QAC 度量元

QAC（v9.7）共包含度量元指标 71 个，其中函数度量元（Function-Based Metrics）36 个、文件度量元（File-Based Metrics）30 个、以及工程度量元（Project metrics）5 个。

说明：原版可参考用户手册 qac-manual.pdf 附录 A 章节"The Calculation Of Metrics"。

Note:
Project metrics and STM29 are generated by the cross-module analysis component RCMA.

工程度量元

STCYA：工程圈复杂度

计算方式如下：
n = Number Of Functions across the project

$$\sum_{i=0}^{n} STCY\ C_i$$

即：工程中所有函数的圈复杂度之和

STNEA：工程中入口点数量

通过计算调用图表和统计有未被调用的外部链接的结点数量来计算 STNEA
RCMA computes STNEA by examining the call graph and counting the number of nodes that have external linkage which are not called.

STNFA：工程中的函数个数

STNFA 计算方式如下
N = 工程中的文件数量

$$\sum_{i=0}^{n} STFNC_i$$

即所有文件中的函数数量之和

STNRA：工程中的递归数量

包括：直接递归函数和间接递归函数之和，推荐为 0

- ✓ 直接递归函数是直接调用自身的函数
- ✓ 间接递归函数是通过其他函数来调用自身的函数

STNGV：工程中定义的有外部链接的变量数量

函数度量元

Metric Description	Metric Description
STAKI	Akiyama's Criterion
STAV1	Average Size of Statement in Function (variant 1)
STAV2	Average Size of Statement in Function (variant 2)
STAV3	Average Size of Statement in Function (variant 3)
STBAK	Number of Backward Jumps
STCAL	Number of Functions Called from Function
STCYC	Cyclomatic Complexity
STELF	Number of Dangling Else-Ifs
STFDN	Number of distinct operands in a Function
STFDT	Number of distinct operators in a Function
STFN1	Number of Operator Occurrences in Function
STFN2	Number of Operand Occurrences in Function
STGTO	Number of Goto statements
STKDN	Knot Density
STKNT	Knot Count
STLCT	Number of Local Variables Declared
STLIN	Number of Code Lines

Metric Description	Metric Description
STLOP	Number of Logical Operators
STM07	Essential Cyclomatic Complexity
STM19	Number of Exit Points
STM29	Number of Functions Calling this Function
STMCC	Myer's Interval
STMIF	Deepest Level of Nesting
STPAR	Number of Function Parameters
STPBG	Residual Bugs (STPTH-based est.)
STPDN	Path Density
STPTH	Estimated Static Program Paths
NPATH	Strict Estimated Static Program Paths
STRET	Number of Return Points in Function
STST1	Number of Statements in Function (variant 1)
STST2	Number of Statements in Function (variant 2)
STST3	Number of Statements in Function (variant 3)
STSUB	Number of Function Calls
STUNR	Number of Unreachable Statements
STUNV	Unused or Non-Reused Variables
STXLN	Number of Executable Lines

STAKI: Akiyama 度量

STCYC（圈复杂度）和 STSUB（函数数量）之和
即：STAKI = STCYC + STSUB

STAVx：函数平均语句数

衡量程序的可读性的一项指标。
用来发现函数中过长的表达式，如果表达式过长的话会不利于理解，降低可读性。
STAV1 = (STFN1 + STFN2) / STST1
STAV2 = (STFN1 + STFN2) / STST2

STAV3 = (STFN1 + STFN2) / STST3
其中：
STFN1：函数中的操作符个数
STFN2：函数中的操作数个数
STSTx：函数体中语句数量

STBAK：函数中回跳点数量

程序的跳跃是不被推荐的，尤其是回跳更是应该避免。
这个指标统计函数中出现跳跃的次数。（例如：goto 语句的使用是一种跳跃）

STCAL：相异函数调用数

统计在函数中调用其他函数的次数。
区别于 STSUB：
1）多次调用相同函数，算一次调用
2）且不计算函数指针的调用

STCYC：圈复杂度

圈复杂度的值=决策点的数量+1
圈复杂度高时意味着函数的模块化不充分或者函数内逻辑过于复杂。软件度量研究标明，圈复杂度大于 10 的函数都可能存在复杂度的问题。

```
int divide(int x, int y)
{
     if (y != 0)                          /* 1 */
     {
          return x / y;
     }
     else if (x == 0)                     /* 2 */
     {
          return 1;
     }
     else
     {
          printf('div by zero\n');
          return 0;
     }
}
```

注意：QAC 中的 STCYC 的统计是基于判定语句的（statements），针对一些复合判定条件的语句，并未按照判定条件进行统计，参见如下描述：

Some metrication tools include use of the ternary operator ? : when calculating cyclomatic complexity. It could also be argued that use of the && and ||operators should be included. Instead, STCYC calculation is based on statements alone.

针对如下情况咨询了原厂，回复供参考：

- As you have correctly pointed out, STCYC metrics is calculated as the number of decisions + 1.
- But if you are also want the use of ternary operators to be included in the metrics calculation. Then you can make use of STMCC Myer's Interval
- STMCC Myer's Interval is an extension to the cyclomatic complexity metric. Myer's Interval is defined as STCYC : STCYC + L
- Cyclomatic complexity (STCYC) is a measure of the number of decisions in the control flow of a function.
- L is the value of the Helix QAC for C STLOP metric which is a measure of the number of logical operators (&&, ||) in the conditional expressions of a function.

详细解释参考如下链接：
https://www.perforce.com/blog/qac/what-cyclomatic-complexity

STELF：不完整的 Else-Ifs 个数

用来统计没有以 else 结束的 if 语句数量。

STFDN：函数中相异操作数的个数

操作数：指的是操作符相邻的变量或常量。 例如：a+b，'+'是操作符，a,b 是操作数。

- ✧ STFN2:统计函数体中所有的操作数数量
- ✧ **STFDN:统计一个函数中所有的唯一操作数数量**
- ✧ STM20：统计文件中所有的操作数数量
- ✧ STOPN:统计文件中所有的唯一操作数数量，同一操作数重复出现，只记做一次

示例：

```
extern int result;          /* 1 -> result */
static int other_result;    /* 2 -> other_result */

void main()                 /* 3 -> main */
  {
    int x;                  /* 4 -> x */
    int y;                  /* 5 -> y */
    int z;                  /* 6 -> z */

    x = 45;                 /* 7 -> 45 */
    y = 45;                 /* 8 -> 45 */
    z = 1;                  /* 9 -> 1 */
    result = 1;             /* 10 -> 1 */
    other_result = 0;       /* 11 -> 0 */

    return (x + other_result);
  }
```

上面的示例中包括六个自定义变量，五个常量，常量即使有相同的值也视作不唯一。STFDN=11。

STFDT：函数中相异操作符的个数

操作符：代码中非用户定义的符号，例如关键词、标点、操作符。

✧ STFN1：统计函数体中所有的操作符数量
✧ **STFDT：统计函数体中所有的唯一操作符数量**
✧ STM21：统计文件中所有的操作符数量
✧ STOPT：统计文件中所有的唯一操作符数量，同一操作符重复出现，只记做一次
示例：

```
extern int result;              /* 1,2,3 -> extern, int, ; */
static int other_result;        /* 4 -> static */
main()                          /* 5,6 -> ()*/
{                               /* 7 -> { */
    int x;
     int y;
     int z;
     x = 45;                    /* 8 -> = */
     y = 45;
     z = 1;
     result = 1;
     other_result = 0;
     return (x + other_result);     /* 9,10 -> return, + */
}                                   /* 11 -> } */
```

在上面的示例中，extern、int 等关键字，";" , ","等标点，"+"等操作符一共不重复的出现了 11 次，STFDT=11。

STFN1: 函数操作符出现次数

操作符：代码中非用户定义的符号，例如关键词、标点、操作符。
✧ **STFN1：统计函数体中所有的操作符数量**
✧ STFDT：统计函数体中所有的唯一操作符数量
✧ STM21：统计文件中所有的操作符数量
✧ STOPT：统计文件中所有的唯一操作符数量，同一操作符重复出现，只记做一次

STFN2: 函数操作数出现次数

操作数：指的是操作符相邻的变量或常量。 例如：a+b，'+'是操作符，a,b 是操作数。
✧ **STFN2:统计函数体中所有的操作数数量**
✧ STFDN:统计一个函数中所有的唯一操作数数量
✧ STM20：统计文件中所有的操作数数量
✧ STOPN:统计文件中所有的唯一操作数数量，同一操作数重复出现，只记做一次

STGTO: Goto 语句数量

计算 Goto 语句的数量。

有些 Goto 语句会简化错误的处理，然而，应当尽可能的避免使用它们，尤其是避免使用其回跳（即 Goto 到该语句之前的处理）

STKDN：交叉点密度

平均每一行可执行代码出现结点（交叉点）的个数
STKDN = STKNT / STXLN

STKNT：交叉点个数

统计函数中结点的个数
结点是控制结构中的转折点，明确表示跳出当前控制结构，一般由 continue，break 等关键词控制。度量值由统计以下几个关键词的个数得出：Goto，continue，break，return
该度量对交叉点的计算不是通过统计控制结构的交叉，而是通过计算下列代表交叉点的关键字：

- ✓ goto 语句
- ✓ 循环中的 continue 语句
- ✓ 循环或 switch 语句中的 break 语句，顶层的 switch 除外
- ✓ 所有 return 语句，顶层函数的除外

STLCT：声明的局部变量个数

该度量表示在函数中声明的具有 auto，register 或 static 存储类型的局部变量的数目

```
int other_result;
extern int result;
int test()
{
      int x;                              /* 1 */
      int y;                              /* 2 */
      return (x + y + other_result);
}
```

STLIN：可维护代码行数

该度量表示函数定义中（大括号之间但不包括大括号）的代码总行数，包括注释行和空行。
该度量值与代码可读性相关，越长的函数越难读，为便于单屏或单页阅读，建议该度量值上限为 200
注意：函数中的#include 行以及括号内的宏定义不在此计算范围内

STLOP：逻辑操作符个数

统计 do-while,for,if,switch,while 等表达式的条件中逻辑操作符（&&,||）的数量

```
void fn( int n, int array[], int key )
{
        while (( array[n] != key ) && ( n > 0 ))
        {
                if (( array[n] == 999) || ( array[n] == 1000 ))
                {
                        break;
                }
                else
                {
                        ++n;
                }
        }
}
```

1

2

STM07：基本圈复杂度

基本圈复杂度的计算方式和 STCYC 一样，不过它是基于一个简化的控制流图。简化流图的目的是为了检查组件是否遵从结构化编码的规则。如果流图能够被简化为圈复杂度是 1，说明它是结构化的，否则将展示出控制图中不遵守结构化编码规则的元素。

McCabe 标识了以下四种导致非结构化控制图的情况：

- ✓ a branch into a decision structure; 判定结构分支
- ✓ a branch from inside a decision structure; 判定结构内部的分支
- ✓ a branch into a loop structure; 循环结构分支
- ✓ a branch from inside a loop structure. 循环结果内部的分支

STM19：函数 Return 语句数

该度量通过 return 语句数量来计算软件模块出口数

注意：

- ✓ 没有 return 语句的函数，虽然执行最后一条语句后会退出，但其 STM19 值仍为 0。
- ✓ 该度量值也考虑 return 是否有返回值（例如，return void）。
- ✓ 并且，也不考虑特定出口函数的调用，例如：exit()，abord()。

STM29：函数被调用的频次（RCMA 分析参与时有效）

表示某个函数被调用的次数，反映当前函数被调用的频次

一个函数被调用的越频繁表明这个函数越重要，从而这个函数应该更可靠

STMCC：Myer's 内部复杂度

圈复杂度的扩展，以一对冒号分割的数字表达“SYCYC:STCYC+L”。实际上，STMCC=STCYC+L

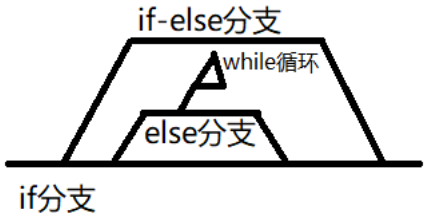
其中，

- ✓ SYCYC 是圈复杂度
- ✓ L 是 STLOP，表示函数里面函数表达式中的逻辑操作符数量。L 过高表示有很多的复合决策，导致程序难以理解。

导出 metrics 或在 metrics browser 中，显示的值都是 L 的值，而不是 SYCYC:STCYC+L.

STMIF：嵌套深度

该度量表示代码中控制流图中最大的嵌套深度。推荐最大值为 5。
降低该值的办法之一是将嵌套拆分成多个函数，这样可以改进代码可读性，降低嵌套数和函数圈复杂度。
一般，switch,do,while,if,for 语句会增加嵌套深度。




```
int divide(int x, int y)
{
     if (y != 0)                                    /* 1 */
     {
          return (x/y);
     }
     else if (x == 0)                               /* 2 */
     {
          return 1;
     }
     else
     {
          printf("Divide by zero\n");
          while(x > 1)                              /* 3 */
              printf( “x = %i ”, x);
          return 0;
     }
}
```

STPAR：函数参数个数

统计函数的参数个数，会忽略省略的参数

STPBG：基于路径的 Bug 数统计

霍普金斯发现了静态路径数（STPTH）和已发现 bug 数(STPBG)的关系。
STPBG = $\log_{10}$ (STPTH)

STPDN：路径密度

路径密度，是函数中路径的条数对比可执行代码的行数。
STPDN = STPTH / STXLN
其中，当 STXLN 为 0 时，STPDN 统计为 0

STPTH：估计静态路径数

给出函数控制流中可能的路径数目的上限，是函数中非循环的执行路径数。
一系列相同嵌套级别语句的 NPATH 值由每一个语句的 NPATH 值和嵌套结构产生
计算方式如下：

- NPATH(sequence of non control statements) = 1
- NPATH(if) = NPATH(body of then) + NPATH(body of else)
- NPATH(while) = NPATH(body of while) + 1
- NPATH(do while) = NPATH(body of while) + 1
- NPATH(for) = NPATH(body of for) + 1
- NPATH(switch) = Sum(NPATH(body of case 1) ... NPATH(body of case n))

注意：

- ✓ else 和 default 不管是否在代码中出现，都会被计算在内
- ✓ switch 语句中，同一分支上的多个 case 选项只计算一次
- ✓ 如果函数中有 GOTO 语句，则不能计算 NPATH 值。
- ✓ 函数中条件为真的路径数通常遵循如下不等式：圈复杂度 ≤ 真值路径数 ≤ 静态路径数

NPATH: 严苛的估计静态路径数

与 STPTH 略有不同

- NPATH(sequence of non control statements) = Product(NPATH(expression statements))
- NPATH(if) = NPATH(controlling expression) + NPATH(body of then) + NPATH(body of else)
- NPATH(while) = NPATH(controlling expression) + NPATH(body of while) + 1
- NPATH(do while) = NPATH(controlling expression) + NPATH(body of while) + 1
- NPATH(for) = NPATH(initialization expression) * (NPATH(controlling expression) +
- NPATH(body of for) + NPATH(increment expression) + 1)
- NPATH(switch) = Sum(NPATH(body of case 1) ... NPATH(body of case n))
- NPATH(?: expression) = NPATH(condition) + NPATH(true branch) + NPATH(false branch)
- NPATH(expression) = (number of logical operators) + 1

示例：

```
n && p;                    /* block 9, paths 2 */
```

STRET：函数中 Return 点的个数

统计函数中可达的 return 语句的个数，函数结束处的隐式 return 也算作一个点
结构化的编程要求每个函数只有一个入口和一个出口，这个度量元用来找出那些过多使用 return 语句的函数。

```
void foo( int x, int y )
{
        printf( "x=%d, y=%d\n", x, y );
        /* Here with implicit return. Hence STRET = 1*/
}
```

STSTx：函数语句个数

这个度量元用来统计函数中语句的个数

- ✧ STST1 是基本定义，统计函数中的所有语句
- ✧ STST2 在 STST1 的基础上去掉 block，empty statements 和 labels
- ✧ STST3 在 STST2 的基础上去掉声明语句

语句类型	STST1	STST2	STST3
块	yes	ignore	ignore
以;号结尾的简单语句	yes	yes	yes
空语句	yes	ignore	ignore
声明语句	yes	yes	ignore
标签	yes	ignore	ignore
break	yes	yes	yes
continue	yes	yes	yes
do	yes	yes	yes
for	yes	yes	yes
goto	yes	yes	yes
if	yes	yes	yes
return	yes	yes	yes
switch	yes	yes	yes
while	yes	yes	yes

用来作为程序可维护性的指标。这个值越大，语句越多，操作符和操作数也通常越多，因此程序更难被理解。当函数中的 statement 过多时，建议重构成更小的函数。

STSUB：调用子函数总次数（区别 STCAL）

在一个函数中调用其他函数的总次数

- ✓ 调用同一函数多次，计次数；
- ✓ 函数指针调用计算入内

STSUB 过高通常预示程序设计不良。过多的调用其他函数，可能会使函数不易理解，因为

函数功能位于多个组件。

STUNR：不可达语句个数

统计函数中永远不会被执行到的 statement 个数，statement 和 STST1 相同

STUNV：未使用变量的个数（Unused or Non-Reused）

unused variable：定义但未被引用
nonreused variable：分配了数值，但后续未使用

```
int other_result;
extern int result;

int test()
{
        int y;                                  /* Unused */
        int z;

        z = 1;                                  /* Non-Reused */

        return (result + other_result);
```

STXLN：可执行代码行

函数中可执行代码的行数。注释，括号和声明不被算作其中

文件度量元

Metric Name	Metric Description
STBME	Embedded Programmer Months
STBMO	Organic Programmer Months
STBMS	Semi-detached Programmer Months
STBUG	Residual Bugs (token-based estimate)
STCDN	Comment to Code Ratio
STDEV	Estimated Development (programmer-days)
STDIF	Program Difficulty
STECT	Number of External Variables Declared
STEFF	Program Effort
STFCO	Estimated Function Coupling
STFNC	Number of Functions in File
STHAL	Halstead Prediction of STTOT
STM20	Number of Operand Occurrences
STM21	Number of Operator Occurrences
STM22	Number of Statements
STM28	Number of Non-Header Comments
STM33	Number of Internal Comments
STOPN	Number of Distinct Operands
STOPT	Number of Distinct Operators
STSCT	Number of Static Variables Declared
STSHN	Shannon Information Content
STTDE	Embedded Total Months
STTDO	Organic Total Months
STTDS	Semi-detached Total Months
STTLN	Total Preprocessed Source Lines
STTOT	Total Number of Tokens Used
STTPP	Total Unpreprocessed Code Lines
STVAR	Total Number of Variables
STVOL	Program Volume
STZIP	Zipf Prediction of STTOT

COCOMO METRICS

这个度量元的计算基于代码的规模，需要考虑系统类型的因素。
系统类型被划分为以下三种：

- ✧ Embedded：在这个环境中，软件高度依赖环境。这些系统一般包括实时软件，是一个更大的基于硬件的系统的一部分例如导弹制导系统。
- ✧ Organic：在这个环境中，开发者几乎没有外部接口或者硬件约束。Organic 系统一般使用数据库，更关注数据交换，例如银行或账号系统。
- ✧ Semi-detached:在这个环境中，开发一般在一个更大的产品中，这个产品的多个组件之间有很复杂的接口，例如 windows。

STBME：Embedded Programmer Months

Embedded 嵌入式环境中，预估的所需程序员月
STBME=3.6 * (STTPP / 1000) $^{1.20}$
其中，STTPP 为预处理之后的代码行数

STBMO：Organic Programmer Months

Organic 环境中，预估的所需程序员月
STBMO = 2.4 * (STTPP / 1000) $^{1.05}$

STBMS：Semi-detached Programmer Months

Semi-detached 环境中，预估的所需程序员月
STBMS = 3.0 * (STTPP / 1000) $^{1.20}$

STBUG：Residual Bugs

估计的 BUG 数
STBUG = 0.001 * STEFF $^{2/3}$
其中，STEFF 为程序员努力度

STCDN：Comment To Code Ratio

代码注释率=*注释中可见字符个数 / 注释外可见字符个数*
可通过设置-comment *{a|n|i}*进行不同的注释统计，默认为 n

- ✓ 该值过大表示可能注释太多，程序难以阅读
- ✓ 太小表示可能注释太少，程序难以理解

STDEV：Estimated Development(programmer-days)

预估开发源代码所需要的程序员/天，不同于 cocomo 统计，它只与代码总量相关，这个评估

由文件的难度推导得出，是一个更精确的开发时间评估，尤其是当缩放因素对于特定的软件环境调整过后。
STDEV = STEFF / dev_scaling
其中，dev_scaling 是一个缩放因素，在 qac.cfg 中定义，默认值是 6000

STDIF: Programmer Difficulty

这是一个对于翻译单元（预编译后的代码）的难度度量，c 工程的平均难度值是 12，任何显著高于这个值的代码，都有丰富的词汇表，还存在潜在的理解困难。
STDIF = STVOL / ((2 + STVAR) * $\log_2$ (2 + STVAR))

STECT：Number Of External Variables Declared

外部链接声明的变量个数
期望：将对全局数据的依赖降到最低

STFCO：Estimated Function Coupling

估计函数的耦合度，较高的数字表示调用树的级别之间的复杂性发生了很大的变化。
STFCO = Σ(STSUB) - STFNC + 1
其中，STSUB 为“函数调用次数”（function metric），即调用子函数的总次数

STEFF: Programmer Effort

这个度量元用来衡量对代码开发投入的努力，被用来估算开发时间
STEFF = STVOL * STDIF

STFNC:Number Of Functions In File

文件中的函数总数

STHAL: Halstead Prediction of STTOT

这个度量元和 STZIP 预测 STOTT 应该是什么（由词汇表分析度量元推导出），如果他们和 STOTT 相差超过 2，表明要么函数过于重复，要么有过于丰富的词汇表。
STZIP = (STOPN + STOPT) * (0.5772 + ln (STOPN + STOPT))
STHAL = STOPT * $\log_2$ (STOPT) + STOPN * $\log_2$ (STOPN)

STM20：Number Of Operand Occurrences

文件中操作数的数量

void f(int a)　　/ 1,2 -> f, a */*

STM21：Number Of Operator Occurrences

文件中操作符的数量

void f(int a)　　/ 1,2,3,4 -> void, (, int,) */*

STM22：Number Of Statements

文件中语句的数量，也是软件中分号的数量，除了以下几种分号

- ✧ 在 for 表达式中的分号
- ✧ 在 struct 或 union 定义中的分号
- ✧ 注释中的分号
- ✧ 预处理指令中的分号
- ✧ 常量（literals）中的分号
- ✧ 老式 c 语言的参数列表中的分号

STM28：Number Of Non-Header Comments

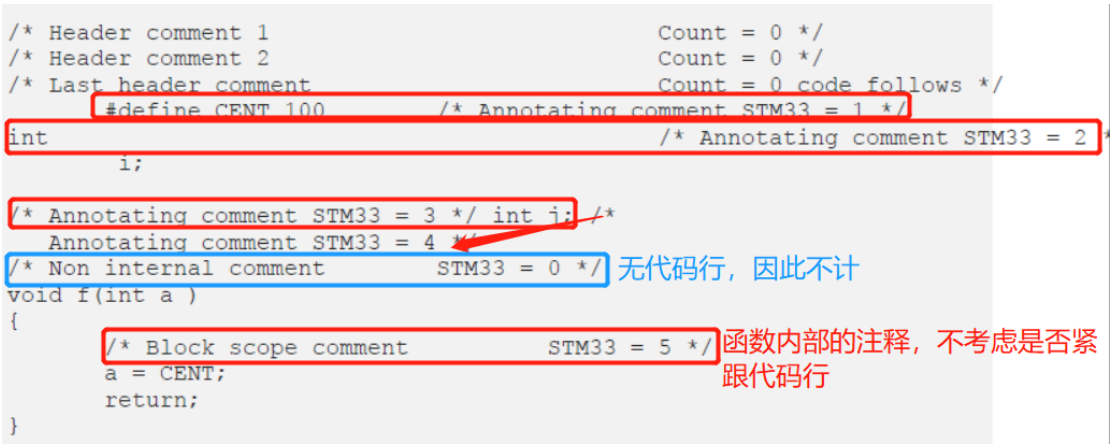
统计源文件中注释出现的次数，除了那些在文件头部出现的注释

```
/* Header comment 1     Count = 0 */
/* Header comment 2     Count = 0 */
/* Last header comment code follows */

#define CENT 100        Count = 0

/* Non header comment  Count = 1 */
void f( int a )
{
        /* Block scope comment Count = 2 */ a = CENT;
        return;
}
```

STM33：Number of Internal Comments

用来统计注释出现的次数。

- ✧ 函数外部的注释，只有当注释的开头行或结尾行有代码时才算做一次有效出现。如果注释所在行，或注释开头或注释结尾都没有代码，则这处注释被忽略。
- ✧ 函数内部的注释，必定被计数。



STOPN：Number Of Distinct Operands

统计相异操作符的数量，相同的操作符视作一个操作符

STOPT：Number Of Distinct Operators

统计相异操作数的数量，相同的操作数视作一个操作数

STSCT：Number of Static Variables Declared

文件中静态变量或函数出现的次数（static 修饰）

STSHN：Shannon Information Content

用来估算编码源文件中的函数所需要的空间，以 bit 为单位。
STSHN = STZIP * log2 (√ (STOPN + STOPT) + ln (STOPN + STOPT))

STTDE: Embedded Total Months

Embedded 嵌入式系统中开发源代码所需要经过的时间
$STTDE = 2.5 * STBME^{0.32}$

STTDO: Organic Total Months

Organic 系统中开发源代码所需要经过的时间
$STTDO = 2.5 * STBMO^{0.38}$

STTDS: Semi-detached Total Months

Semi-detached 系统中嵌入式系统中开发源代码所需要经过的时间

$STTDS = 2.5 * STBMS^{0.35}$

STTLN:Total Preprocessed Source Lines

代码预编译后的行数

STTOT:Total Number Of Tokens Used

符号的总个数

STTPP:Total Unpreprocessed Code Lines

预编译前的代码行数

STVAR:Total Number Of Variables

总的变量数，被定义的函数也算一个变量

STVOL: Program Volume

计算将一个工程文本转换成二进制编码所需要的字节数，它的值被用来计算不同的 halstead vocabulary metrics
$STVOL = STTOT * \log_2 (STOPN + STOPT)$

STZIP:Zipf Prediction of STTOT

$STZIP = (STOPN + STOPT) * (0.5772 + \ln (STOPN + STOPT))$