules".
  - Ported files may retain pre-existing `#pragma`s. See "General Rules".
  - Ported files may contain `#pragma`s to disable prevalent warning messages.
- All lines must end with CRLF (Carriage Return Line Feed); 0x0D followed by 0x0A.
- All files must end with CRLF. See "General Rules".
- Every new file must begin with a "File Heading" comment block. See "File Heading"

3.3.1 Files: Horizontal Spacing

- Always put space before and after binary operators. See "Horizontal Spacing".

- Do not put space between unary operators and their object. See "Horizontal Spacing".
- Horizontal spacing for multi-line function calls should line up one or two tab stops after the beginning of the function name. See "Horizontal Spacing".
- Always put space after commas or semicolons that separate items. See "Horizontal Spacing".
- Always put space before an open parenthesis, except for macro definitions. See "Horizontal Spacing".
- Put space before an open brace if it is not on its own line. See "Horizontal Spacing".
- Do not put space around the structure member, ' `.` ', and pointer, ' `->` ', operators. See "Horizontal Spacing".
- Do not put space before the open brackets, ' `[` ' of array subscripts. See "Horizontal Spacing".
- Align a continuation line with the part of the line that it continues. See "Horizontal Spacing".
- Use parentheses instead of relying upon knowledge of C precedence ordering. See "Horizontal Spacing".

3.3.2 Include Files

- Every header file must have a ' `#ifndef FILE_NAME_H_` ' and ' `#endif` ' guard surrounding all code. See "Include Files".
  - The `#ifndef` must be the first line of code following the file header comment.
  - The `#endif` must appear alone on the last line in the file.
- All C include files shall use the same extension and it shall be `.h` . See "Include Files".
- Include statements shall not contain absolute paths or paths that contain '..'. See "#include"
- Functional macros are discouraged except for includes, debug, CR, and linked lists. See "Macros".
- Macros should be defined with the maximum use of parentheses to remove any possible ambiguity. See "Macros".
- Include files must contain either public or private data, not both. See "Include Files".
- Include files must not contain code generating statements. See "Include Files".

- Every parameter must have the proper `IN`, `OUT`, `OPTIONAL`, etc., modifiers. See "Function Definition Layout".
- Only use UEFI data types. Use of standard C data types is prohibited. See "Common Data Types".

3.3.3 Code Files

- Only use UEFI data types. Use of standard C data types is prohibited. See "Common Data Types".
- Code files should not contain `#define` and `typedef` statements.
- Do not use inline assembler in the source files. See "General Rules".
- Function definitions, as well as `if`, `for`, `while`, and `switch` statements, must follow strict rules. See "Function Definition Layout", "Flow Control Statements, and "Introducing Doxygen".
- Enumerated types must end with a maximum element. See "Enumerated Types".
- Enumerated types should begin with a minimum element.
- Structures are always defined with a `typedef` format. See "Structure Definitions".
- The open and closing braces of a function definition are in column one and on their own lines. See "Function Definition Layout".

3.3.4 Code Files: Vertical Spacing

- There shall be only one statement per line. See "Vertical Spacing".
- Open braces, '`{`', shall be on the same line as the closing parenthesis, '`)`', of one-line predicate expressions. See "Vertical Spacing".
- Open braces, '`{`', shall be on a line by themselves and aligned with the first character of the associated flow control statement when following a multi-line predicate expression. See "Vertical Spacing".
- Close braces, '`}`', always go at the beginning of the last line of the body. See "Vertical Spacing".
- A close brace may share a line with the `else {`, `else if () {`, and do-while constructs. See "Vertical Spacing".
- Each sub-expression of a complex predicate expression must be on a separate line. See "Vertical Spacing".

3.4 Documentation

3.4.1 Documentation: Commenting

- Comments must explain why the code does what it does. See "Comments".
- Every file must have a properly formatted file header. See "File Heading".
- Every function and functional macro must have a correct function header in both the source and include files. See "Macros" & "Function Headings".

3.4.1.1 Documentation: Internal comments

- Local comments must use the C++ comment style, ' `//` '. See "Internal Comments".
- Local comments must have a blank line before the comment block. See "Internal Comments".
- Comments must be indented to match the code. See "Internal Comments".
- If a comment applies to more than one block of code, there should be a blank line after the comment. See "Internal Comments".
- If a comment applies to a single block of code, there should **not** be a blank line separating the comment from the code. See "Internal Comments".

3.4.1.2 Documentation: What not to comment

- **No** comment markers are allowed in code, including:
  - `BUGBUG`
  - Your name
  - Your initials
  - Special markers, such as `FIX_THIS` , `TEST` . See "What NOT to Comment".
- Use your bug tracking system to track bugs instead of markers within the code. If you really must mark the code, use Doxygen's `@bug` or `@todo` commands. See "What NOT to Comment".

3.4.2 Doxygen

- Doxygen comments are used to document global and file-scope elements. See "Global Comments".
- C-style comment blocks are of the form:

```
/** Brief Description.
 *  ... More text ...
*/
```

- C++ style comment blocks begin with `///`.
- Comments precede the semantic element they document. See "Special Documentation Blocks".
- The special form, `///< ...`, allows documentation to be after the documented element. See"Putting Documentation after Members".
- Comment blocks automatically start with a brief description and end at the first period. See "Special Documentation Blocks" .
- The most frequently used Doxygen commands are: (See "Special Commands").

```
@file [<name>]
@param[in, out] <parameter name> { parameter description }
@retval <return value> { description }
@sa { references }
@test { description of a test case }
```

4 NAMING CONVENTIONS

4.1 General Naming Rules

Use good naming practice when declaring variable names.

Studies show that programs with names averaging 10 to 16 characters are the easiest to debug. The name length is just a guideline*; the most important thing is that the name conveys a clear meaning of what it represents*.

Do not overload commonly used terms.

For example, EFI has an event model, so don't call some abstraction that you define an Event. People will get confused. Make sure someone reading the code can tell what you are talking about.

Each word or concept must start with a capital letter and be followed by lower-case letters.

The intent is for names to be consistent and easily readable. Each word in a compound name should be visually distinct.

4.1.1 Identifiers that are always reserved

Identifiers beginning with an underscore are always reserved

Define them only in the special ways allowed elsewhere in this document.

Identifiers that are defined in the Standard C and POSIX libraries are always reserved.

This includes macros, typedefs, variables, and functions, whether with external linkage or file scope. The only exception is with modules that are mutually exclusive with these libraries. These reserved identifiers are listed in "Reserved Identifiers" and reserved keywords are listed in "Reserved Keywords".

4.1.2 Common Opposites in Variable Names

Use the correct opposites when declaring names.

Table 1 Common Opposites

add / remove	begin / end	create / destroy
increment / decrement	first / last	get / release
lock / unlock	put / get	up / down
old / new	min / max	next / previous
source / destination	open / close	show / hide
	source / target	start / stop

4.1.3 Abbreviation Usage

4.1.3.1 The use of abbreviations shall be regulated.

This document describes a common set of abbreviations that can be freely used. If you must make up abbreviations, remember the name is most important to the reader of the code, not the writer.

4.1.3.2 New abbreviations must be documented in the header of each using file.

Any abbreviation used, which is not documented in this specification, or in the *UEFI Specification* shall be placed into a Glossary section of the File header as specified in See "File Heading".

Do not define a new abbreviation to replace an abbreviation that is already defined in this document. For example, do not define No to mean Number, because Num is the supported abbreviation.

"EFI Supported Abbreviations" below lists the abbreviations that are standardized by this document and do not require a defining comment.

Table 2 EFI Supported Abbreviations

Abbreviation	Description
Ptr	Pointer
Str	Unicode string
Ascii	ASCII string
Len	Length
Arg	Argument
Max	Maximum
Min	Minimum
Char	Character
Num	Number
Temp	Temporary
Src	Source
Dst	Destination
BS	EFI Boot Services Table
RT	EFI Runtime Table
ST	EFI System Table
Tpl	EFI Task Priority Level

4.1.3.3 Powers of 2 and 10

You are encouraged to use the IEC international abbreviations for powers of 2 (KiB for 2^10, MiB for 2^20, GiB for 2^30, etc.) rather than the old KB, MB, and GB, which IEC now reserves for powers of 10 (10^3, 10^6, 10^9). Given that many readers of the code may not have made the conversion to add the 'i', do not use KB, MB, and GB for powers of 10 Instead, use e.g. "2*10^6 bytes" instead of 2MB to avoid confusion. Note that GiB is derived from the G in 'Giga', the 'i' in binary, and the B in 'Byte'.

4.1.4 Acronym Usage

4.1.4.1 The use of acronyms shall be limited.

Please remember the golden rule: Code for the person who will have to read and maintain your code. Making up your own vocabulary to describe your module can lead to lots of confusion.

4.1.4.2 Created Acronyms must be fully defined.

If you must create acronyms, they must be fully defined in the documentation

4.1.4.2.1 Translation tables are required for each module using a created acronym

Each module that uses the acronym must contain a translation table comment in the file header. This definition is required so that others can understand your names and comments.

4.1.4.3 Industry-Standard Acronyms are allowed

It's okay to use acronyms for industry standards.

Acronyms such as Pci, Acpi, Smbios, Isa, (capitalized per the variable naming convention) are all legal to use without defining their meaning.

If you reference an industry standard acronym, the file header must define to which version of the specification the code is written. Thus, a PCI resource manager would state that it was coded to follow the PCI 2.2 Specification and which optional features it included support for.

4.1.4.4 Capitalize Acronyms in comments and documentation to match their industry standard use.

For example, use "PCI" in comments and documentation, and "Pci" for functions, files, etc.

The table below lists the acronyms that are considered integral to the EDK II vernacular, and may be used without defining their meaning in a comment.

Table 3 EFI Supported Acronyms

Acronyms	In an Identifier	Description
ACPI	Acpi	Advanced Configuration and Power Interface
AGP	Agp	Accelerated Graphics Port
ANSI	Ansi	American National Standards Institute
ASCII	Ascii	American Standard Code for Information Interchange
ATA	Ata	Advanced Technology Attachment
ATAPI	Atapi	Advanced Technology Attachment Packet Interface
BFD	Bfd	Boot Flash Device
BIOS	Bios	Basic Input/Output System
BIS	Bis	Boot Integrity Services

CMOS	Cmos	Complementary metal oxide semiconductor
CPU	Cpu	Central processing unit
CRC	Crc	Cyclic Redundancy Check
DMA	Dma	Direct Memory Access
DXE	Dxe	Driver Execution Environment
EFI	Efi	Extensible Firmware Interface
FD	Fd	Flash Device
FIFO	Fifo	First In First Out
FV	Fv	Firmware Volume
GUID	Guid	Globally Unique Identifier
IEC	Iec	International Electrotechnical Commission
ISA	Isa	Industry Standard Architecture
ISO	Iso	International Standards Organization
NVRAM	Nvram	Nonvolatile Random Access Memory
PCI	Pci	Peripheral Component Interconnect
PEI	Pei	Pre-EFI Initialization environment
RAM	Ram	Random Access Memory
ROM	Rom	Read-Only Memory
SRAM	Sram	Static Random Access Memory
TPL	Tpl	Task Priority Level
UEFI	Uefi	Unified Extensible Firmware Interface
UNDI	Undi	Universal Network Driver Interface
USB	Usb	Universal Serial Bus
VGA	Vga	Video graphics array

4.2 File Names

4.2.1 There is no limit to file name lengths.

Do not assume that file names must be 8.3 compatible. Be reasonable though. Let the file names be as long as necessary, but no longer. Some operating systems limit file names to 32 characters.

4.2.2 Spaces in file and directory names are NOT permitted.

Allowing spaces would cause problems with certain versions of existing industry tools and does not provide additional clarity.

4.2.3 Never start file names with numbers.

Most source control systems will not be able to handle file names that start with numbers.

4.2.4 Non-standard characters shall not occur in file names.

All file names within an EDK II source tree must comply with the following regular expression:

```
[A-Za-z][_A-Za-z0-9-]*[A-Za-z0-9]+
```

That is, a letter followed by zero, or more, letters, underscores, dashes, or digits followed by a period followed by one or more letters or digits.

4.3 Identifiers

4.3.1 Identifiers shall not rely on the significance of more than 31 characters.

Identifiers (variable names, labels, structure tags, derived macro names, etc.) may be an arbitrary length. The ISO standard only guarantees that language processors only pay attention to the first 31 symbols when comparing identifiers. Note that the same ISO standard requires that external labels (those visible to the linker) be unique in their first six characters. Since it has been confirmed that 31 character / case significance is supported by EDK II supported tool chains, there is no requirement to ensure uniqueness of externals within the first 6 characters.

4.3.2 Always make identifier names that are visually distinguishable.

While not as big an issue as it has been in the past, when choosing labels ensure that the label is unlikely to be confused with other labels used in the file. Ensure that long label names vary by more than one or two characters. Ensure that labels don't vary between zero and oh (0 and O), one and ell (1 and l). Some also consider 2 and Z, and 5 and S to be similar.

4.3.3 Hungarian Prefixes

4.3.3.1 Use of Hungarian notation is not allowed

This set of detailed guidelines for naming variables and routines is a convention widely used with the C programming language, especially in Microsoft *Windows* programming. An example of a non-compliant variable named with the Hungarian conventions follows:

```
bmRequestType;  // Byte mask, First byte in the USB message header
pachInsert;     // A pointer to an array of characters to insert.
```

Global data and module data shall be prefixed with 'g' or 'm', respectively. Pointer variables may optionally be prefixed with 'p'. These are the only exceptions to the prohibition against Hungarian notation.

4.3.3.2 Any variable with file scope, or better, shall be prefixed by an 'm' or 'g'

There are no exceptions to this rule. The '`m`' prefix identifies a variable with module scope, while a '`g`' prefix identifies a global variable.

```
gThisIsAGlobalVariableName
mThisIsAModuleVariableName
```

4.3.3.3 Pointer variables may optionally be prefixed with a 'p'

Time has shown that pass-by-value vs. pass-by-reference errors are significantly reduced with only the introduction of a '`p`' prefix for pointer variables.

4.3.3.4 Reasons use of Hungarian prefixes not allowed

The abstraction of abstract data types is ignored. Instead, base types based on programminglanguage integers or long integers are abstracted. Thus, the names are focused on data types instead of the object-oriented abstraction that they represent. This focus is of little value and forces manual type checking that can be accomplished easily by the compiler with warnings promoted to errors.

Hungarian notation combines data meaning with data representation. If you change a data type you have to rename the variable. There is no mechanism to ensure that the names are accurate.

Studies have shown that Hungarian notation tends to encourage lazy variable names. It's common to focus on the Hungarian prefix without putting effort into a descriptive name.

4.3.4 Function and Data Names

4.3.4.1 Identifiers shall contain mixed upper- and lower-case text.

Use of all upper- or all lower-case is very difficult to read because compound words cannot be clearly separated.

4.3.4.2 The names of newly created global entities (such as structures, macros, and defines) shall not use an `EFI_` prefix.

From now on, the use of `DXE_` and `PEI_` prefixes shall be reserved for DXE and PEI drivers, respectively. If a structure happens to apply equally to PEI and DXE, it should use the prefix `DXE_`. If a structure is local to a particular module only, no special prefix is required.

4.3.4.3 Acronyms are not capitalized in Function and Data Names.

When all letters in an acronym are capitalized, it makes the prior and subsequent words visually difficult to distinguish.

```
ThisIsAnExampleOfWhatToDoForPci
```

4.3.4.4 Never use C keywords or the names of symbols declared in the standard header files as internal symbols.

When you need to use the name of an existing library function for a user-defined function, each use of the user-defined function must be paired with a corresponding comment. The ISO standard does not, however, guarantee that the user-defined function will take priority over the library function.

4.3.4.4.1 List of the C-reserved keywords.

In principle, the ISO standard, reserves all names beginning with underscore + capital letter, or with underscore + underscore. External symbols names shall not begin with an underscore.

Table 4 Reserved Keywords

auto	break	case	char	const
continue	default	do	double[a]	else
enum	extern	float[a]	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while	inline	restrict	wchar_t
bool	true	false	NULL	_Bool[b]
_Complex[b]	_Imaginary[b]	and[c]	and_eq[c]	bitand[c]
bitor[c]	compl[c]	not[c]	not_eq[c]	or[c]
or_eq[c]	xor[c]	xor_eq[c]		

[a]. Floating point operations are not recommended in UEFI firmware. ↩

[b]. These keywords are specific to C99 and are not reserved in C++. ↩

[c]. Macros defined in header iso646.h. These identifiers are reserved in C++ and are defined as part of the C Standard Library. ↩

In addition to those listed, the identifiers asm and fortran are common language extensions and should also be treated as reserved.

4.3.5 Type and Macro Names

4.3.5.1 Use all capital letters for both #define and typedef declarations.

This clearly differentiates static declarations from dynamic data types.

4.3.5.2 Each word of a concept shall be separated by an underscore character.

The underscore effectively separates the words, making names more readable.

4.3.5.3 The use of the "_t" suffix, designating a type, is not allowed.

```
typedef UINT32 THIS_IS_AN_EXAMPLE_OF_WHAT_TO_DO_FOR_PCI;
```

4.3.5.4 The names of guard macros shall end with an underscore character.

The guard macro, used in the `#ifndef` at the start of an include file, uses a postfix underscore character '`_`', in its name in order to prevent collision with other names that follow the naming convention. This may not be sufficient for header files that don't have a unique name. In that case, additional text may have to be added to the macro name in order to make it unique. This may not be required if the header files are mutually exclusive.

```
#ifndef FILE_NAME_H_
#define FILE_NAME_H_
...
#if (A_NUMBER > 72)
...
#else // NOT (A_NUMBER > 72)
...
#endif // (A_NUMBER > 72)
...
#endif /* FILE_NAME_H_ */
```

4.3.5.5 The #else and #endif clauses of conditional compilation blocks shall be commented to identify their context.

If a conditional compilation construct spans more than seven lines, a comment shall be added to the construct's `#else` and `#endif` clauses identifying the block the clause is associated with. This is illustrated in the preceding code example. The comment shall be on the same line as the `#else` or `#endif` clause.

4.4 Global & Module Variables

There is often confusion about what constitutes module variables versus global variables. Technically, both global and module variables are defined at file scope with external linkage. A module variable is intended to only be accessed across a small set of related routines that have strict rules for accessing the data; in effect, constrained to the set of files described within a single .inf file. A global variable is intended to be accessed throughout the firmware, usually indirectly through a protocol pointer or similar mechanism.

This is important when the time comes to maintain a module. A module variable should be fairly safe and easy to change because it is only accessed from a small number of routines. On the other hand, a global variable is accessed throughout the firmware and as the firmware evolves more code will tend to access the data resulting in a large number of uses to track down.

4.4.1 Recommendations for Global and Module Variables

4.4.1.1 The use of global and module data is strongly discouraged.

Global variables are appropriate for GUID, protocol, PPI definitions and other immutable objects. Attempting to create global variables can cause many problems, including: increased image size and variables actually residing in ROM.

The use of global and module variables may be appropriate for solving certain programming issues. A module is defined to be a set of data and routines that act on that data. Thus, in EFI a protocol could be thought of as a module. A complicated protocol may be built out of several smaller modules.

4.4.1.2 Use locking to protect access to global and module variables.

This protection is strongly encouraged and especially useful for data that is accessed at various task priority levels.

4.5 Name Space Rules

ISO C defines several name spaces (see ISO/IEC 9899:1994 6.1.2.3). The same name could be used in a separate name space for a completely different item.

Name spaces are defined as:

- label names
- tags of structures, unions and enumerators
- Members of structures or unions
- All other identifiers.

Note: Name space and scope are not synonymous. Name space rules do not apply to scope. Scope is described in "Scoping Rules".

4.5.1 Names shall be used consistently within the same type.

For example, structure tags may only be reused as structure types, and union tags may be reused only for union types.

```
typedef struct MyStruct {
  int  one;
  int  two;
  int  three;
} MY_STRUCT;
```

Because of the similarity of `MyStruct` to `MY_STRUCT`, they may only be used to refer to the same structure type.

4.5.2 No identifier in one name space may be reused as an identifier in another name space

Exceptions are structure member and union member names.

```
typedef struct StructOne {
  INT32             one;
  INT16             two;
  struct StructOne  *MySelf;
} STRUCT_ONE;

typedef struct StructTwo {
  INT16             one;
  INT8              *two;
  struct StructTwo  *MySelf;
} STRUCT_TWO;

typedef struct {
  STRUCT_ONE  *StructOne;     // NOT ALLOWED
  STRUCT_TWO  *StructTwoPtr;  // ALLOWED, it is unique
} BAD_STRUCT;
```

4.5.3 A typedef name shall be a unique identifier.

The name that appears at the end of a typedef (`STRUCT_ONE` and `STRUCT_TWO` in the example in Section 4.5.2) is known as a *typedef name*. Because of ambiguity in the C specifications, and to avoid confusion, and once a typedef name is used in a structure declaration, it may not be declared elsewhere

---

Note: Including the declaration in a header file that is then included in a number of files is not a violation of this rule.

4.5.4 A tag name shall be unique.

The name after the `struct` in structure definitions (`StructOne` and `StructTwo` in the example in 4.5.2) is known as a *structure tag* or simply a *tag*. To avoid confusion, once a tag is used for declaring a structure it shall not be declared elsewhere.

Note: Including a header file that contains a structure definition is not a violation of this rule.

4.5.5 Prefix module-scope identifiers for cleaner namespaces.

The use of prefixes is not an absolute requirement, but has been shown as a successful method of avoiding namespace pollution and makes it easier to meet other naming requirements. A useful prefix is the module's name. For example, the UEFI Shell uses the prefix "Shell" for its identifiers.

```
SHELL_FREE_NON_NULL (Buffer);
ShellCommandLibConstructor (ImageHandle, SystemTable);
```

5 SOURCE FILES

5.1 General Rules

5.1.1 Lines shall be 120 columns, or less

Preferably, limit line lengths to 80 columns or less. When this doesn't leave sufficient space for a good postfix style comment, extend the line to a total of 120 columns. Having some level of uniformity in the expected width of the source is useful for viewing and printing the code.

5.1.2 Do not use tab characters

Tabs shall be set in all editors to expand to two spaces. All indentation is on two space boundaries. There are no exceptions to this rule! This rule makes code look the same in all editors.

5.1.3 Files may only contain the ASCII characters 0x0A, 0x0D, and 0x20 through 0x7E

Files should be saved using either ASCII or UTF8 encoding.

5.1.4 Only escape sequences defined in the ISO C standard shall be used in string literals.

Implementing hexadecimal escape sequences, '\x20' for example, is prohibited because they vary from compiler to compiler.

The ISO standard defines octal escape sequences: '\102', for example. Because of the similarity to decimal values, and octal having fallen into disuse, use of this construct is prohibited. The only exception to this rule is '\0'.

Other than '\0', the only permissible escape sequences are:

Table 5 Permissible Escape Sequences (ISO/IEC 9899:1990 6.1.3.4)

`\a`	Alert: Visual or audible alert
`\b`	Backspace: Move to previous position
`\f`	Form Feed: Move to next logical page
`\n`	New Line: Move to start of next line
`\r`	Carriage Return: Move to start of line
`\t`	Horiz. Tab: Move to next tab position
`\v`	Vert. Tab: Move to next Vert. Tab pos.
`\'`	Single Quote
`\"`	Double Quote
`\0`	NUL character
`\\`	Single Backslash

5.1.5 Octal constants (Base 8) shall not be used.

The C language specification has defined numbers whose first digit is zero as octal, so 010 is decimal 8 The use of octal has declined considerably since C was first defined but this construct remains for backwards compatibility. Its use is prohibited. In particular, do not be tempted to use the zero prefix in tables of numbers to ensure visual alignment:

```
Address[0] = 00130;  // Decimal 88
Address[1] = 01450;  // Decimal 808
Address[3] = 81200;  // Decimal 81200
```

5.1.6 Only use CRLF (Carriage Return Line Feed) line endings.

Use Windows, or MSDOS, style line endings. Do NOT use Unix / Linux / MacOS / OS-X file formats.

5.1.7 All files must end with CRLF

The last two characters in any source file within EDK II must be a CRLF: 0x0D followed by 0x0A.

5.1.8 Trigraphs shall not be used

Trigraphs are a construct to allow character representations that do not support all ASCII characters to enter the equivalent of the ASCII character. Trigraphs are three characters long (hence the "tri"). The first two characters are "??" while the third character disambiguates the trigraph. Technically therefore, a[5] could be written a??(5??). Trigraphs have provided both confusing and unnecessary and are prohibited.

5.1.9 In-line assembler shall not be used

There are really no reasons for in-line assembler to be used in EDK II code. The only exceptions in this case are largely associated with the lowest level Architectural Protocols. Using in-line assembly language deviates against the Scope rules defined in Section 1.3 "Scope" because it is an extension to standard C. This.

5.1.10 Do not use #pragma, except for #pragma pack (#).

The only other exception for this would be in the `ProcessorBind.h` file.

5.2 Spacing

Some people claim that spacing is not important and that spacing rules in a coding style are burdensome and bogus. Review the following example to see why spacing is important, especially in C:

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hello,
world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Dishonorable mention, Obfuscated C Code Contest, 1984.

Author requested anonymity.

5.2.1 Vertical Spacing

Use blank lines to

- make code more readable
- group logically related sections together.

5.2.1.1 There shall be only one statement on a line

This requirement does not include `for` loops, where the initial, conditional, and loop statements may go on a single line.

5.2.1.2 The if and while statements are no exception

The conditional and code always go on a separate line.

It should be:

```
if (MyVar != 0) {
  MyTrueCode ();
}
```

5.2.1.3 An open brace '{' goes on the same line as the closing parenthesis ')' of simple predicate expressions

This requirement includes functions and data. A *simple predicate expression* is one containing zero or one operator which fits on the same line as the keyword.

```
while (MyVar != 0) {
```

5.2.1.4 A close brace '}' always goes at the beginning of the last line of the body

Indent the brace to match the first line of the construct.

```
while (MyVar != 0) {
  MyTrueCode ();
}
```

5.2.1.5 A close brace may share a line with the else {, else if () {, and do-while constructs

```
} else {
} else if (foo) {
} while (bar);
```

5.2.1.6 Each sub-expression of a complex predicate expression must be on a separate line

Predicate expressions containing multiple operators with sub-expressions joined by && or || must have each sub-expression on a separate line. The opening brace, ' `{` ' of the body shall be on a line by itself and aligned in the starting column of the associated keyword.

```
while ( ( Code == MEETS_STANDARD)
  && ( Code == FUNCTIONAL))
{
  ShipIt();
}
```

5.2.2 Horizontal Spacing

5.2.2.1 Unless explicitly stated otherwise, space may be one or more spaces long

5.2.2.2 Always put space before and after binary operators.

This space makes both operands much easier to read.

```
if (MyVar != 0) {
  MyTrueCode ();
}
```

5.2.2.3 Do not put space between unary operators and their object

```
If ((--MyInteger) > 0) {
```

5.2.2.4 Subsequent lines of multi-line function calls should line up two spaces from the beginning of the function name

If a function call or function like macro invocation is broken up into multiple lines, then:

- One argument per line, including the first argument on its own line.
- Indent each argument 2 spaces from the start of the function name. If a function is called through a structure or union member, of type pointer-to-function, then indent each argument 2 spaces from the start of the member name.
- Align the close parenthesis with the start of the last argument

```
CopyMem (
  Destination,
  Source,
  SIZE_4KB
  );

Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (DRIVER_NAME_INSTANCE),
                &PrivateData
                );

DEBUG ((
  DEBUG_INFO,
  "The addresses of the 4 buffers are %p, %p, %p, and %p\n",
  Buffer1,
  Buffer2,
  Buffer3,
  Buffer4
  ));
```

5.2.2.5 Always put space after commas or semicolons that separate items

This punctuation is not necessary if no code or comments follow the comma or semicolon.

```
EfiLibAllocateCopyPool (Size, DevicePath);
for (Size = 0; FileName[Size] != 0; Size++) {...
```

5.2.2.6 Always put space before an open parenthesis

The only exception is macro definitions.

```
if (...
while (...
EfiLibAllocateCopyPool (...
```

5.2.2.7 Put a space before an open brace if it is not on its own line

```
if () {
while () {
```

5.2.2.8 Do not put spaces around structure member and pointer operators

```
Data.Index
Pointer->Index = *Ptr;
```

5.2.2.9 Do not put spaces before open brackets of array subscripts

```
Array[(Max + Min) / 2]
```

5.2.2.10 Use extra parentheses rather than depending on in-depth knowledge of the order of precedence of C

(The order of precedence should be the compiler's job, not yours.)

Consider the following expression:

```
8 | 8 == 8
```

On first glance, one might think that the expression would evaluate to `TRUE`. This is not the case. The bitwise OR operator, '`|`', has lower precedence than the equality operator, '`==`'. This results in the expression being evaluated as if one had entered:

```
8 | ( 8 == 8 )
```

This evaluates to the value 9.

5.2.2.11 Align a continuation line with the part of the line that it continues.

```
Value = (GetData (BubbaBaes + BUBBA_HIGH_DATA) << 8) |
         GetData (BubbaBaes + BUBBA_LOW_DATA);
```

5.2.3 File Heading

5.2.3.1 Every new file shall begin with a file header comment block

This block shall begin on the first line of the file. This is a Doxygen file descriptor comment block, so line 1 of the file will be:

```
/** @file
```

And the comment will end with:

```
**/
```

The File Heading comment block is comprised of the following sections: File Description, Copyright, License, and the optional Specification Reference and Glossary sections.

```
/** @file
  File Description.

  Copyright
  License

  Specification Reference

  Glossary Section
**/
```

The following example begins each body line with a tab (two spaces). This is the preferred indentation, but two tabs (four spaces) is also acceptable.

Example

```
/** @file
  Brief description of the file’s purpose.

  Detailed description of the file’s contents and other useful
  information for a person viewing the file for the first time.

  Copyright (C) --20XX, Acme Corporation. All rights reserved.<BR>
  SPDX-License-Identifier: BSD-2-Clause-Patent

  @par Revision Reference:
    - PI Version 1.0

  @par Glossary:
    - IETF - Internet Engineering Task Force
    - NASA - National Aeronautics and Space Administration
**/
```

5.2.3.2 File Description

The file description consists of a brief, one sentence, description of the file's purpose as well as a detailed description of the files purpose and content.

The brief description begins on the second line of the comment block and terminates at the first period. Do not include text indicating that the file is a header, include, declaration, or definition file as this is intrinsically obvious.

The detailed description follows the brief description, separated by a blank line. All subsequent text and blank lines are part of the detailed description until terminated by a Doxygen command or the end of the comment.

Note: The Copyright Notice and License comprise the last paragraph of the detailed file description. They are described separately, below.

The file description describes the purpose of the file (why the file exists), a general description of what it contains, and the relationship of the file to a particular module or modules, if any.

5.2.3.3 Copyright

The first line of the last paragraph of the file description is made up of the copyright notice. The copyright notice must consist of the following text with the `FIRST` and `LAST` symbols replaced with the year the file was created and the year the file was last edited, respectively.

```
Copyright (C) FIRST - LAST, Acme Corporation. All rights reserved.<BR>
```

A file that has been created but not edited in subsequent years would have a copyright notice with a single date, such as:

```
Copyright (C) 2007, Acme Corporation. All rights reserved.<BR>
```

If this file is subsequently edited, the copyright notice would be updated as follows.

```
Copyright (C) 2007 - 2014, Acme Corporation. All rights reserved.<BR>
```

The `FIRST - LAST` format for the copyright date, as described above, is the only format allowed. Do not use a comma separated list or keep updating a single date. The space surrounding the hyphen, ' `-` ', between the `FIRST` and `LAST` dates is optional.

The `<BR>` at the end of the line is required. Doxygen uses XML for its internal format, so repeated spaces and new lines are treated as a single space. The `<BR>` will force Doxygen to start the following text, the license notice, on a new line.

5.2.3.4 License

EDK II code files may contain one of several different licenses, depending upon the location and content of the file. The correct license will be determined by the project leader at the time the file is created. In most cases, the license will be the same as for other files in the module or package.

The preferred license for EDK II is the "BSD+Patent" license. The license for a file is provided in the file header using an SPDX identifier. The following shows the SPDX identifier for the "BSD+Patent" license.

```
SPDX-License-Identifier: BSD-2-Clause-Patent
```

The license follows the copyright notice. The license is separated from the Specification Reference, if present, by a single blank line.

5.2.3.5 Specification Reference

If the file declares or implements something described in one or more UEFI or Industry Standard specifications, it must include a specification reference section in the file description comment block. The section begins with a Doxygen directive:

```
@par Revision Reference:
```

Each specification is listed on a separate line. All specification references must use the list format. An indented line beginning with a dash character, '-', indicates list format.

```
@par Revision Reference:
  - UEFI Version 2.2
  - PI Version 1.0
```

5.2.3.6 Glossary

If the file uses a non-standard Abbreviation or Acronym, it must include a Glossary section in the file description comment block. The section begins with a Doxygen directive:

```
@par Glossary:
```

Each Glossary definition is listed on a separate line. All Glossary definitions must use the list format.

```
@par Glossary:
  - IETF - the Internet Engineering Task Force
  - NASA - National Aeronautics and Space Administration
```

5.3 Include Files

5.3.1 All C include file names shall contain only one extension and it must be .h.

Using only one extension dramatically simplifies build issues and standardizes naming.

5.3.2 Every include file shall have a unique name.

The name of all include files, within the EDK II code base, must be unique. The uniqueness test applies to the entire path used in the #include directive. For example:

```
#include <Peach/Pit.h>
#include "Olive/Pit.h"
#include <MoneyPit.h>
#include "OpenPit.h"
```

These are unique, even if the file name itself is not, as long as the preceding path component is included.

To see why this is necessary, consider a package, `FruitPkg`, that has the following in its DEC file:

```
[Includes]
  Include
  Include/Olive
  Include/Peach
```

Both the `Include/Olive` and `Include/Peach` directories contain a file `Pit.h`. It then becomes ambiguous which file is being referenced if one encounters `#include <Pit.h>` in a source file. This ambiguity would be compounded if another package, also dependent upon `FruitPkg`, contained a `Pit.h` file in its `Include` directory.

Existing automatically generated include files, such as `AutoGen.h`, are exempted from this rule. It is not mandatory to rename pre-existing include files.

Unique file names may also be formed by appending or prepending a short character sequence, such as the module or package name in which the file resides, or an abbreviation of one of these, to the file name.

```
PeachPit.h OlivePit.h MoneyPit.h PitBoss.h
```

5.3.3 Include files shall not consist mainly of include directives.

This type of omnibus header file can significantly increase the time required to rebuild a project. If any of the included files changes, every file that includes the omnibus file will have to be recompiled; even if that particular file did not use the changed header file.

5.3.4 Include files may include only those headers that it directly depends upon.

This maintains proper dependency relationships between the files and allows the header file to be used without prior knowledge of its dependencies.

5.3.5 All include file contents must be protected by a #include guard.

Also known as a "macro guard", use #include guards to avoid multiple inclusions when dealing with the include directive. Adding `#include` guards to a header file makes that file idempotent.

```
#ifndef FILE_NAME_H_
#define FILE_NAME_H_
...
#endif // FILE_NAME_H_
```

Avoid duplicating the guard macro name in different header files. Including a duplicate guard macro name prevents the symbols in the other from being defined. Names starting with one or two underscores, such as `_MACRO_GUARD_FILE_NAME_H_` , must not be used. They are reserved for compiler implementation.

With modern programming practices, particularly include files including other include files, it is almost impossible to avoid including the same file more than once. This can only slow down the processing time and may cause difficult to diagnose issues. To avoid this, the use of *macro guards* is required in all include files to protect against inadvertent multiple inclusion.

- The `#ifndef` shall be on the first line following the file header comment. This location ensures that all code is contained.
- The `#endif` shall appear as the last line in the file. The `#endif` is followed by a comment consisting solely of the guard token. The line shall end with a carriage return (new line) as the last thing in the file, thus ensuring that all code is contained.

5.3.6 Include files shall contain only public or only private data.

Include files must not contain both types of information. Examples of public include files would be protocol definitions or industry standard specifications (EFI, SAL, ACPI, SMBIOS, etc.). Private data would include functions and internal data structure declarations used only within a single module.

5.3.7 Include files shall not generate code or define data variables.

Including code or defining variables can result in issues that are very difficult to debug.

The sample below shows the suggested order of declarations in an include file. Not all types of declarations are present in every file.

```
/** @file
  Public declarations of bizarre protocols for something.

  This is a detailed description of the something for which this
  file exists. Something rotates upon the blarvitz allowing
  implementation of bizarre protocols. If the blarvitz is NULL,
  describe what is happening in more detail. If non-null, then
  you should probably also explain your rationale.

  Copyright (c) 20XX, Acme Corporation. All rights reserved.<BR>
  SPDX-License-Identifier: BSD-2-Clause-Patent

  @par Specification Reference:
    - UEFI 2.3, Chapter 9, Device Path Protocol
    - PI 1.1, Chapter 10, Boot Paths
**/
#ifndef SAMPLE_THING_H_
#define SAMPLE_THING_H_

/* Include directives for dependent header files */

/* Simple defines of such items as status codes and macros */

/* Type definitions */

/* Function prototype declarations */

/* Protocol declarations */

#endif /* SAMPLE_THING_H_ */
```

5.4 Code File Structure

All code generating files use the following general structure. Typically these are C files with an extension of "`.c`".

```
/** @file
  Definitions of bizarre protocols for something.

  This is a detailed description of the something for which this
  file exists. Something rotates upon the blarvitz allowing
  implementation of bizarre protocols. If the blarvitz is NULL,
  describe what is happening in more detail. If non-NULL, then
  you should probably also explain your rotationale.

  Copyright (c) 20XX, Acme Corporation. All rights reserved.<BR>
  SPDX-License-Identifier: BSD-2-Clause-Patent

  @par Specification Reference:
    - UEFI 2.3, Chapter 9, Device Path Protocol
    - PI 1.1, Chapter 10, Boot Paths
**/
/* Include necessary header files here */
#include <DxeCore.h>
#include <Uefi.h>
#include "SampleThing.h"

/* Define external, global and module variables here */

/* Function Definitions */

/* If this is a protocol definition, the
protocol structure is defined and initialized here.
*/
```

5.4.1 Scoping Rules

There are three components of the C language that interact to affect when a particular variable can be used, or not.

- **Scope:** The context in which an identifier is visible constitutes the scope of that identifier..
- **Visibility:** If an identifier can be referenced within a particular scope it is said to be visible.

- **Storage Class:** Allocation of storage space is controlled by the storage class of an identifier.

5.4.1.1 Scope

Understanding the C scoping rules is critical for good programming. The various identifier scopes encountered are: file, prototype, function, and block or local scope.

File Scope

Top-level identifiers and preprocessor macros are said to have *file scope*. Their scope extends from their declaration point to the end of the source program file. A preprocessor macro can go out-of-scope earlier if a `#undef` command that cancels its definition is encountered.

Prototype Scope

The identifiers making up the formal parameters in function prototypes have *prototype scope*. Identifiers with prototype scope have scopes extending from their declaration point to the end of the prototype.

Function Scope

Statement labels have *function scope*. Function scope identifiers have scope which encompasses the entire function body in which they appear. This means that statement labels can be referenced before they are declared.

Block (local) Scope

Formal parameters in function definitions and data defined within compound statements have block scope. Any group of statements that are encompassed within a pair of braces, { }, is a compound statement. The body of a function is also a compound statement. Compound statements can be nested, with each creating a new scope.

Data declarations may follow the opening brace of a compound statement, regardless of nesting depth, and before any code generating statements have been entered. Other than at the outermost block of a function body, this type of declaration is strongly discouraged.

Note: Visual C++ gives all structure declarations File Scope, even though ISO/IEC 9899 specifies that structure declarations may have Block Scope.

5.4.1.2 Visibility

A declaration of an identifier is *visible* in some context if a use of the identifier in that context is associated with that declaration. A declaration might be visible throughout its scope, but it might also be hidden by other declarations whose scope and visibility overlap that of the first declaration.

5.4.1.2.1 No two different identifiers in a function may have the same name nor may any of the names in a function be the same as those declared at a global or module level.

- Identifiers with file scope have the outermost scope. Those with block scope have the innermost scope. Nesting successive blocks creates more inner scopes.
- Only disallow instances where an outer definition is hidden by a second inner definition. This rule is not violated when the first definition is not hidden by the second definition.

The following example shows how the scope visibility of identifiers can overlap and hide each other. Never write code that does this.

```
 1 UINTN MyVar = 7; // File scope
 2
 3 VOID
 4 MyFunction (
 5   OUT UINT32 *MyVar // Function scope
 6   )
 7 {
 8   UINT32 i;
 9
10   for ( i = 0; i < 5; ++i) {
11     UCHAR8 MyVar = i; // Block scope
12     INT16 i = 12;
13
14     MyVar += 'A';
15     process ( MyVar, i);
16   }
17   *MyVar = i;
18 }
19
20 main()
21 {
22   UINT32 George = 4;
23
24   MyFunction ( &George);
25   process ( MyVar, 0);
26 }
27
```

In the above example, there are three declarations of `MyVar` :

- On line 1, `MyVar` is defined with file scope and value 7 This scope extends from line 1 through line 26.
- On line 5, `MyVar` is declared as a formal parameter of MyFunction. Its context extends from line 5 through the end of the function on line 18.
- On line 11, `MyVar` is declared with block scope. From its declaration point on line 11 through the end of the block in which it was declared, this declaration of `MyVar` will be in scope. It is initialized to the current value of `i`.

To test yourself on your understanding of scope and visibility rules, determine the value and associated declaration for each use of `MyVar` before reading further.

The first use of `MyVar` is on line 14 Its associated declaration is on line 11 Each time through the for loop it will have the values 'A', 'B', 'C', and 'D' in succession.

On line 15, `MyVar` is used again with its associated declaration on line 11 with the same succession of values assigned as on line 14.

`MyVar`, on line 17, associates with the formal parameter declaration on line 5 In this case the value of `i`, which is 5, is stored in the location `MyVar` points to. The location is established in the call to `MyFunction` on line 24.

The last use of `MyVar` occurs on line 25 This occurrence associates with the declaration on line 1 and has the value 7.

Things get extremely tricky on line 12 The identifier `i`, which is the control variable for the enclosing `for` loop, is declared with a different type and value. It is ambiguous as to which `i` was wanted in line 11. How does this affect the `for` loop? What will be the value of `i` at lines 15 and 17?

The scope rules tell us the answers, but they still aren't obvious.

- The `i` on line 10 is outside the block, which is the body of the `for` loop. It associates with the declaration on line 8, so the `for` loop will continue to work as expected.
- On line 11, `MyVar` will be initialized to the current value of `i`. It is compiler dependent whether this is the first value of `i`, 0, or the value of `i` for each loop.
- Line 12 declares a new variable `i` which is given the value 12 C scoping rules state that a scope begins at the declaration point, not at the beginning of the block in which the declaration appears. Because of this, the `i` used on line 11 is not the same `i` declared here.
- The value of `i` used on line 15 is 12, the value of the associated and currently visible identifier with that name.

As you can see, using the same identifier at different scopes can be confusing at best. All it takes is one mistake and a bug is introduced that is very hard to locate.

5.4.1.3 Compile-Time Names

So far we have been talking about variable and function identifiers, which have an existence at run time. However, the scope and visibility rules apply equally to identifiers applied to objects that don't necessarily exist at run time: `typedef` names, type tags, enumeration constants, macros, and labels.

Macros and labels follow the rules outlined for them in Section 5.4.1.1 "Scope". The other compile-time names follow the same scope and visibility rules as any variable defined at the same location.

5.4.2 Storage Class

5.4.2.1 extern

A special case of scope and visibility is the *external* identifier, also called an identifier with *external linkage*. All instances of an external identifier, among all the files making up a C module, are forced to refer to the same object or function. Each instance of the external identifier must be declared with the same type in each file or else the result is undefined.

External names are declared using the `extern` keyword. In C89, and earlier dialects, external identifiers declared at file scope are implicitly extern. But, not all identifiers declared `extern` were external.

The compilers supported by EDK II follow the C99 specification for `extern`. All declarations of the external variable must use the `extern` keyword. There must be a single definition of the external variable which does not use the `extern` keyword. This definition may also include an initializer.

By convention, external names are declared at the top level of a C program and therefore have file scope. For EDK II, external names must be declared in a single header file and must be defined at the top level of a C file as specified in Section 5.4.1.3 "Compile-Time Names".

Thus, while it might be legal C, do **not** declare external variables anywhere other than at the top level of a file as specified by this document.

5.4.2.2 Static

An object declared `STATIC` has either file or block scope.

5.4.2.2.1 Do not reuse an object or function identifier with static storage duration.

Throughout the set of source files defined within a single .inf file, do not reuse an identifier with static storage duration. The compiler may not be confused by this, but the user may confuse unrelated variables with the same name.

5.4.2.2.2 Functions should not be declared STATIC.

Some source-level debuggers are unable to resolve static functions. Until it can be verified that no one is dependent upon a debugger with this limitation, it is strongly recommended that functions not be declared static.

5.5 Preprocessor Directives

5.5.1 #include

5.5.1.1 Use the proper file delimiters when including files

Use either double quotes or angle brackets. A file name delimited by angle brackets, `<FileName.h>`, indicates that the compiler should search the default include paths and those specified by ' `-I` ' directives on the compiler command line (System Include Files). If one uses double quotes to delimit the file name, " `FileName.h` ", the compiler will first search the directory containing the file being compiled before then searching the System include paths.

The general rule to be derived from this is that one should use double quotes to include files in the same directory as, or a subdirectory of, the file being compiled, and angle brackets for all other include files.

```
#include <Uefi.h> /* System include file. */
#include "SampleThing.h" /* In same directory as the C file. */
```

5.5.1.2 Include file paths shall be relative and shall not contain ".." elements.

All include paths must be relative to either the current .c file, or a predefined include directory.

5.5.2 Macros

5.5.2.1 Functional macros are generally discouraged.

Parameterized macros are difficult to debug, have difficult syntax, do not encourage strong commenting, and generally negatively impact maintainability and understandability of code. Macros are appropriate for some concepts, such as containment records and include path abstraction.

5.5.2.2 Macros follow the standard data naming conventions used by typedef and #define.

The main reason for making a macro different from a function is the difference in the order of precedence that can occur between poorly constructed macros and functions.

5.5.2.3 Overusing parentheses is strongly encouraged.

An order-of-precedence bug in a macro is very hard to debug. The following are examples of macro construction:

```
#define BAD_MACRO(a, b) a*b
#define GOOD_MACRO(a, b) ((a)*(b))
```

The following examples should explain the difference between `BAD_MACRO ()` and `GOOD_MACRO ()` :

- `BAD_MACRO (10, 2)` and `GOOD_MACRO (10, 2)` both evaluate to 20.
- `BAD_MACRO (7+3, 2)` returns 13 = 7 + (3*2).
- `GOOD_MACRO (7+3, 2)` returns 20.

Also, consider the following expression:

```
8 | 8 == 8
```

On first glance, one might think that the expression would evaluate to `TRUE` . This is not the case. The bitwise OR operator, ' `|` ', has lower precedence than the equality operator, ' `==` '. This results in the expression being evaluated as if one had entered:

```
8 | ( 8 == 8 )
```

This evaluates to the value 9 The desired result of `TRUE` , (1), can be achieved by specifying the expression as:

```
((8 | 8) == 8)
```

5.5.2.4 Macros must have comment blocks like functions.

Macros can be very difficult to debug, so explicit descriptions of the input, output, and behavior are required. This is true whether it is a parameterized or a simple substitution macro.

5.5.2.5 Parameterized macro definitions shall not have a space between the name and the '('

Failure to do this will cause the build to break.

```
#define GOOD_MACRO(a, b) ((a)*(b))
```

This is because the compiler has no way to differentiate between

```
#define SIMPLE_MACRO (a) (TXT)
```

which substitutes all subsequent occurrences of SIMPLE_MACRO with (a) (TXT), and

```
#define PARAM_MACRO(a) (a) (TXT)
```

which defines a parameterized macro.

5.5.2.6 When using a macro, there must be a space between the name and the '(' to comply with function calling conventions.

Failure to separate macro names from parameters negatively impacts readability and consistency with other coding style rules.

```
GOOD_MACRO (7+3, 2)
```

5.5.2.7 Single-line Functions

Most uses of parameterized macros can be replaced by one or two line functions. The compilers will almost always in-line the function resulting in the same effect as the parameterized macro. An additional benefit of single-line functions is the type checking the compiler can now provide.

5.6 Declarations and Types

5.6.1 Common Data Types

5.6.1.1 The UEFI Specification defines a set of common data types that must be used to ensure portability between different compilers and processor architectures.

Any abstract type that is defined must be constructed from other abstract types or from common EFI data types.

5.6.1.2 The use of int, unsigned, char, void, static, long is a violation of the coding convention.

The corresponding EFI types must be used instead.

"EFI Data Types" below contains the common data types that are referenced in the interface definitions defined by this specification. Per the *UEFI Specification*, version 2.3.1:

"Unless otherwise specified, all data types are naturally aligned. Structures are aligned on boundaries equal to the largest internal datum of the structure, and internal data is implicitly padded to achieve natural alignment."

Table 6 EFI Data Types (slightly modified from UEFI 2.3.1)

Mnemonic	Description
`BOOLEAN`	Logical Boolean. 1-byte value containing a 0 for `False` or a 1 for `True`. Other values are undefined.
`INTN`	Signed value of native width. (4 bytes on IA-32, 8 bytes on X64, and 8 bytes on the Intel(R) Itanium(R) processor family)
`UINTN`	Unsigned value of native width. (4 bytes on IA-32, 8 bytes on X64, and 8 bytes on the Intel(R) Itanium(R) processor family)
`INT8`	1-byte signed value.
`UINT8`	1-byte unsigned value.
`INT16`	2-byte signed value.
`UINT16`	2-byte unsigned value.
`INT32`	4-byte signed value.
`UINT32`	4-byte unsigned value.
`INT64`	8-byte signed value.
`UINT64`	8-byte unsigned value.
`CHAR8`	1-byte character.
`CHAR16`	2-byte character. Unless otherwise specified, all strings are stored in the UTF-16, 2-byte, encoding format as defined by the Unicode 2.1 and ISO/IEC 10646 standards.
`VOID`	Undeclared type.
`EFI_GUID`	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64bit boundary.
`EFI_STATUS`	Status code. Type `UINTN`.
`EFI_HANDLE`	Handle to a device driver. Type `VOID *`.
`EFI_EVENT`	Handle to an event structure. Type `VOID *`.
`EFI_LBA`	Logical block address. Type `UINT64`.
`EFI_TPL`	Task priority level. Type `UINTN`.

"Modifiers for Common EFI Data Types" defines modifiers that are used in function and data declarations. The `IN`, `OUT`, `OPTIONAL`, and `UNALIGNED` modifiers are used only to qualify arguments to a function. They should never appear in a data type declaration. The `EFIAPI` modifier is used to ensure the correct calling convention is used between different modules that are not linked together. Use this modifier at the entry of drivers, events, and member functions of protocols.

The `EFIAPI` modifier must be used for all UEFI defined API functions, as well as for any function that takes a variable number of arguments. All protocol functions as well as public functions exposed by drivers must also be declared `EFIAPI`. This establishes a common calling convention for functions that could be referenced by other code that has potentially been built using a different compiler, with a different native calling convention.

Table 7 Modifiers for Common EFI Data Types (reference the UEFI Specification and Beyond Bios)

Mnemonic	Description
`IN`	Datum is passed to the function.
`OUT`	Datum is returned from the function.
`OPTIONAL`	Datum that is passed to the function is optional, and a `NULL` may be passed if the value is not supplied.
`UNALIGNED`	Datum is byte packed and is not naturally aligned.
`VOLATILE`	Declares a variable to be volatile and thus exempt from optimization to remove redundant or unneeded accesses. Any variable that represents a hardware device should be declared as `VOLATILE`.
`CONST`	Declares a variable to be of type `const`. This type is a hint to the compiler to enable optimization and stronger type checking at compile time.
`EFIAPI`	Defines the calling convention for EFI interfaces. All EFI intrinsic services and any member function of a protocol must use this modifier on the function definition.

5.6.2 Constants

5.6.2.1 EFI Constants

"EFI Constants" below lists the EFI constants that should be used to represent certain concepts.

Table 8 EFI Constants

Mnemonic	Description
`TRUE`	One = (1 < 2) == 1; Any non-zero value is `TRUE`.
`FALSE`	Zero = (2 < 1) == 0
`NULL`	`VOID` pointer to zero. ((void*)0)

5.6.2.2 Enumerated Types

- The elements of the enumerated type must follow the data and function naming convention.

  The `enum` shall be declared as a `typedef` with the name of the `typedef` following the type and macro naming conventions in` "Type and Macro Names".

- The last element of the enum should be a maximum member element.

  This convention allows for bounds checking on an `enum` to support debugging and sanity checking the value that is assigned to an `enum`. It is also recommended that the `enum` members be named carefully, such that their names would not tend to collide with other variable or function names.

  ```
  typedef enum {
    EnumMemberOne,  ///< Automatically initialized to zero.
    EnumMemberTwo,  ///< This has the value 1
    EnumMemberMax   ///< The value 2 here indicates there are two elements.
  } ENUMERATED_TYPE;
  ```

  This obviously will not work if values are explicitly assigned out-of-sequence or are duplicated.

- All constants that will be used "as is" should be declared as enums.

  An enum does not cause code to be generated until the enum is used, whereas a const int will cause space for the int to be allocated as well as the code generated whenever the int is used. The use of enums allows type checking to be performed, while the use of macros does not `.`

5.6.2.3 Macro Constants

Constants that will be used to construct other values should be declared as macros. These include bit field definitions and masks.

5.6.2.4 Pointers and Constants

There are three different ways pointers and constants can interact:

- Pointer to Constant:

  ```
  CONST UINTN * PointerToConst;
  ```

  `PointerToConst` is a variable pointer to a constant `UINTN`.

- Constant pointer to variable:

```
UINTN * CONST ConstPointer;
```

`ConstPointer` is a constant pointer to a variable `UINTN`.

- Constant pointer to constant:

```
CONST UINTN * CONST ConstPointerToConst;
```

`ConstPointerToConst` is a constant pointer to a constant `UINTN`.

5.6.3 Structure Declaration

Structures shall be declared as a `typedef` with one of two different styles depending on the use of the structure. If the structure is not self-referential, or there is no forward reference to it, the structure may be defined anonymously; see Section "Structure Reference".

This anonymous definition is valid because we typedef the structure in the definition.

- The structure name and typedef name shall follow the type and macro naming conventions in "Type and Macro Names" on page 24`
- Structure instances: Variables, parameters, members, etc., must follow the file, function, and data naming conventions in "Identifiers" through "Name Space Rules".
- Structures are always defined in a `typedef struct name {...} type;` format.

The "name" tag is allowed only if the structure is self-referential or the target of a forward reference.

5.6.3.1 Structures shall not be directly declared.

The following are not allowed:

```
struct name {...}; // OK if object of forward reference
struct {...} variable; // Never OK
struct name {...} variable; // Never OK
```

5.6.3.2 Structure Declaration with Forward Reference or Self-Reference

```
/// Sample forward declaration of a structure.
typedef struct EFI_STRUCT_NAME EFI_STRUCT_NAME;

/// Sample self-referential structure declaration.
typedef struct EFI_STRUCTURE_NAME {
  ...
  struct EFI_STRUCTURE_NAME *StructPointer;  ///< Sample self reference
} EFI_STRUCT_NAME;
```

5.6.3.3 Structure Declaration without Forward Reference

```
/** Brief description of sample structure declaration.
  *
  * Detailed description of purpose and use of this structure.
**/
typedef struct {
  Atype  memberOne;  ///< Briefly describe memberOne
  ...
  Ztype  memberN;    ///< Briefly describe memberN
} EFI_STRUCTURE_NAME;
```

5.6.3.4 Bit Fields

A member of a structure or union may be declared to consist of a specified number of bits (including a sign bit, if any). That member is referred to as a bit-field. Bit fields differ from other members in that:

- Bit fields may only be of type `INT32`, `signed INT32`, `UINT32`, or a typedef name defined as one of the three `INT32` variants.
- It is compiler defined whether `INT32` is signed or unsigned.
- The order of allocation of bit-fields within a storage unit is compiler defined.
- The alignment of the addressable storage unit is unspecified.
- A bit-field may not extend from one storage unit into another.

A bit-field with only a colon and a width (no declarator), indicates an unnamed bit-field. Unnamed bit-fields are useful for padding to conform to externally imposed layouts.

Specifying a bit-field with a width of 0 (zero) indicates that no further bit-fields are to be packed into the unit in which the previous bit-field, if any, was placed.

5.6.3.4.1 Visual C++ Specific

- The alignment requirement for each non-bit-field member is the same as the largest alignment requirement of the members. Thus, every member will have the same alignment.
- A "plain" `int` bit-field is treated as a `signed int` bit field.
- Bit fields are allocated within a storage unit from least-significant to most-significant bit.

5.6.3.4.2 GCC Specific

- The alignment requirement for non-bit-field members of structures is determined by the target ABI.
- By default, a "plain" `int` bit-field is treated as a `signed int`, but this may be changed by the '-funsigned-bitfields' option.
- The order of allocation of bit-fields within a unit is determined by the target ABI.

5.7 C Programming

5.7.1 Function Definition Layout

Every function should be constructed to the conventions described in this section.

5.7.1.1 Precede the function with a Doxygen style comment block.

The function header comment block must describe the intent and purpose of the function, parameters used, and return values.

5.7.1.2 The line immediately following the function header comment block optionally specifies the storage class of the function.

If the storage class is not specified, the return type will be on the first line following the comment block.

5.7.1.3 The next line of the function definition must specify the function's return type.

If the function does not return a value, this line must contain `VOID`.

5.7.1.4 The next line contains any optional functional modifiers.

For example, `EFIAPI`, or other valid modifiers.

5.7.1.5 The next line contains the function name, left justified, followed by the beginning of the parameter list, " (."

No parameters are allowed on this line. If no parameters are present, it is acceptable to place the closing parenthesis on the same line, " `(VOID)` ".

5.7.1.6 A function that takes no parameters shall be declared with VOID as the parameter list.

Declaring a function with an empty parameter list, `()`, is prohibited.

5.7.1.7 The next lines contain parameters.

Each line will contain a single argument and will start indented two spaces (one tab stop). Type and argument columns should be aligned to maximize readability and should include appropriate spacing to ensure this alignment. No comments are allowed in this region. Parameters are documented clearly in the function header comment block.

5.7.1.8 The closing parenthesis is on its own line and is also indented two spaces.

A valid exception exists if the function has no parameters.

5.7.1.9 Function prototypes have the same form as function definitions, with the exception of requiring a semicolon after the closing parenthesis of the parameter list.

5.7.1.10 The opening brace of the function body is alone on the next line.

Both the opening and closing brace of the function body must be aligned in the left most column. All other lines of the function body are indented in multiples of two spaces.

```
/** Brief and Detailed Descriptions.

  @param[in]      Arg1  Description of Arg1.
  @param[in]      Arg2  Description of Arg2, which is optional.
  @param[out]     Arg3  Description of Arg3.
  @param[in,out]  Arg4  Description of Arg4.

  @retval  EFI_SUCCESS   Description of what EFI_SUCCESS means.
  @retval  !EFI_SUCCESS  Failure.

--*/
EFI_STATUS
EFIAPI
FooName (
  IN UINTN      Arg1,
  IN UINTN      Arg2, OPTIONAL
  OUT UINTN     *Arg3,
  IN OUT UINTN  *Arg4
  );
{
  UINTN Local;
  ...
}
```

5.7.1.11 Each argument variable's type specification should be preceded by IN and/or OUT modifiers.

The modifiers are used to indicate whether the argument is an input or output variable. It is strongly suggested that the `IN` variables are first and `OUT` variables come next. If data is both passed in and passed out through a variable, then it should be marked as both `IN` and `OUT`. A buffer that is passed into a routine that modifies the contents of the buffer is marked as `IN` and `OUT`. "Parameter Modifiers" below describes the usage of the `IN` and `OUT` modifiers.

The `IN` and `OUT` modifiers may be omitted if the mandatory `@param[in,out]` descriptions are provided in the function header comment.

Table 9 Parameter Modifiers

Mnemonic	Description
`IN`	Passed by value. For C, this modifier is any argument whose name **is not** preceded by an asterisk (*).
`IN`	Passed by reference, and referenced data **is not** modified by the routine. An asterisk precedes the argument name.
`OUT`	Passed by reference, and the referenced data **is** modified by the routine. The passed-in state of the referenced data is not used by the routine.
`IN OUT`	Passed by reference, and the passed-in referenced data is consumed and then modified by the routine.
`OPTIONAL`	Indicates that if a pointer argument is `NULL`, it is not present. If the value is not `NULL`, then it is a valid argument and may be used.

5.7.1.12 The body of a function is contained within open and close braces that must be in the first column.

5.7.1.13 File-Scope data definitions must be the first code in a module.

The type definition must start indented and be followed by the variable name with at least one indent between the two. Each variable name must have its own line; do not use a comma to separate multiple declarations. Do not comment data declarations; they should contain selfdescribing names. If comments are required for complex data declarations, place the comments in the include file that defines the complex data type, or make comments in the routine description block. This restriction does not apply to the members of structures, unions, or enums.

5.7.1.14 File-Scope Data definitions appearing anywhere but at the beginning of the module are illegal.

5.7.1.15 Function Headings

5.7.1.15.1 Every new function created as part of EDK II must have a function header comment block immediately preceding the function definition.

The function header comment block takes the form shown in the example below.

```
/**
  Brief description of the function's purpose.

  Detailed description of the function's purpose and how it
  works. Describe algorithms, side-effects, or other attributes
  of the function that would be of use to a programmer unfamiliar
  with the code.

  @param[in] Arg1  Description of Arg1 This can span multiple
                   lines if needed.
  @param[in] Arg2  Description of Arg2.

  @retval  EFI_SUCCESS  Procedure returned successfully.
  @retval  VALUE        Line for each possible return value.
**/
EFI_STATUS
Foo (
  IN UINTN  Arg1,
  IN UINTN  Arg2
  )
{
  // Function body
}
```

5.7.1.15.2 Function comment headings must be present in both the .c file containing the function implementation and in the .h file containing the function prototype.

The function comment heading in the .c file shall, at a minimum, be a duplicate of the heading in the include file. In all cases, the function comment heading in the include file is the authoritative source. Function comment headings in .c files may contain implementation specific comments which are not permitted in the `.h` file.

5.7.1.15.3 Any function that is shared must have the function prototype in a shared .h file.

Private functions may be declared in the .c file implementing them if they are not used in any other file. All such private function declarations must be present at the beginning of the `.c` file, before any function implementations and after global data.

5.7.1.15.4 All public types, macros, and declarations for a module shall be in a single include file resident in the hierarchy of the top-level include directory of the package the module is part of.

5.7.2 Predicate Expressions

5.7.2.1 Boolean values, variable type `BOOLEAN`, do not require explicit comparisons to `TRUE` or `FALSE`.

Non-Boolean comparisons must use a compare operator (`==` , `!=` , `>` , `< >=` , `<=`).

5.7.2.2 A comparison of any pointer to zero must be done via the `NULL` type.

"Predicate Expression Examples" below shows examples using the following:

```
BOOLEAN Done;
UINTN Index;
VOID *Ptr;
```

Table 10 Predicate Expression Examples

Incorrect	Correct
`if (Index) {`	`if (Index != 0) {`
`if (!Index) {`	`if (Index == 0) {`
`if (Done == TRUE) {`	`if (Done) {`
`if (Done == FALSE) {`	`if (!Done) {`
`if (Ptr) {`	`if (Ptr != NULL) {`
`if (Ptr ==0) {`	`if (Ptr == NULL) {`

5.7.2.3 Comparison of unsigned integer types to be >=0 is permitted.

There has been some controversy on this since unsigned integers are always >=0, and some compilers will emit a warning when this type of construct is seen. This is perfectly valid code and, with some compilers, is needed for limit checking against enumerated types. These compilers will assign the type to the enum based upon the range of values the enum is assigned at compile time.

```
if ((foo >= 0) &&
    (foo < MaxVal))
{ ...
```

If one wants to be perfectly safe in their comparison, the potentially unsigned term can be cast to a signed integer type of sufficient size; if the range of possible values is known. `if ((INTN)foo >= 0) { ...`

5.7.2.4 The ordering of terms in predicate expressions may impact performance significantly.

This is because only enough of the expression has to be evaluated to determine the outcome. For example, the first term in an OR expression that evaluates to `TRUE` is sufficient to determine that the entire OR expression will be `TRUE`.

The following code sample consists of a predicate expression within a for loop. The predicate expression is a compound AND expression with four terms. Each term is itself a simple predicate expression.

This is a real example from a previous version of the EDK II code and was used in the implementation of the PERF_END functionality. It finds the first open (EndTimeStamp is zero) performance trace record matching the Handle, Token, and Module parameters of a PERF_END invocation.

```
for (Index = 0; Index < NumberOfEntries; Index++) {
  if (( LogEntryArray[Index].Handle == (EFI_PHYSICAL_ADDRESS)(UINTN) Handle)
        && AsciiStrnCmp (LogEntryArray[Index].Token, Token, PEI_PERFORMANCE_STRING_LENG
TH) == 0
        && AsciiStrnCmp (LogEntryArray[Index].Module, Module, PEI_PERFORMANCE_STRING_LE
NGTH) == 0
        && LogEntryArray[Index].EndTimeStamp == 0
      )
  {
    break; // Exit the enclosing for loop.
  }
}
```

While there is nothing really wrong with this code, a little knowledge of the data it will be working on will allow us to significantly speed things up. Again taking from a real-world example:

- NumberOfEntries = 25,172
- Target entry is at Index 25,159
- There are 532 completed measurement records with the same Handle, Token, and Module values prior to the target.
- There are 6 measurement records with the same Handle and different Token and Module values prior to the target.
- There are 25,157 completed entries prior to the target.

From this information, we can see that the terms, in order of importance, are:

1. EndTimeStamp 1 match (2 max over life of a session)
2. Handle 538 matches
3. Module 532 matches
4. Token 532 matches

We also can determine that this ordering is valid for any measurement. Re-ordering the predicate expression using this information produces:

```
for (Index = 0; Index < NumberOfEntries; Index++) {
  if ( LogEntryArray[Index].EndTimeStamp == 0
       && LogEntryArray[Index].Handle == (EFI_PHYSICAL_ADDRESS)(UINTN) Handle
       && AsciiStrnCmp (LogEntryArray[Index].Module, Module, PEI_PERFORMANCE_STRING_LE
NGTH) == 0
       && AsciiStrnCmp (LogEntryArray[Index].Token, Token, PEI_PERFORMANCE_STRING_LENG
TH) == 0
     )
  {
    break; // Exit the enclosing for loop.
  }
}
```

The new ordering results in 538 fewer 64-bit integer comparisons and 1,069 fewer string comparisons for this single PERF_END invocation. Considering that there will be 25,170 PERF_END invocations one can see how the savings add up to a sizable amount: around 2.7 seconds if the string comparisons only took 100ns each.

5.7.3 Flow Control Statements

5.7.3.1 The body of looping statements must be compound statements.

The `if`, `for`, `do`, and `while` statements shall use a compound statement as their bodies, even if it is necessary to use a null compound.

5.7.3.2 Null compound statements shall occupy three source lines.

The opening brace shall be on the same line as the controlling keyword, followed by a line consisting of a properly indented semicolon, then closed with a closing brace, aligned in the same column as the opening keyword, as the first non-white-space character on the third line.

5.7.3.3 Any loop that contains no code in the body must use a null compound statement as the body.

For example, the following statement has a null compound for its body.

```
for (Index = 0; Index < MAX_INDEX; Foo[Index] = Index++) {
  ;
}
```

Avoid this type of construct because it is harder to read than other control structures.

5.7.3.4 if Statements

5.7.3.4.1 The if statement shall be followed by a space and then the open parenthesis.

- The open parenthesis is followed by a conditional (predicate) expression and a close parenthesis.
- There is a space after the close parenthesis followed by an open brace.
- The code body follows and is indented by two spaces.
- The close bracket is on its own line and indented to the same level as the if.

The following is the allowed if construct:

```
if (TRUE) {
  IamTheCode ();
}
```

5.7.3.4.2 When an else is used, it may start on the same line as the close brace of the if, or be on the following line and aligned with the closing brace.

- A single space must separate `else` from the close and open braces.
- An open brace always follows the `else`.

The following are the allowed `if` and `else` constructs:

```
if (EXPR_1) {
  IamTheCode ();
} else if (EXPR_2) {
  IamTheCode ();
}
else { // Alternative format
  IamTheCode ();
}
```

5.7.3.5 while and do - while Statements

`while` statements execute their body zero or more times depending upon the result of a predicate expression. The expression is evaluated at the beginning of the loop and the body is executed if the expression evaluates to `TRUE`. Code must be constructed so that nothing outside of the while loop is dependent upon the body of the loop having been executed.

```
while (TRUE) {
  IamTheCode ();
}
```

For those cases where the body of the loop must be executed at least once, there is the do-while statement. In this statement, the predicate is evaluated at the end of the loop, with the statement's body being re-executed as long as the predicate evaluates to `TRUE`.

```
do {
  IamTheCode ();
} while (TRUE);
```

5.7.3.6 for Statements

A `for` loop occupies a minimum of three lines. It is comprised of the `for` keyword followed by three expressions within parentheses, followed by a compound statement. The three expressions within parentheses are separated by semicolons, ';', and consist of the initialization, conditional, and increment expressions. Any one or combination of these expressions may be omitted if needed.

```
for (Index = 0; Index < MAX_INDEX; Index++) {
  IamTheCode (Index);
}
```

5.7.3.7 switch Statements

The following is a switch statement:

```
switch (Variable) {
case 1:
  IamTheCode ();
  break;

case 2:
  IamTheCode ();
  break;

default:
  IamTheCode ();
  break;
};
```

The case statements are indented either zero tab stops or one tab stop and the bodies of each case one tab stop from the case.

The closing brace of the switch statement should be in the same column as the '`s`' in the `switch` keyword.

Always include the `default` case. It should always end with a `break` statement, even if the `break` is the last thing before the closing brace.

The indenting of the case statements should be consistent with all other files in the module containing the new .c file. Use the legacy, zero indent, style if any other file within the module uses that style, otherwise you may indent a single tab stop.

A descriptive comment is required in cases where the intention is for a preceding `case` to fall through to the next `case`. That is to say, a descriptive comment is required where there is no `break` preceding the second or beyond `case` in a `switch` block.

5.7.3.8 Goto Statements should not be used (in general)

In almost all cases, it is possible to write the code so that a `goto` is not needed. If a `goto` is used, be ready to defend it during review.

It is common to use `goto` for error handling and thus, exiting a routine in an error case is the only legal use of a `goto`. A `goto` allows the error exit code to be contained in one place in the routine. This one place reduces software life cycle maintenance issues, as there can be one copy of error cleanup code per routine.

The `goto` follows the normal rules for C code. The label must be indented one level less than the code it is marking.

```
Status = IAmTheCode ();
if (EFI_ERROR (Status)) {
  goto ErrorExit;
}

IDoTheWork ();

ErrorExit:
  return Status;
```

The above example could have been rewritten as below, eliminating the need for a `goto`.

```
Status = IAmTheCode ();
if (! EFI_ERROR (Status)) {
  IDoTheWork ();
}
return Status;
```

5.7.4 Structure Definitions

5.7.4.1 Structure Reference

```
/** Sample reference to a structure.
 *
 * Sample code showing a reference to a structure as a
 * function parameter. This function should be called as:
 * FooName (&Structure);
 *
 * @param[in] Arg1 A pointer to the sample structure.
**/
VOID
EFIAPI
FooName (
  IN EFI_STRUCTURE_NAME  *Arg1
  );
```

In any situation, definition of structure instances shall be in the following form:

```
EFI_STRUCTURE_NAME  StructureName;
```

Never pass structures as function parameters by value. Use the "address of" operator, '`&`', and pass structures by reference.

```
FooName (&StructureName);
```

5.8 Error Handling and ASSERT

`ASSERT` macros are development and debugging aids and shall never be used for error handling.

Assertions are used to catch conditions caused by programming errors that are resolved prior to product release.

The EDK II PCD, `PcdDebugPropertyMask`, can be used to enable or disable the generation of code associated with `ASSERT` usage. Thus, all code must be able to operate, and recover in a reasonable manner with `ASSERT`s disabled.

Parameters and conditions that are beyond the programmers control need to be checked programmatically. Care must be taken, though, to ensure that the need for programmatic error handling is minimized.

The `ASSERT_EFI_ERROR`, `ASSERT_PROTOCOL_ALREADY_INSTALLED`, and all other `ASSERT` macros defined in `DebugLib.h` are covered by this rule.

6 DOCUMENTING SOFTWARE

6.1 Documentation Concepts

A program is meant to be read by the programmer, by another programmer at a future date, and by a machine. In this sense it is a kind of publication.

However, the machine is concerned with whether the program compiles, while people are necessarily concerned about coding conventions or style.

Sensible and consistently applied typographic (stylistic) conventions are important to the creation of a clear presentation. Conventions are created to facilitate clarity, should not become an end in and of themselves. As Rob Pike points out, we should avoid typographic silliness and decoration:" . . . keep comments brief and banner free. Say what you want to say in the program, neatly and consistently. Then move on."

6.1.1 Requirements

The EDK II code documentation shall fulfill the following main functions:

6.1.1.1 Instruction

It shall serve as:

- Working instructions (e.g. error and interrupt handling, data archiving, ...)
- A working basis to avoid duplication of work
- A working basis for software maintenance (e.g. updates, upgrades, troubleshooting)
- A basis for brief instruction and training of new staff members; simple training courses

6.1.1.2 Verification, Validation, and Objective Evidence

- Establishes traceability between the code and product requirements
- Reference for internal and external auditors (product liability, ISO 9001 certification)
- Facilitates project management and monitoring
- Increases testability of the software

6.1.1.3 Communications

- Establishes an uniform communication basis for:
- All software developers
- Contractors and customers
- Increases software re-usability
- Increases transparency of the software

6.1.2 Interfaces or Protocols

The EDK II architecture forces explicit documentation as it encourages the use of multiple, independent modules. The mechanism for these modules to communicate is a protocol. A protocol is an instance of an API (a class in C++ vernacular) that is named by a GUID. The GUID defines the data representation and member functions of a protocol. Any change to the behavior of a protocol requires the GUID to change. Given this property, each protocol used in EDK II must have an interface document.

6.2 Comments

Commenting has always been a delicate matter, requiring taste and judgment. One should assume that others reading the code know the implementation language. Code that is clear, using good type and variable names, should be self-explanatory - at a micro level. Comments should explain the less-than-obvious aspects of the code, at a macro level.

Note: The comments need to explain why things were done and the big picture of how something works and the ways in which the various pieces relate to each other.

By definition, misleading comments can cause confusion. When you modify code, you should always check the comments to ensure that they accurately document the modified code. Comments are not checked by the compiler, and are therefore not guaranteed to be correct. Comments are even more of a risk following code modification.

Write your comments assuming that they are going to be read by someone with minimal familiarity with the code. Clarity is important, but one should also strive for terse and concise comments. One should be able to see both the comment, and the code being commented on, on the same screen.

6.2.1 Only use C style, "/*", comments on the same line as pre-processor directives, and in Doxygen-style file and function header comment blocks.

Compile can vary in their support for use of `//` in preprocessor directives (e.g. `#define`). Note that the mixing of `/* ... */` and `//` is not handled consistently. This goes beyond style issues; various (pre C99) compilers may not behave the same.

6.2.2 Avoid banners or other decoration around blocks of comments.

Banners can be useful to highlight logical divisions within a file; such as before vital sections. This type of usage should be minimized.

6.2.3 Do not include jokes or obtuse references in comments.

```
"Out of cheese error! Redo from start."
```

6.3 What NOT to Comment

6.3.1 Do not repeat the code or explain it in a comment.

Comments should clarify the intent of the code or explain higher-level concepts. The code itself should be clear and self-documenting.

There is a famously bad comment:

```
i = i + 1; // Add one to i
```

This comment provides no information beyond what is already obvious from reading the code. There are even worse ways to do it, such as:

```
/**********************************
* *
* Add one to i *
* *
**********************************/

      i=i+1;
```

6.3.2 Do not leave markers in the code.

Don’t leave flags, such as your name, in the code. They may have meaning to you but they do not to other people or projects. Don’t make fancy patterns in your comments so you can search for them later. Always consider that the code and comments represent both you and your company and will very likely be publicly available. The presence of flags, like `BugBug`, or `ToDo` statements indicating that comments should be added, reflect poorly upon the programmer.

6.3.3 Sections of code shall not be “commented out”.

Where sections of source code must not be compiled, use conditional compilation (such as `#if` or `#ifdef` constructs with a descriptive comment) to meet this requirement. C does not support nested comments, and the application of start and end comment markers to meet the requirement of source code that must not be compiled is a dangerous practice. This is because comments already existing in the section of code would change the outcome.

6.3.4 Do not comment out, or otherwise disable, previous revisions of the code.

Rely on your source control system to retain history, not your code.

6.3.5 Do not use the character sequence “/*” within a comment.

C does not support the nesting of comments, despite the existence of such support (as a language extension) within some compilers. Comment start with `/*` and end when the first `*/` is encountered.

6.4 What You Must Comment

6.4.1 Comment function declarations if public, or implementations if private and not declared.

You must describe the purpose of the function and any side effects. Also describe the purpose or meaning of each parameter and return value.

6.4.2 Comment complicated, tricky, or sensitive pieces of code.

What the code is doing must be made clear. Making the code cleaner is often better than adding more comments.

6.4.3 Comment higher-level concepts in the code.

Focus on the why and not the how.

6.4.4 Comment data structure declarations and #define statements.

The include files should be sufficient to understand what data or code are for. It should not be necessary to search for all references to something to understand its purpose and use. If more than one instance of a structure or union is instantiated, comment each one as to its intended purpose. It should be clear from these comments why there are multiple instances and how each instance differs.

6.4.5 File comments should include the version number of the industry standard to which you are coding.

When possible, you should also list the requirements that are satisfied by the code.

6.4.6 Comment spurious variable assignments.

A compiler or static code analyzer may warn that an object with automatic or allocated storage duration is read without having been initialized, while visual inspection reveals that this is impossible.

In order to suppress such a warning (which is emitted due to invalid data flow analysis), developers explicitly assign the affected object the value to which the same object would be initialized automatically, had the object static storage duration, and no initializer. (The value assigned could be arbitrary; the above-mentioned value is chosen for stylistic reasons.) For example:

```
UINTN LocalIntegerVariable;
VOID  *LocalPointerVariable;

LocalIntegerVariable = 0;
LocalPointerVariable = NULL;
```

This kind of assignment is difficult to distinguish from assignments where the initial value of an object is meaningful, and is consumed by other code without an intervening assignment. Therefore, each such assignment must be documented, as follows:

```
UINTN LocalIntegerVariable;
VOID  *LocalPointerVariable;

//
// set LocalIntegerVariable to suppress incorrect compiler/analyzer warnings
//
LocalIntegerVariable = 0;
//
// set LocalPointerVariable to suppress incorrect compiler/analyzer warnings
//
LocalPointerVariable = NULL;
```

6.5 Types of Comments

Comments can be either global or internal.

6.5.1 Global Comments

Global, or strategic, comments occur outside of function definitions and provide structural or algorithmic information about the file or program. Global comments are almost always special Doxygen comment blocks. See Section 6.6 "Introducing Doxygen".

6.5.1.1 Comments are allowed on structure member declarations:

```
typedef struct {
  EFI_SAMPLE_STRUCTURE *StructurePointer; ///< Sample comment #1
  UINT64               SimpleVariable;    ///< Sample comment #2
} EFI_STRUCTURE_NAME;
```

6.5.2 Internal Comments

Internal, or tactical, comments occur within the body of a function definition and are used to convey special information of use to someone actively reading the code. These comments are never special Doxygen comments.

6.5.2.1 For internal code comments, use C++ style (//) comment lines.

6.5.2.2 Include a blank line above a block of comment lines containing text.

These blank lines make comments visually distinct without relying on the editor.

6.5.2.3 A blank line may optionally follow a block of comments.

This should generally indicate that the comment is for a large block of code. No blank line implies that the comment is for the next few lines of code.

6.5.2.4 Comments are allowed on the parameters of a function call.

These comments provide supplemental information about the parameters for that specific function call. The information in parameter comments should not repeat the information in the descriptive text for the `@param` entries in the special documentation block describing the function. Function call parameter comments are never Doxygen comments.

```
Status = TestString (
           String,     // Comment for first parameter
           Index + 3,  // Comment for second parameter
           &Value      // Comment for third parameter
           );
```

6.5.2.5 Indent the comments with the code.

This indentation conveys the scope of the comment, as well as maintaining readability of the code.

```
/// Do nothing, carefully.
void
NoFun (VOID)
{
  // Only process data if mTest is TRUE.
  // This comment block applies to the entire if/else statement.
  if (mTest) {
    // This is an example comment to explain why this behavior
    // is appropriate if mTest is true.
    // This comment block only applies when mTest is true.

    ThisIsTheCode();
  } else {
    // Explain what we do if (mTest) is false.
    // This comment block only applies when mTest is false.

    ThisIsMoreCode();
  }
}
```

6.6 Introducing Doxygen

Doxygen is a tool that allows one to generate documentation directly from a project's source code. There are no complicated, specially formatted documentation tags required for one to get significant benefit from Doxygen. By simply adding an additional asterisk (`` `*) `` or slash (`/`) to existing comments, they become special documentation blocks that Doxygen will add to any generated documentation. A few simple Doxygen commands allow the majority of special documentation tasks to be performed.

Please use Doxygen style comment blocks, at a minimum, when writing new code. If you have time, or are already making modifications, please update existing comments. You will be glad you did.

This document describes the Doxygen elements and style as it applies to EDK II. Other supported Doxygen tag and comment formats are to be eschewed in favor of the style documented here.

6.7 How Doxygen Works

Doxygen understands the `#include` and `#define` preprocessor directives as well as the syntax of C and C++. Thus, Doxygen can generate documentation using just the existing syntactic elements of the source files, or one can supplement the syntactic documentation using Section 6.8 "Special Documentation Blocks". These special documentation blocks are just C or C++ comments with one or more special tags added.

For structures, unions, functions, and global data, you have two documentation options:

- Place a special documentation block in front of the declaration or definition. For enum, structure, and union members, you are also allowed to place documentation directly after a member. See Section 6.8 "Special Documentation Blocks", to learn more about special documentation blocks.
- Place a special documentation block somewhere else (another file or location) and put a structural command in the documentation block. A structural command links a documentation block to a certain entity that can be documented (e.g. a union, structure, function, or file).

The text inside a special documentation block is parsed before it is written to the output files. During parsing, the following steps take place:

1. The special commands inside the documentation block are executed.
2. If a line starts with some whitespace followed by one or more asterisks (*) and then optionally more whitespace, then all leading whitespace and asterisks are removed.
3. All resulting blank lines are treated as paragraph separators. This saves you from placing new-paragraph commands yourself in order to make the generated documentation readable.
4. Links to members are automatically created when certain patterns are found in the text. These patterns include: URLs and email addresses, names of documented classes and files, function and variable names, typedef, enum types, enum values, and defines.
5. HTML tags that are in the documentation are interpreted and converted to the proper equivalents for the selected output type.

6.8 Special Documentation Blocks

A special documentation block is a C or C++ comment block with some additional markings, so that Doxygen knows it is a piece of documentation that needs to end up in the generated documentation.

Normally, Doxygen expects special documentation blocks to immediately precede the syntactic element being documented (Prefix). There is a mechanism to allow documentation to be placed after the element to be documented (Postfix). Prefix style comment blocks are described here, while postfix comment blocks are described in "Putting Documentation after Members" on page 66.

Each code item can have two types of descriptions. Together, a brief description and a detailed description form the documentation: Both are semantically optional. More than one brief or detailed description per code item, however, is not allowed. We require at least one of these descriptions to be present.

A brief description is short, usually a single line. A detailed description provides longer and more detailed documentation.

You may place these descriptions in a comment block in several ways:

- Comment blocks will automatically start a brief description which ends at the first period followed by a space or new line. For example:

```
/** Brief description which ends at this dot.
This sentence begins the detailed description. The detailed description
continues...
**/
```

- The behavior also exists for multi-line special C++ comments:

```
/// Brief description which ends at this dot.
/// This sentence begins the detailed description.
```

It is preferred that any special documentation block longer than two lines use C-style commenting as shown in the first example above. While not required by Doxygen, we require the blank line between the brief description and the detailed description.

Within the body of a C-style comment block, Doxygen will ignore any leading spaces and asterisks, '*'. This means that the first example could be rewritten as follows without any change in the generated documentation.

```
/** Brief description which ends at this dot.
 *
 * This sentence begins the detailed description. The detailed
 * description continues...
**/
```

This type of comment decoration is acceptable because the line of '*' along the left make it easier to determine the extent of the comment.

As you can see, Doxygen is quite flexible. For more information, see the full Doxygen documentation.

6.9 Putting Documentation after Members

For example, to document the members of a `struct`, `union`, or `enum`, and to put the documentation for those members inside the compound, it might be desirable to place the documentation block after the member. Do this by putting an additional `<` marker in the comment block.

For example:

```
int var; ///< Brief description after the member
```

Note: These blocks have the same structure and meaning as special comment blocks. The `<` indicates that the member to be documented is located before the comment.

Note: Only be used these blocks to document members and parameters. Do not use them to document files, classes, unions, structs, groups, namespace, or enums themselves. Furthermore, structural commands (such as `@struct`) are ignored inside these comment blocks.

6.10 Special Commands

All Doxygen commands start with an "at" sign (@). A commands may have one or more arguments. The following typographic conventions are used to identify each arguments range:

- If **<**sharp**>** braces are used, the argument is a single word.
- If **(**round**)** braces are used, the argument extends until the end of the line on which the command was found.
- If **{**curly**}** braces are used, the argument extends until the next paragraph. Paragraphs are delimited by a blank line or by a section indicator.
- If **[**square**]** brackets are used, the argument is optional.

The following sections describe the special Doxygen commands.

6.10.1 @example

Indicates that a comment block contains documentation for a source code example.

6.10.2 @file [ ]

Indicates that a comment block contains documentation for a source or header file with name . If the file name is omitted, Doxygen uses the name of the file that contains the `@file` command. This is the preferred method.

6.10.3 @attention { attention text }

Starts an indented paragraph where you can enter a message about an issue that needs attention.

6.10.4 @param[in,out] { parameter description }

Starts a parameter description for a function parameter named .

6.10.5 @post { description of a postcondition }

Starts an indented paragraph where you can describe a postcondition of an entity.

6.10.6 @pre { description of a precondition }

Starts an indented paragraph where you can describe a precondition of an entity.

6.10.7 @retval { description }

Starts a description of a return value from a function. Include a separate `@retval` for each unique return value.

6.10.8 @return { description of what is returned }

Starts a description for a function's return values when those values aren't easily described by `@retval` commands (that is, the return values are something other than a small set of unique values with discrete meanings).

6.10.9 @sa { references }

Starts a paragraph where you can specify one or more cross-references to functions, structures, variables, files, or URLs.

6.10.10 @since { text }

This tag can be used to specify when (version or time) an entity is available.

6.10.11 @test { paragraph describing a test case }

Starts a paragraph where you can describe a test case. The description will also add the test case to a separate test list. The two instances of the description will be cross-referenced. Each test case in the test list will be preceded by a header that indicates the origin of the test case.

6.10.12 @todo { paragraph describing what is to be done }

Starts a paragraph where a TODO item is described. The description will also add an item to a separate TODO list. The two instances of the description will be cross-referenced. Each item in the TODO list will be preceded by a header that indicates the origin of the item.

6.10.13 HTML Commands

For greater control over the format of generated documentation, you may add HTML commands to special documentation blocks. "HTML Character Entities" below lists the special HTML character entities that Doxygen recognizes.

"HTML Commands" lists the HTML commands that you may use inside the documentation.

Finally, to put invisible comments inside comment blocks, you may use HTML style comments:

```
/** <!-- This is a comment within a comment block --> Visible text */
```

Table 11 HTML Character Entities

Entity	Description
`&copy;`	Copyright symbol
`&tm;`	Trade mark symbol
`&reg;`	Registered trade mark symbol
`&lt`	Less-than symbol
`&gt`	Greater-than symbol
`&`	Ampersand
`&apos;`	Single quotation mark (straight)
`&quot;`	Double quotation mark (straight)
`&lsquo;`	Left single quotation mark
`&rsquo;`	Right single quotation mark
`&ldquo;`	Left double quotation mark
`&rdquo;`	Right double quotation mark
`&ndash;`	En-dash (for numeric ranges, e.g. 2-8)
`&mdash;`	Em-dash (for parenthetical punctuation-like this)
`&?uml;`	Where ? is one of {A,E,I,O,U,Y,a,e,i,o,u,y}, writes a character with a diaeresis accent (ä).
`&?acute;`	Where ? is one of {A,E,I,O,U,Y,a,e,i,o,u,y}, writes a character with an acute accent (á).
`&?grave;`	Where ? is one of {A,E,I,O,U,a,e,i,o,u,y}, writes a character with a grave accent (à).
`&?circ;`	Where ? is one of {A,E,I,O,U,a,e,i,o,u,y}, writes a character with a circumflex accent (â).
`&?tilde;`	Where ? is one of {A,N,O,a,n,o}, writes a character with a tilde accent (ã).
`&szlig;`	Sharp s (ß).
`&?cedil;`	Where ? is one of {c,C}, writes a c-cedille (ç).
`&?ring;`	Where ? is one of {a,A}, writes an 'a' with a ring (å).
` `	A non breaking space.

Table 12 HTML Commands

Command	Description
`<A HREF="...">`	Starts an HTML hyper-link.
`<A NAME="...">`	Starts a named anchor.

`</A>`	Ends a link or anchor.
`<B>`	Starts a piece of text displayed in a bold font.
`</B>`	Ends a `<B>` section.
`<BODY>`	Does not generate any output.
`</BODY>`	Does not generate any output.
` `	Forces a line break.
`<CENTER>`	Starts a section of centered text.
`</CENTER>`	Ends a section of centered text.
`<CAPTION>`	Starts a caption. Use within a table only.
`</CAPTION>`	Ends a caption. Use within a table only.
`<CODE>`	Starts a piece of text displayed in a typewriter font.
`</CODE>`	End a `<CODE>` section.
`<DD>`	Starts an item description.
`<DFN>`	Starts a piece of text displayed in a typewriter font.
`</DFN>`	Ends a `<DFN>` section.
`<DIV>`	Starts a section with a specific style
`</DIV>`	Ends a section with a specific style
`<DL>`	Starts a description list.
`</DL>`	Ends a description list.
`<DT>`	Starts an item title.
`</DT>`	Ends an item title.
`<EM>`	Starts a piece of text displayed in an italic font.
`</EM>`	Ends a `<EM>` section.
`<FORM>`	Does not generate any output.
`</FORM>`	Does not generate any output.
`<HR>`	Writes a horizontal rule.
`<H1>`	Starts an unnumbered section.
`</H1>`	Ends an unnumbered section.
`<H2>`	Starts an unnumbered subsection.
`</H2>`	Ends an unnumbered subsection.
`<H3>`	Starts an unnumbered sub subsection.

`</H3>`	Ends an unnumbered sub subsection.
`<I>`	Starts a piece of text displayed in an italic font.
`<INPUT>`	Does not generate any output.
`</I>`	Ends a `<I>` section.
`<IMG>`	This command is written with attributes to the HTML output.
`<LI>`	Starts a new list item.
`</LI>`	Ends a list item.
`<META>`	Does not generate any output.
`<MULTICOL>`	Ignored by Doxygen.
`</MUTLICOL>`	Ignored by Doxygen.
`<OL>`	Starts a numbered item list.
`</OL>`	Ends a numbered item list.
`<P>`	Starts a new paragraph.
`</P>`	Ends a paragraph.
`<PRE>`	Starts a pre-formatted fragment.
`</PRE>`	Ends a pre-formatted fragment.
`<SMALL>`	Starts a section of text displayed in a smaller font.
`</SMALL>`	Ends a `<SMALL>` section.
`<SPAN>`	Starts an inline text fragment with a specific style.
`</SPAN>`	Ends an inline text fragment with a specific style.
`<STRONG>`	Starts a section of bold text.
`</STRONG>`	Ends a section of bold text.
`<SUB>`	Starts a piece of text displayed in subscript.
`</SUB>`	Ends a `<SUB>` section.
`<SUP>`	Starts a piece of text displayed in superscript.
`</SUP>`	Ends a `</SUP>` section.
`<TABLE>`	Starts a table.
`</TABLE>`	Ends a table.
`<TD>`	Starts a new table data element.
`</TD>`	Ends a table data element.
`<TR>`	Starts a new table row.
`</TR>`	Ends a table row.

`<TT>`	Starts a piece of text displayed in a typewriter font.
`</TT>`	Ends a `<TT>` section.
`<KBD>`	Starts a piece of text displayed in a typewriter font.
`</KBD>`	Ends a `<KBD>` section.
`<UL>`	Starts an unnumbered item list.
`</UL>`	Ends an unnumbered item list.
`<VAR>`	Starts a piece of text displayed in an italic font.
`</VAR>`	Ends a `</VAR>` section.

APPENDIX A COMMON EXAMPLES

File Heading

```
/** @file
  Brief description of file's purpose.

  Detailed description of file's purpose.

  Copyright (c) 2006 - 2014, Acme Corporation. All rights reserved.<BR>
  SPDX-License-Identifier: BSD-2-Clause-Patent

  @par Specification Reference:
  - UEFI 2.3, Chapter 9, Device Path Protocol
  - PI 1.1, Chapter 10, Boot Paths
**/
#ifndef FOO_BAR_H_
#define FOO_BAR_H_

// Body of the file goes here

#endif // FOO_BAR_H_
```

Functioon Declarations

```
/** Brief description of this function's purpose.

  Follow it immediately with the detailed description.

  @param[in]       Arg1  Description of Arg1.
  @param[in]       Arg2  Description of Arg2 This is complicated and requires
                         multiple lines to describe.
  @param[out]      Arg3  Description of Arg3.
  @param[in, out]  Arg4  Description of Arg4.

  @retval  VAL_ONE  Description of what VAL_ONE signifies.
  @retval  OTHER    This is the only other return value. If there were other
                    return values, they would be listed.

**/
EFI_STATUS
EFIAPI
FooBar (
  IN     UINTN  Arg1,
  IN     UINTN  Arg2, OPTIONAL
     OUT UINTN  *Arg3,
  IN OUT UINTN  *Arg4
  );
```

Type Declarations

```
/// Brief description of this enum.
/// Detailed description if justified.
typedef enum {
  EnumMenberOne,  ///< First member description.
  EnumMemberTwo,  ///< Second member description.
  EnumMemberMax   ///< Number of members in this enum.
} ENUMERATE_TYPE;

/// Structure without forward reference.
typedef struct {
  UINT32                   Signature;  ///< Signature description.
  EFI_HANDLE               Handle;     ///< Handle description.
  EFI_PROD_PROT1_PROTOCOL  ProdProt1;  ///< ProdProt1 description.
  EFI_PROD_PROT2_PROTOCOL  ProdProt2;  ///< ProdProt2 description.
} DRIVER_NAME_INSTANCE;

/// Self referential Structure.
typedef struct EFI_CPU_IO_PROTO {
  struct EFI_CPU_IO_PROTO     *Mem;
  EFI_CPU_IO_PROTOCOL_ACCESS  Io;
} EFI_CPU_IO_PROTOCOL;

/// Forward reference
typedef struct StructTag MyStruct;

/// Forward reference target
struct StructTag {
  INT32 First;
  INT32 Second;
};
```

Function Calling

```
Status = TestString ();
Status = TestString (String, Index + 3, &Value);
Status = TestString (
           String,
           Index + 3,
           &Value
           );
```

Control Statements

```
if (Test && !Test2) {
  // This is an example comment to explain why this behavior
  // is appropriate.
  IamTheCode ();
} else if (Test2) {
  // This is an example comment to explain why this behavior
  // is appropriate.
  IamTheCode ();
} else {
  // This is an example comment to explain why this behavior
  // is appropriate.
  IamTheCode ();
}

while (TRUE) {
  IamTheCode ();
}

do {
  IamTheCode ();
} while (TRUE);

for (Index = 0; Index < MAX_INDEX; Index++) {
  IamTheCode (Index);
}

switch (Variable) {
case 1:
  IamTheCode ();
  break;

case 2:
  IamTheCode ();
  break;

default:
  IamTheCode ();
  break;
};
```

APPENDIX B RESERVED IDENTIFIERS

`__bool_true_false_are_defined`		`_Complex_I`	`_Exit`
`_IOFBF`	`_IOLBF`	`_IONBF`	`abs`
`abort`	`acos`	`acosf`	`acosh`
`acoshf`	`acoshl`	`acosl`	`and`
`and_eq`	`asctime`	`asin`	`asinf`
`asinh`	`asinhf`	`asinhl`	`asinl`
`assert`	`atan`	`atan2`	`atan2f`
`atan2l`	`atanf`	`atanh`	`atanhf`
`atanhl`	`atanl`	`atexit`	`atof`
`atoi`	`atol`	`atoll`	`BUFSIZ`
`bitand`	`bitor`	`bool`	`bsearch`
`btowc`	`cabs`	`cabsf`	`cabsl`
`cacos`	`cacosf`	`cacosh`	`cacosl`
`cacoshf`	`cacoshl`	`calloc`	`carg`
`cargf`	`cargl`	`casin`	`casinf`
`casinl`	`casinh`	`casinhf`	`casinhl`
`catan`	`catanf`	`catanl`	`catanh`
`catanhf`	`catanhl`	`cbrt`	`cbrtf`
`cbrtl`	`ccos`	`ccosf`	`ccosh`
`ccoshf`	`ccoshl`	`ccosl`	`ceil`
`ceilf`	`ceill`	`cexp`	`cexpf`
`cexpl`	`CHAR_BIT`	`CHAR_MIN`	`CHAR_MAX`
`cimag`	`cimagf`	`cimagl`	`clearerr`
`clock`	`clock_t`	`CLOCKS_PER_SEC`	`clog`
`clogf`	`clogl`	`compl`	`complex`
`conj`	`conjf`	`conjl`	`copysign`
`copysignf`	`copysignl`	`cos`	`cosf`
`cosl`	`cosh`	`coshf`	`coshl`
`cpow`	`cpowf`	`cpowl`	`cproj`
`cprojf`	`cprojl`	`creal`	`crealf`
`creall`	`csin`	`csinf`	`csinh`
`csinhf`	`csinhl`	`csinl`	`csqrt`
`csqrtf`	`csqrtl`	`ctan`	`ctanf`

ctanl	ctanh	ctanhf	ctanhl
ctime	CX_LIMITED_RANGE	DBL_DIG	DBL_EPSILON
DBL_MAX	DBL_MANT_DIG	DBL_MAX_10_EXP	DBL_MAX_EXP
DBL_MIN	DBL_MIN_10_EXP	DBL_MIN_EXP	DECIMAL_DIG
difftime	div	div_t	double_t
EDOM	EILSEQ	EOF	ERANGE
erf	erff	erfl	erfc
erfcf	erfcl	errno	exit
EXIT_FAILURE	EXIT_SUCCESS	exp	expf
expl	exp2	exp2f	exp2l
expm1	expm1f	expm1l	fabs
fabsf	fabsl	false	fclose
fdim	fdimf	fdiml	FE_ALL_EXCEP
FE_DFL_ENV	FE_DIVBYZERO	FE_DOWNWARD	FE_INEXACT
FE_INVALID	FE_OVERFLOW	FE_TONEAREST	FE_TOWARDZER
FE_UNDERFLOW	FE_UPWARD	feclearexcept	fegetenv
fegetexceptflag	feholdexcept	fegetround	feof
FENV_ACCESS	fenv_t	feraiseexcept	ferror
fesetenv	fesetexceptflag	fesetround	fetestexcept
feupdateenv	fexcept_t	fflush	fgetc
fgetpos	fgetwc	fgetws	fgets
FILE	FILENAME_MAX	float_t	floor
floorf	floorl	FLT_DIG	FLT_EPSILON
FLT_EVAL_METHOD	FLT_MANT_DIG	FLT_MAX	FLT_MAX_10_E
FLT_MAX_EXP	FLT_MIN	FLT_MIN_10_EXP	FLT_MIN_EXP
FLT_RADIX	FLT_ROUNDS	fma	fmaf
fmax	fmaxf	fmaxl	fmin
fminf	fminl	fmod	fmodf
fmodl	fopen	FOPEN_MAX	FP_CONTRACT
FP_FAST_FMA	FP_FAST_FMAF	FP_FAST_FMAL	FP_ILOGB0
FP_ILOGBNAN	FP_INFINITE	FP_NAN	FP_NORMAL
FP_SUBNORMAL	FP_ZERO	fpclassify	fputc
fputs	fpos_t	fprintf	fputwc
fputws	fread	free	freopen
frexp	frexpf	frexpl	fscanf
fseek	fsetpos	ftell	fwide
fwprintf	fwrite	fwscanf	getc

getchar	getenv	gets	getwc
getwchar	gmtime	HUGE_VAL	HUGE_VALF
HUGE_VALL	hypot	hypotf	hypotl
I	ilogb	ilogbf	ilogbl
imaginary	imaxdiv_t	imaxabs	imaxdiv
INFINITY	INT_FASTN_MIN	INT_FASTN_MAX	int_fastN_t
INT_LEASTN_MIN	INT_LEASTN_MAX	int_leastN_t	INT_MAX
INT_MIN	INTMAX_C	INTMAX_MAX	INTMAX_MIN
intmax_t	INTN_C	INTN_MIN	INTN_MAX
intN_t	intptr_t	INTPTR_MIN	INTPTR_MAX
isalnum	isalpha	isblank	iscntrl
isdigit	isfinite	isgraph	isgreater
isgreaterequal	isinf	isless	islessequal
islessgreater	islower	isnan	isnormal
isprint	ispunct	isspace	isunordered
isupper	iswalnum	iswalpha	iswblank
iswcntrl	iswctype	iswdigit	iswgraph
iswlower	iswprint	iswpunct	iswspace
iswupper	iswxdigit	isxdigit	jmp_buf
L_tmpnam	labs	LC_ALL	LC_COLLATE
LC_CTYPE	LC_MONETARY	LC_NUMERIC	LC_TIME
LDBL_DIG	LDBL_EPSILON	LDBL_MANT_DIG	LDBL_MAX
LDBL_MAX_EXP	LDBL_MAX_10_EXP	LDBL_MIN	LDBL_MIN_EXP
LDBL_MIN_10_EXP	ldexp	ldexpf	ldexpl
ldiv	ldiv_t	lgamma	lgammaf
lgammal	llabs	lldiv	lldiv_t
LLONG_MAX	LLONG_MIN	llrint	llrintf
llrintl	llround	llroundf	llroundl
localeconv	localtime	log	logf
logl	log10	log10f	log10l
log1p	log1pf	log1pl	log2
log2f	log2l	logb	logbf
logbl	LONG_MAX	LONG_MIN	longjmp
lrint	lrintf	lrintl	lround
lroundf	lroundl	mal	malloc
MATH_ERREXCEPT	math_errhandling	MATH_ERRNO	MB_CUR_MAX
MB_LEN_MAX	mblen	mbrlen	mbrtowc

mbsinit	mbstate_t	mbstowcs	mbsrtowcs
mbtowc	memchr	memcmp	memcpy
memmove	memset	mktime	modf
modff	modfl	NAN	nan
nanf	nanl	NDEBUG	nearbyint
nearbyintf	nearbyintl	nextafter	nextafterf
nextafterl	nexttoward	nexttowardf	nexttowardl
not	not_eq	NULL	or
or_eq	offsetof	perror	pow
powf	powl	PRIdN	PRIiN
PRIoN	PRIuN	PRIXN	PRIxN
PRIdLEASTN	PRIdFASTN	PRIdMAX	PRIdPTR
PRIiLEASTN	PRIiFASTN	PRIiMAX	PRIiPTR
printf	PRIoLEASTN	PRIoFASTN	PRIoMAX
PRIoPTR	PRIuLEASTN	PRIuFASTN	PRIuMAX
PRIuPTR	PRIXLEASTN	PRIxLEASTN	PRIXFASTN
PRIxFASTN	PRIXMAX	PRIxMAX	PRIXPTR
PRIxPTR	PTRDIFF_MAX	PTRDIFF_MIN	ptrdiff_t
putc	putchar	puts	putwc
putwchar	qsort	raise	rand
RAND_MAX	realloc	remainder	remainderf
remainderl	remove	remquo	remquof
remquol	rename	rewind	rint
rintf	rintl	round	roundf
roundl	scalbn	scalbnf	scalbnl
scalbln	scalblnf	scalblnl	scanf
SCHAR_MAX	SCHAR_MIN	SCNdN	SCNdFASTN
SCNdLEASTN	SCNdMAX	SCNdPTR	SCNiN
SCNiFASTN	SCNiLEASTN	SCNiMAX	SCNiPTR
SCNoN	SCNoFASTN	SCNoLEASTN	SCNoMAX
SCNoPTR	SCNuN	SCNuFASTN	SCNuLEASTN
SCNuMAX	SCNuPTR	SCNxN	SCNxFASTN
SCNxLEASTN	SCNxMAX	SCNxPTR	SEEK_CUR
SEEK_END	SEEK_SET	setbuf	setjmp
setlocale	setvbuf	SHRT_MAX	SHRT_MIN
SIG_ATOMIC_MAX	SIG_ATOMIC_MIN	sig_atomic_t	SIG_DFL
SIG_ERR	SIG_IGN	SIGABRT	SIGFPE

SIGILL	SIGINT	signal	signbit
SIGSEGV	SIGTERM	sin	sinf
sinl	sinh	sinhf	sinhl
SIZE_MAX	size_t	snprintf	sprintf
sqrt	sqrtf	sqrtl	srand
sscanf	stderr	stdin	stdout
strcat	strchr	strcmp	strcoll
strcpy	strcspn	strerror	strftime
strlen	strncat	strncmp	strncpy
strpbrk	strrchr	strspn	strstr
strtod	strtof	strtoimax	strtok
strtol	strtold	strtoll	strtoul
strtoull	strtoumax	strxfrm	swprintf
swscanf	system	tan	tanf
tanl	tanh	tanhf	tanhl
tgamma	tgammaf	tgammal	time
time_t	TMP_MAX	tmpfile	tmpnam
tolower	toupper	towlower	towupper
towctrans	true	trunc	truncf
truncl	UCHAR_MAX	UINT_FASTN_MAX	uint_fastN_t
UINT_LEASTN_MAX	uint_leastN_t	UINT_MAX	UINTMAX_C
UINTMAX_MAX	uintmax_t	UINTN_C	UINTN_MAX
uintN_t	UINTPTR_MAX	uintptr_t	ULLONG_MAX
ULONG_MAX	ungetc	ungetwc	USHRT_MAX
va_arg	va_copy	va_end	va_list
va_start	vfprintf	vfscanf	vfwprintf
vfwscanf	vprintf	vscanf	vsnprintf
vsprintf	vsscanf	vswprintf	vswscanf
vwprintf	vwscanf	WCHAR_MAX	WCHAR_MIN
wchar_t	wcscat	wcschr	wcscmp
wcscoll	wcscpy	wcscspn	wcsftime
wcslen	wcsncat	wcsncmp	wcsncpy
wcspbrk	wcsrchr	wcsrtombs	wcsspn
wcsstr	wcstod	wcstof	wcstoimax
wcstok	wcstol	wcstold	wcstoll
wcrtomb	wcstombs	wcstoul	wcstoull
wcstoumax	wcsxfrm	wctob	wctomb

wctrans	wctrans_t	wctype	wctype_t
WEOF	WINT_MAX	WINT_MIN	wint_t
wmemchr	wmemcmp	wmemcpy	wmemmove
wmemset	wprintf	wscanf	xor
xor_eq	_Imaginary_I		

APPENDIX C OPTIMIZATION AND PERFORMANCE

You might be tempted to use all sorts of clever techniques to optimize the code. However, this optimization comes at a penalty.

Rob Pike provides these comments on program complexity, https://www.lysator.liu.se/c/pikestyle.html, *Notes on Programming in C*.

"Most programs are too complicated - that is, more complex than they need to be to solve their problems efficiently . . . But programs are often complicated at the microscopic level . . ."

The following are also condensed from Mr. Pike's 1989 *Notes on Programming in C*

Rule 1. Don't guess at optimization

- Prove the bottleneck location, then put in a speed hack.

Rule 2. Measure before tuning for speed

- Be judicious. Only tune for speed if absolutely necessary.

Rule 3. Use simple algorithms

- Fancy algorithms are slow.

Rule 4. Use simple data structures.

Pike believes that these data structures are almost always enough:

- array
- linked list
- hash table
- binary tree

Rule 5. Data structures are central to programming; algorithms are not.