A Software Audit Report for the

Princeton Shape Benchmark
Computer Software Configuration Item (CSCI)

Submitted to

Princeton University

`http://shape.cs.princeton.edu`

by

AuditSoft, Inc.
860 River Oaks Drive
Cropwell, Alabama 35054
`http://www.AuditSoft.com`

February 14, 2014

A static audit including findings of potential issues and suggested improvements

Contents

1 AUDITSOFT, INC. SERVICES

Our mission is to provide the business community with the best software selection and improvement services concerning firmware and software projects. We do this through the use of our excellent staff, facilities, and resources. AuditSoft, Inc. is primarily in business to provide software auditing, assurance, verification and validation (V&V) services. Our services are available for software under development, software already installed, and software under consideration.

- Requirements Documentation and Review: Documentation of requirements is facilitated by discovery and documentation methods that yield categorized, verifiable, and traceable requirements. AuditSoft's trained experts can serve as facilitators in the discovery and documentation process. We can support both conventional formal documentation process, or newer Agile Modeling, using techniques that result in a consensus requirements document.

- Development Auditing: Often maintainability and other quality issues may be pushed so far down the priority stack as to not occur. AuditSoft offers non-invasive techniques to evaluate quality of software under development and development processes. AuditSoft can report on its findings and, if appropriate, make recommendations on how risks may be ameliorated.

- Design Review: Once your software development lifecycle has begun, AuditSoft can assist in moderating your Software Design Reviews. It is likely that we can contribute in identifying improvement opportunities.

- Verification and Validation: AuditSoft can assist in writing Acceptance Test Plans, and/or Qualification Test Plans based on your requirements specifications. For mission critical software, AuditSoft can engineer such tests, and identify requirements that are assured by tests.

- Programming Style and Technique Training: AuditSoft has a customizable style and technique document that can be tailored for your developers to aid in long-term software maintenance. AuditSoft's style and technique requirements are few, easily justifiable, and have little overhead.

- Quality and Safety Assurance: AuditSoft can assist in establishment of Software Quality and Safety Assurance not directly addressed above; such as configuration management method, defect handling and corrective action, customer satisfaction assurance, definitions and terms.

2 HOW TO READ THE REPORT

The following report is a result of AuditSoft's non-invasive techniques to evaluate quality of sources. The service is called StaticAudit. Every line of code is automatically analyzed; AuditSoft personnel review items generated by the automatic analysis. Through careful examination AuditSoft identifies the cause of each instance and prepares recommendations. Security and safety implications are tagged to separate them from other quality issues.

The State of the Software section provides an overview of progress on development. At different stages of software development, different types of review are indicated. This section may give insight into status of development processes.

During the initial report analysis, AuditSoft's Code Assessment Team scrutinizes findings and addresses items that may negatively affect security, safety, and/or quality analysis. Each instance is handled independently, referencing the specific line of code or function in question, explaining the reason for concern, and providing resolution recommendations. Additional instructions may follow the recommendation if such recommendation may introduce additional errors. This process is repeated for each quality notice generated.

After each cycle of review, the Team reviews the rules used to assess software. Rules that generate unnecessary notices may be disabled. If additional static checking is defined, additional rules may be added.

This report is an example of a first cycle of generating these reports for this project. We expect that as the project becomes cleaner the content of the report will change to more follow-on analysis of notices, and that fewer requests for program modifications will occur.

3 SOURCE CODE METRICS

StaticAudit measures source code for quantity and quality metrics. It is designed to process source code from `C`, ANSI `C`, `C++`, ANSI `C++`, Java 2.0, and `C#`. StaticAudit expects that each source file be compilable. StaticAudit does not read preprocessor directives; it analyzes each source file independently.

Size metrics are described by counting "lines of code". Source code quality analysis is measured by semantic analysis of the code beyond the syntax rules of the language.

Various key words and statements are provided for code analysis. Readability and code quality can easily be determined by analyzing the quality notices, comment percentage, white space content and key words used within the source code.

4 DEFINITIONS

4.1 Words of Art

In this report many *words of art* are used. The reader is expected to have an understanding of many of them. The following group of definitions are ones that we believe are necessary to understand the report, but are not commonly known.

- Comment Lines: lines that contain a comment. This includes lines that have both code and comments on the same physical line.
- Computer Software Component (**CSC**, pronounced "C S C"): A functionally or logically distinct part of a computer software configuration item.
- Computer Software Configuration Item (**CSCI**, pronounced "C S C I"): An aggregation of software that is designated for configuration management and treated as a single entity in the configuration management process.
- Cyclomatic Complexity: (**CycloCmp**) is the McCabe Cyclomatic complexity for a function. The McCabe cyclomatic complexity has become the defacto industry standard for measuring the structural complexity of a function. Cyclomatic complexity as defined by McCabe is the number of logical pathways through a function. This metric can either be determined by counting the regions, nodes and edges or number of predicate nodes (branching points) with a flow graph. The following equations defined McCabe Cyclomatic Complexity: "Software Engineering, A Practioners Approach", Roger S. Pressman, McGraw Hill. The number of regions in a flow graph. $V(G) = E - N + 2$, where E are the edges and N are the nodes. $V(G) = P + 1$, where P are the predicate nodes.
- Effective Lines of Code (**eLOC**, prounced "e loc"): An effective line of code is a line of code that is not a stand-alone brace `{}`, or parenthesis `()`. This peculiar metric more accurately defines the quantity of work performed in a source code module. We have found that eLOC is the metric we intuitively estimate as experienced software engineers.
- Function Point: a unit of measurement to express the amount of business functionality an information system provides to a user. Function points are an ISO recognized software metric to size an information system based on the functionality that is perceived by the user of the information system, independent of the technology used to implement the information system.
- Interface Complexity: (**InterCmp**) is defined by the package as the number of input parameters to a function plus the number of return states from that function. Class interface complexity is the sum of all function interface complexity metrics within that class.
- Lines of Code (**LOC**, pronounced "L O C"): A line of code is defined as a line within a source file that is not a comment or blank line. Lines that contain both source code and comments are counted as an instance of each.
- Logical Lines of Code (**lLOC**, pronounced "el loc"): Logical Lines of code are defined as code statements or those lines that end in a semicolon. For example, a "for" loop structure accounts for one lLOC. Logical lines within a source file can exceed the physical lines within a file code and comments occur on the same physical line. The sum of code, blanks and comments equates to the logical lines. Metrics programs that show code,

comments and blank lines equal to the physical lines do not account for a second instance of code or comments.

```
Source code line              LOC   eLOC  lLOC  Comment  Blank
-------------------------------------------------------------
if (x<10)   // test  range     x      x           x
{                              x
  // update y coordinate                         x
                                                          x
   y = x + 1;                  x      x     x
}                              x
-------------------------------------------------------------
```

4.2 Quality Notice Definitions

1 Physical line length > 80 characters.

2 Function name length > 32 characters.

3 Ellipsis '...' are identified as function parameters.

4 Assignment '=' within 'if' statement.

5 Assignment '=' within 'while' statement.

6 Pre-decrement '--' operator identified.

7 Pre-increment '++' operator identified.

8 'realloc' function identified.

9 'GOTO' keyword identified.

10 Non-ANSI function prototype is identified within the code.

11 Open and closed brackets '[ ]' are not balanced within a file.

12 Open and closed parenthesis '()' are not balance within a file.

13 'Switch' statement does not have a 'default'.

14 'Case' conditions do not equal 'break'.

15 A friend class is identified within the code.

16 Function/class/struct/interface white space < 5.0%.

17 Function comment content less than 5.0%.

18 Function eLOC >maximum 200 eLOC.

19 File white space < 5.0%.

20 File comment content < 5.0%.

22 'if', 'else', 'for', or 'while' not bounded by scope.

23 '?' ternary operator identified.

24 ANSI `C++` keyword identified within C file.

26 'Void *' generic type identified.

27 Number of function return points > 2.

28 Cyclomatic complexity > 10.

29 Number of function parameters > 6.

30 TAB character has been identified.

31 Class/struct comments are < 5.0%.

32 The construct 'using namespace' has been identified.

33 A class/struct definition is identified within a function definition.

34 Class contains a pointer.

35 A class definition contains public data.

36 A class definition contains protected data.

37 A base class, with virtual functions, does not contain a virtual destructor.

38 Exception handling is present within a function.

39 The number of class/struct methods > 100 (public, protected and private).

40 The depth of the inheritance tree > 4.

41 The number of direct derived classes > 10.

42 Multiple inheritance has been identified.

43 The use of 'continue' in logical structures.

44 Keyword 'break' identified outside a 'switch' structure.

45 File does not have equal counts of new and delete key word counts.

46 Function/class blank line content less < 1.0%.

47 File Blank Line content < 5.0%.

48 Function lLOC <= 0, non-operational function.

49 Function appears to have null or blank parameters.

50 A variable is assigned to a literal value.

51 There is no comment before a function block.

52 There is no comment before a class block.

53 There is no comment before a struct block.

54 There is no comment before an interface block.

55 Scope exceeds the specified maximum in the configuration file.

56 Sequential break statements are identified.

102 Under `C++` the use of malloc and sister routines is disparaged.

104 A line containing just a ';' has been identified.

105 A symbolic constant using `#define` has been identified.

107 A double ;; has been identified.

109 Double pointer indirection identified.

116 Pointer variable uninitialized.

117 C style macro identified.

118 Keyword `struct` identified in a `C++` module.

119 Return is not a function.

120 'const' followed by a lowercase variable.

121 Class name not proper cased.

122 A variable name with a capital letter has been found.

124 M2 Test Message.

125 A data member in the header is not of the form `m_*`.

201 A 'Rights in Software Product' message not found in file.

202 The construct 'using namespace' has been identified. (overrideable version of 32)

203 C style macro identified (overrideable version of 117)

5 THE BENCHMARK

The Princeton Shape Benchmark (PSB) provides a repository of 3D models and software tools for evaluating shape-based retrieval and analysis algorithms. The motivation is to promote the use of standardized data sets and evaluation methods for research in matching, classification, clustering, and recognition of 3D models. Researchers are encouraged to use these resources to produce comparisons of competing algorithms in future publications.

The benchmark contains a database of 3D polygonal models collected from the World Wide Web. The files are located in the "db" subdirectory. For each 3D model, there is an Object File Format (.off) file with the polygonal geometry of the model, a model information file (e.g., the URL from whence it came), and a JPEG image file with a thumbnail view of the model. Version 1 of the benchmark contains 1,814 models.

For ease of parsing, all models have been converted into the Geometry Center's Object File Format (.off). Documentation about the .off format can be found in `documentation/off_format.html`, and sample source code for parsing .off files can be found in `util/offstats`.

To locate models, each model has a unique identifier referred to as the "model id," a positive integer. Because of the large number of models, groups of models are split into subdirectories to limit the number of files in any single directory. For a given model id, its folder is in `db/<folderId>/m<modelId>/` , where `<folderId>` is calculated as follows `<folderId> = floor(<modelId> / 100)`. As an example, files for model 812 are in `db/8/m812/`, and files for model 7 are in `db/0/m7/`. Note that there is no model 762.

6 STATE OF THE SOFTWARE

The benchmark set of models has been split into a training database and a test database. Algorithms should be trained on the training database (without influence of the test database). Then, after all exploration has been completed and all algorithmic parameters have been frozen, results should be reported for the test database. In Version 1, the training database contains 907 models, and the test database contains 907 models.

In Version 1, we provide a "base" classification that reflects primarily the function of each object and secondarily its form. The base training classification contains 90 classes, and the base test classification contains 92 classes. We expect to provide alternate classifications in the near future, and we encourage other researchers to submit interesting classifications for inclusion in future versions of the benchmark.

We provide free source code to help parsing and working with the benchmark files. For instance, we provide sample code for:

- parsing Object File Format (.off) files,
- parsing classification (.cla) files,
- visualizing .off files in an interactive OpenGL viewer,
- visualizing classifications with interactive Web pages,
- creating plots of precision and recall for a shape retrieval, and
- analyzing the retrieval results by a variety of statistics.

Source code and Windows executables can be found in the util subdirectory.

7 METRICS

7.1 Computer Generated Metrics

Date generated: Friday, February 14, 2014 Reviewer: Dr. Leonard J. Jowers

```
                        ~~ Total Project Summary ~~

LOC 334      eLOC 284      lLOC 212      Comment 69       Lines      520
Average per File, metric/3 files
LOC 111      eLOC 94       lLOC 70       Comment 23       Lines      173
Function Points         FP(LOC) 6.0     FP(eLOC) 5.1    FP(lLOC)     3.8

------------------------------------------------------------------------
                     ~~ Project Functional Analysis ~~

Total Functions .......:         11  Total Physical Lines ..:        364
Total LOC .............:        282  Total Function Pts LOC :        5.3
Total eLOC ............:        232  Total Function Pts eLOC:        4.4
Total lLOC.............:        186  Total Function Pts lLOC:        3.5
Total Cyclomatic Comp. :         64  Total Interface Comp. .:         41
Total Parameters ......:         23  Total Return Points ...:         18
Total Comment Lines ...:         24  Total Blank Lines .....:         74
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:      33.09
Avg LOC ...............:      25.64  Avg eLOC ..............:      21.09
Avg lLOC ..............:      16.91  Avg Cyclomatic Comp. ..:       5.82
Avg Interface Comp. ...:       3.73  Avg Parameters ........:       2.09
Avg Return Points .....:       1.64  Avg Comment Lines .....:       2.18
       ------     -----     -----     ------     ------     -----
Max LOC ...............:         75
Max eLOC ..............:         60  Max lLOC ..............:         45
Max Cyclomatic Comp. ..:         25  Max Interface Comp. ...:          6
Max Parameters ........:          5  Max Return Points .....:          3
Max Comment Lines .....:          7  Max Total Lines .......:        101
       ------     -----     -----     ------     ------     -----
Min LOC ...............:         10
Min eLOC ..............:          7  Min lLOC ..............:          5
Min Cyclomatic Comp. ..:          1  Min Interface Comp. ...:          1
Min Parameters ........:          0  Min Return Points .....:          1
Min Comment Lines .....:          0  Min Total Lines .......:         13

------------------------------------------------------------------------
```

```
                    ~~ Project Quality Profile ~~

Type  Count Percent  Quality Notice
------------------------------------------------------------------------

1        31    9.23  Physical line length > 80 characters
3         2    0.60  Ellipsis '...' are identified as function parameters
7        12    3.57  Pre-increment operator '++' identified
17        7    2.08  Function comment content less than 5.0%
22        5    1.49  if, else, for or while not bound by scope
23        1    0.30  '?' ternary operator identified
27        2    0.60  Number of function return points > 2
28        1    0.30  Cyclomatic complexity > 10
30      225   66.96  TAB character has been identified
43        2    0.60  Keyword 'continue' has been identified
44        1    0.30  Keyword 'break' identified outside a 'switch' structure
49        1    0.30  Function appears to have null or blank parameters
50       22    6.55  Variable assignment to a literal number
51        7    2.08  No comment preceding a function block
53        1    0.30  No comment preceding a struct block
102       8    2.38  Dynamic memory using malloc is not initialized
105       1    0.30  A symbolic constant using #define
107       1    0.30  A double ;; has been identified
109       4    1.19  Double pointer indirection identified
201       2    0.60  A "Rights in Software Product" string was not found
------------------------------------------------------------------------

        336  100.00  Total Quality Notices

                    ~~ Quality Notice Density ~~

Basis: 1000 (K)

Quality Notices/K LOC   =     1006.0 (100.60%)
Quality Notices/K eLOC  =     1183.1 (118.31%)
Quality Notices/K lLOC  =     1584.9 (158.49%)

------------------------------------------------------------------------
```

Filename	Timestamp	LOC	eLOC	Cmm't	Lines
PSBClaParse	1600/12/31 18:00:00	15	15	17	47
bestMatch.c	1600/12/31 18:00:00	178	152	24	247
PSBClaParse	1600/12/31 18:00:00	141	117	28	226

Table 1: Summary of files included in analysis

7.2 Quality Notices

We have reviewed Quality Notices in the order they appear in the automatically generated report. The notices have been sorted by Quality Notice number to facilitate addressing notices of a particular type.

1 Physical line length > 80 characters.

(13 occurrences). Lines 52, 64, 70, 74, 81, 95, 113, 115, 124, 126, 136, 140, 148. `parseFile` of `PSBClaParse`.
(2 occurrences). Lines 152, 154. `printMainPage` of `bestMatch.c`.
(2 occurrences). Lines 5, 6. `PSBClaParse` outside of any function.
(4 occurrences). Lines 204, 227, 229, 231. `doModelQuery` of `bestMatch.c`.
(5 occurrences). Lines 14, 17, 41, 43, 44. `PSBClaParse` outside of any function.
Line 172. `error` of `PSBClaParse`.
Line 191. `defineCategory` of `PSBClaParse`.
Line 207. `isCategoryDefined` of `PSBClaParse`.
Line 217. `createFullName` of `PSBClaParse`.
Line 29. the structure `PSBCategory` of `PSBClaParse`.

3 Ellipsis '...' are identified as function parameters.

Line 153. `parseFile` of `PSBClaParse`.
Line 39. `PSBClaParse` outside of any function.

7 Pre-increment '++' operator identified.

(2 occurrences). Lines 121, 131. `parseFile` of `PSBClaParse`.
(2 occurrences). Lines 148, 153. `printMainPage` of `bestMatch.c`.
(2 occurrences). Lines 183, 193. `doModelQuery` of `bestMatch.c`.
(4 occurrences). Lines 51, 62, 63, 66. `createModelClassMapping` of `bestMatch.c`.
Line 199. `isCategoryDefined` of `PSBClaParse`.
Line 215. `createFullName` of `PSBClaParse`.

17 Function comment content less than 5.0%.

0.0% in `createFullName` of `PSBClaParse`.
0.0% in `defineCategory` of `PSBClaParse`.
0.0% in `error` of `PSBClaParse`.
0.0% in `isCategoryDefined` of `PSBClaParse`.
0.0% in `main` of `bestMatch.c`.
3.8% in `parseFile` of `PSBClaParse`.
3.8% in `printMainPage` of `bestMatch.c`.

22 'if', 'else', 'for', or 'while' not bounded by scope.

(3 occurrences). Lines 117, 130, 148. `parseFile` of `PSBClaParse`.
Line 151. `printMainPage` of `bestMatch.c`.
Line 200. `isCategoryDefined` of `PSBClaParse`.

23 '?' ternary operator identified.

Line 122. `compareRanks` of `bestMatch.c`.

27 Number of function return points > 2.

Returns are 3 for Line 129. `compareRanks` of `bestMatch.c`.
Returns are 3 for Line 204. `isCategoryDefined` of `PSBClaParse`.

28 Cyclomatic complexity > 10.

CC of 25 for Line 150. `parseFile` of `PSBClaParse`.

30 TAB character has been identified.

(12 occurrences). Lines 31, 32, 33, 33, 34, 34, 35, 37, 38, 39, 40, 41. `main` of `bestMatch.c`.
(156 occurrences). Lines 171, 172, 173, 174, 175, 176, 177, 178, 181, 182, 183, 184, 184, 185, 185, 186, 186, 186, 187, 187, 187, 188, 188, 189, 189, 190, 190, 190, 191, 191, 191, 192, 192, 193, 193, 193, 194, 194, 195, 195, 196, 196, 199, 199, 200, 200, 201, 201, 201, 202, 202, 203, 203, 204, 204, 205, 205, 205, 206, 206, 207, 207, 208, 208, 209, 209, 210, 210, 211, 211, 212, 212, 214, 214, 215, 215, 215, 216, 216, 216, 217, 217, 217, 218, 218, 218, 219, 219, 219, 220, 220, 220, 220, 221, 221, 221, 221, 222, 222, 222, 222, 223, 223, 223, 224, 224, 224, 224, 225, 225, 225, 227, 227, 227, 228, 228, 228, 228, 229, 229, 229, 231, 231, 231, 232, 232, 232, 233, 233, 233, 234, 234, 234, 235, 235, 235, 235, 236, 236, 236, 237, 237, 238, 238, 239, 239, 239, 239, 239, 242, 242, 243, 244, 244, 245, 246. `doModelQuery` of `bestMatch.c`.
(17 occurrences). Lines 92, 93, 99, 99, 100, 100, 100, 101, 101, 101, 102, 102, 103, 103, 104, 104, 106. `readMatrix` of `bestMatch.c`.
(2 occurrences). Lines 124, 127. `compareRanks` of `bestMatch.c`.
(2 occurrences). Lines 76, 77. `createModelClassMapping` of `bestMatch.c`.
(30 occurrences). Lines 49, 50, 51, 52, 52, 53, 56, 57, 58, 59, 61, 62, 63, 63, 64, 64, 64, 65, 65, 65, 66, 66, 66, 67, 67, 67, 68, 69, 70, 71. `createModelClassMapping` of `bestMatch.c`.
(5 occurrences). Lines 138, 154, 154, 155, 156. `printMainPage` of `bestMatch.c`.
Line 168. `printMainPage` of `bestMatch.c`.

43 The use of 'continue' in logical structures.

Line 151. `printMainPage` of `bestMatch.c`.
Line 187. `doModelQuery` of `bestMatch.c`.

44 Keyword 'break' identified outside a 'switch' structure.

Line 91. `parseFile` of `PSBClaParse`.

49 Function appears to have null or blank parameters.

Line 73. `createModelClassMapping` of `bestMatch.c`.

50 A variable is assigned to a literal value.

(2 occurrences). Lines 148, 153. `printMainPage` of `bestMatch.c`.
(3 occurrences). Lines 86, 97, 98. `readMatrix` of `bestMatch.c`.

(4 occurrences). Lines 84, 85, 119, 121. `parseFile` of `PSBClaParse`.
(5 occurrences). Lines 50, 51, 61, 62, 63. `createModelClassMapping` of `bestMatch.c`.
(6 occurrences). Lines 174, 181, 183, 191, 212, 214. `doModelQuery` of `bestMatch.c`.
Line 199. `isCategoryDefined` of `PSBClaParse`.
Line 215. `createFullName` of `PSBClaParse`.

51 There is no comment before a function block.

Line 132. `printMainPage` of `bestMatch.c`.
Line 153. `error` of `PSBClaParse`.
Line 172. `defineCategory` of `PSBClaParse`.
Line 191. `isCategoryDefined` of `PSBClaParse`.
Line 208. `createFullName` of `PSBClaParse`.
Line 30. `main` of `bestMatch.c`.
Line 50. `parseFile` of `PSBClaParse`.

53 There is no comment before a struct block.

Line 22. the structure `PSBCategory` of `PSBClaParse`.

102 Under `C++` the use of malloc and sister routines is disparaged.

(2 occurrences). `createModelClassMapping` of `bestMatch.c`.
(2 occurrences). `defineCategory` of `PSBClaParse`.
(2 occurrences). `parseFile` of `PSBClaParse`.
(2 occurrences). `readMatrix` of `bestMatch.c`.

105 A symbolic constant using `#define` has been identified.

Line: 30 A symbolic constant using `#define` has been identified. `PSBClaParse` outside of any function.

107 A double ;; has been identified.

`doModelQuery` of `bestMatch.c`.

109 Double pointer indirection identified.

`defineCategory` of `PSBClaParse`.
`parseFile` of `PSBClaParse`.
`readMatrix` of `bestMatch.c`.
the structure `Rank` of `bestMatch.c`.

201 A 'Rights in Software Product' message not found in file.

`createFullName` of `PSBClaParse`.
`doModelQuery` of `bestMatch.c`.

8 SUMMARY

Several relaxations of Quality Notice (QN) triggers have been made. The QN "return is not a function" has been disabled (a) because of continued use of parentheses around the return arguments of return statements; (b) because the basis of flagging it is not clear (a knowledgeable programmer will not mistake a return statement for a function call). The trigger on QN 27 "number of return statements" has been upped from 1 to 2 so that the need to address those routines with several return statements is made clearer; however, we recommend that developers be strongly encouraged to have only one return statement per function. QNs that are triggered by a 10% percentage of blank space or comments have been modified to trigger on 5%; on review of source we determine that in many triggered cases source was adequately legible due to structure and variable name choices.

The following notices represent opportunities for errors in the future and need to be modified. They are not considered good practice because they are prone to induce errors in programming and maintenance.

7 Pre-increment '++' operator identified. This can be a problematic construct in compound statements, such as when an increment operation is performed within the conditional of an 'if' statement. This is an unnecessary complication. Post-increment operators are frequently used. When a seldom-used Pre-increment operator is used is may be difficult to recognize its different operation.

23 '?' ternary operator identified: The '?' operator creates the code equivalent of an "if" - "else" construct. However the resultant source is far less readable.

27 Number of function return points > 2: A well-constructed function has one entry point and one exit point. Functions with multiple return points are difficult to debug and maintain.

28 Cyclomatic Complexity: (CycloCmp) is the McCabe Cyclomatic complexity. Industry-wide, cyclomatic complexity greater than 10 is considered risky; greater than 20 is generally considered unacceptable. The McCabe cyclomatic complexity has become the defacto industry standard for measuring the structural complexity of a function. Cyclomatic complexity as defined by McCabe is the number of logical pathways through a function. This metric can either be determined by counting regions, nodes and edges, or number of predicate nodes (branching points) with a flow graph. The following equations defined McCabe Cyclomatic Complexity: "Software Engineering, A Practioners Approach", Roger S. Pressman, McGraw Hill. The number of regions in a flow graph. $V(G) = E - N + 2$, where E are the edges and N are nodes. $V(G) = P + 1$, where P are predicate nodes.

43 Use of 'continue' in logical structures causes a disruption in linear flow of logic. This style of programming can make maintenance and readability difficult.

44 Keyword 'break' identified outside a 'switch' structure: Use of 'break' outside a 'switch' block disrupts the linear logic flow of a function. This style of programming can make code maintenance and readability difficult.

102 Under `C++` use of malloc and sister routines is disparaged. We recommend that alloc type routines be removed and either static structures or new and deleted be used.

107 A double ';;' has been identified: ';;' should not be coded. Where necessary, one should consider a comment that explains what is being intentionally omitted.

109 A double pointer indirection identified: Double pointer indirections are often difficult to understand. If left, they should trigger a review.

The following notices need to be reviewed to determine what risk they represent. It is believed they may carry significant risk to program correctness.

3 Ellipsis '...' are identified as function parameters. Ellipsis create a variable argument list. This type of design is found in C and `C++`. It essentially breaks the type strict nature of `C++` and should be avoided.

The following notices relate to embedded documentation. Though they may not carry significant risk during development, they do represent low maintainability and should be addressed. Attention to one of these may address several.

1 Physical line length > 80 characters. This width exceeds the standard terminal width of 80 characters. Reproducing source code on devices that are limited to 80 columns of text can cause truncation of the line or wrap the line. Wrapped source lines are difficult to read, thus creating weaker peer reviews.

17 Function comment content less than 5.0%: A programmer must supply sufficient comments to enable understanding source. Typically a comment percentage less than 10 percent is considered insufficient. However, content quality is just as important as comment quantity. The reviewer should be able to read the comments and extract the story of the code.

22 if, else, for or while is not bound by scope: Logical blocks should be bound with scope by braces. This will clearly mark boundaries of scope for logical blocks. Many times, code may be added to non-scoped logic blocks; thus, pushing other lines from the active region of the logical construct and giving rise to a logic defect.

30 TAB character has been identified: Tabs embedded into code are device and editor dependent for their space definition and may not display properly. Tab characters create documents that are print and display device dependent. The document may look correct on the screen but it may become unreadable when printed.

49 Function appears to have null or blank parameters. `C` does not support optional parameters. Although `C++` does, allowing optional parameters is not preferred over overloading. Either should be used only after verifying that there is a clear reason why it is necessary. We suggest using explicit parameters for interface clarity.

50 A variable was been identified that is assigned to a literal number. Symbolic constants should be used to enhance maintainability.

51 A function has been identified that does not have a preceding comment. Comments that detail purpose, algorithms, and parameter/return definitions are suggested. We recommend that at a minimum, each function be delimited from the code above by a comment line.

53 A struct has been identified that does not have a preceding comment. Comments that detail purpose, algorithms, and parameter/return definitions are suggested. We recommend that at a minimum, each struct be preceded by an explanation of its purpose.

105 A symbolic constant using `#define`, object-like macro, was identified: Use of `#define`'s to control 'magic numbers' is better than use of literals. Note, it is considered good practice to enclose the consequence of a `#define` in parentheses; e.g. $SQRT2 = (1.414)$;. Studies have shown that this practice reduces certain types of error. Note however, that the practice cannot be arbitrarily applied; in some cases parentheses will cause error. Use of `enum` is generally preferred; if the project style is to accept `#define`s we recommend removal of this notice.

201 A 'Rights in Software Product' message not found in file. (AuditSoft is aware that lack of notice is being flagged on a function basis. The number of occurrences might be overstated.)

Thank you for your business.