

IMPORTANT NOTICE

DISCLAIMER OF WARRANTY

The staff of Programming Research Ltd have taken due care in preparing this document which is believed to be accurate at the time of printing. However, no liability can be accepted for errors or omissions nor should this document be considered as an expressed or implied warranty that the products described perform as specified within.

COPYRIGHT NOTICE

This document is copyrighted and may not, in whole or in part, be copied, reproduced, disclosed, transferred, translated, or reduced to any form, including electronic medium or machine-readable form, or transmitted by any means, electronic or otherwise, unless Programming Research Ltd consents in writing in advance.

TRADEMARKS

PR:QA, the PR:QA logo, and QA C are registered trademarks of Programming Research Ltd. Windows is a registered trademark of Microsoft Corporation.

CONTACTING PROGRAMMING RESEARCH LTD

For technical support, contact your nearest Programming Research Ltd authorised distributor or you can contact Programming Research's head office:

by telephone on	+44 (0) 1 932 888 080
by fax on	+44 (0) 1 932 888 081
or by e-mail on	support@programmingresearch.com

CHAPTER 1.....	7
Introduction to QA C	7
CHAPTER 2.....	10
Projects.....	10
Creating a Project Automatically	10
Creating a Project Manually	12
Reopening Projects.....	12
Making File Selections	14
Relative Path and Environment Variable Support	14
Root Path.....	15
Applying Relative Path and Environment Variables	15
CHAPTER 3.....	19
Configuring QA C.....	19
Configuring Your Compiler Personality	20
Setting System Header Files	20
Setting System Macros	21
Setting Implementation Defined Types	21
Configuring Compiler Extensions.....	22
Configuring Your Analyser Personality	24
Setting Project Header Files	24
Setting Project Macros.....	25
Setting Brace Style and Tab Spacing	25
Configuring Your Message Personality.....	25
Choosing a Message Subset	26
Standard Message File	26
Level 0: Information	26
Level 1: Unused	27
Level 2: Minor	27
Level 3: Major	27
Level 4: Local Standards	27
Level 5: Unused.....	28
Level 6: Portability.....	28
Level 7: Undefined Behaviour	28
Level 8: Language Constraints	29
Level 9: Errors	29
Message Suppression	29
Code-based Suppression Syntax.....	29
Suppression of Warnings with #pragma	30
Project Configuration File.....	31
Personalities and Analysis	32
Application Options.....	33
Choosing Your Editor and Browser.....	33

Changing Your Annotated Source Settings	33
Setting the Housekeeping Options	34
CHAPTER 4	36
Analysing Source Code	36
Commencing Analysis	37
Analysis Output Files	37
.err Files	38
.met Files	38
.i Files	39
.txt and .html files	40
Managing Output Files	40
Secondary Analysis	42
Configuring Secondary Analysis	42
Naming Convention Checking	43
Running Multiple Secondary Analysis	45
Cross-Module Analysis	46
Configuring Cross-Module Analysis	46
CMA Output Files	47
Running Cross-Module Analysis	48
Default-supplied CMA Settings	48
Analysis Log	49
CHAPTER 5	50
Viewing Analysis Results	50
The Message Browser	51
The Source View	52
Warning Summary View	55
Warning List View	56
Viewing Annotated Source Code	56
Changing the Format of Annotated Source	57
Using Annotated Source Code	58
Sample Annotated Source Code Listing	58
The Corrected Source File	65
CHAPTER 6	69
Reports	69
Project Warning Summary	70
Warning Summary Message Count	70
Warning Listing	71
Warning Listing Message Count	71
Identifier Declarations	72
Close Name Analysis	72
External Name Cross-Reference	73
COCOMO Cost Model	73
Project Metrics	76

CHAPTER 7	77
Code Structure	77
Relationships	77
Function Structure	80
Viewing Function Structure Source Code	82
CHAPTER 8	83
Metrics	83
Metric Thresholds in Analysis	83
The Metrics Browser	86
The Demographics Browser	90
Exporting Metrics	93
The Kiviat Diagram	94
CHAPTER 9	96
Running QA C on the Command Line	96
Environment Variables	96
Configuration Options	97
Analysing Source	100
Configuring Primary Analysis	100
Running Secondary Analysis Checks	101
Running CMA Checks	102
Viewing Diagnostic Output	102
Launching the Message Browser	102
Summary Diagnostic Output	103
Annotated Source Generation	104
Warning Listing Report	105
CHAPTER 10	107
Advanced Topics	107
Customising the Message System	107
The Message File (qac.msg)	107
User Message Files	109
Formatting Message Output	111
Environment Variables	116
Writing Secondary Analysis Checks	116
Writing Messages to the Error File	117
Writing Custom Reports	118
APPENDIX A	121
Personalities	121
Message Personality	121
Analyser Personality	124
Compiler Personality	125
APPENDIX B	127
Configuration Options	127

APPENDIX C	164
The Components of Function Structure	164
APPENDIX D	172
The Calculation of Metrics	172
Function-Based Metrics	172
File-Based Metrics	192
Project-Wide Metrics	204
APPENDIX E.....	206
Program Return Codes	206
APPENDIX F	208
Metric Output File	208
Relationship records.....	208
Relationship type records.....	209
External Reference Records	209
Define records	210
Control Graph Records	212
Metrics Records	213
Pragma Records	213
Literal Records	214
APPENDIX G	216
QA C Utilities.....	216
r_basename	216
r_close.....	216
r_fields.....	217
r_grep.....	218
r_sort	219
r_uniq	219
APPENDIX H	220
Code Suppression.....	220
Code-based Annotations.....	220
Location Tag Syntax.....	220
Predefined Location Tags	222
Suppression Syntax	222
Continuous Suppression Syntax	223
Use-Case Examples.....	224
Single instance suppression.....	224
Range suppression using location tags.....	225
Range suppression using line counting.....	226
Continuous suppressions	228
Suppressions in header files	230
Force include suppression entries.....	231
Suppression input failures	233

APPENDIX I	236
Naming Convention Checking	236
Introduction	236
Configuration Basics	236
Configuration File.....	236
Rule Format (JSON Syntax)	237
Rule Names	237
Perl Compatible Regular Expressions	242
Matching Characters.....	242
Matching a Number of Times.....	243
Anchoring.....	244
Alternate Matching.....	244
Escaping Special Characters.....	245
Configuration File Example.....	245
Getting Feedback on Errors.....	245
INDEX.....	248

CHAPTER 1

Introduction to QA C

QA C is a deep flow static analyser for C code, and is designed to help you improve the quality of your software development.

QA C analyses source code on a file-by-file and complete project basis to identify dangerous usage of the C language. A library of over 1000 warning messages is used to highlight source code which is non-portable, difficult to maintain, overly complex, or written in any way that is likely to cause problems.

The tool will also identify language usage which is not compliant with the ISO C standard (ISO/IEC 9899:1990) or which is classified as giving rise to unspecified, undefined or implementation-defined behaviour.

Warning messages for a complete project of source code are displayed through a Message Browser component that categorises and groups message occurrences across all source files. Message output can also be viewed in the form of an annotated source code listing which can optionally be presented in HTML format with links to additional information and advice.

Other features of the tool include:

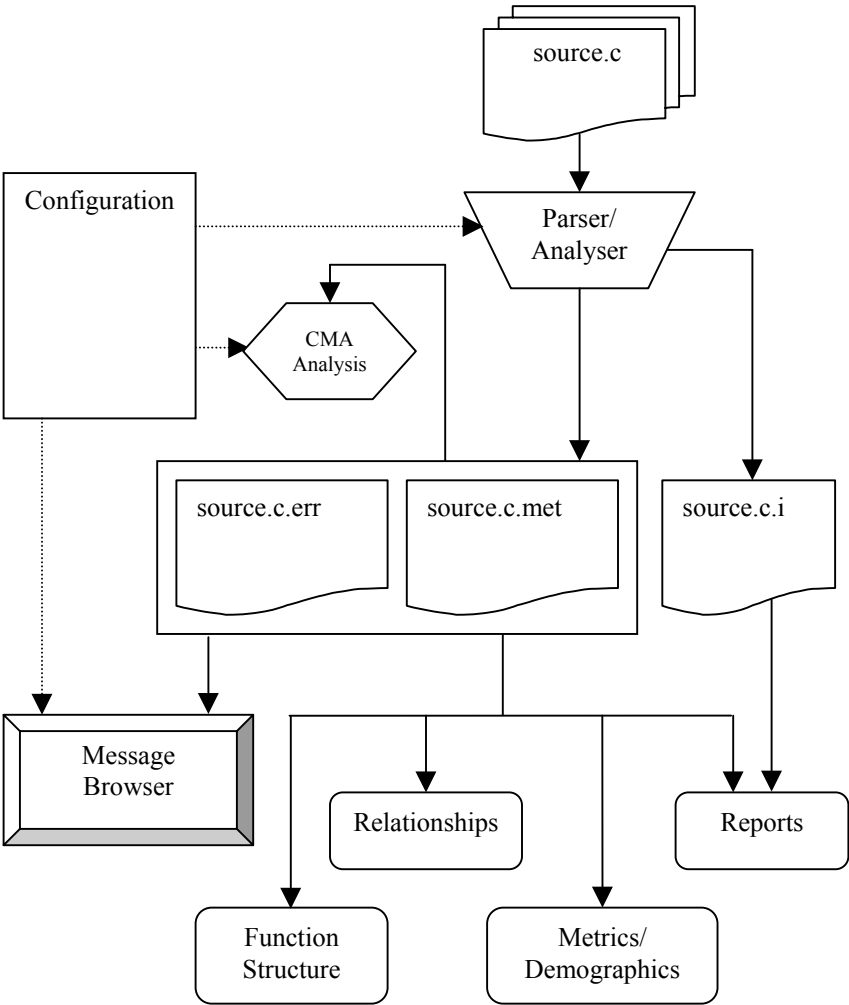
- 33 function-based, 32 file-based, and 4 project-level code metrics which provide a quantifiable measure of numerous code attributes,
- function structure diagrams to provide an insight into control flow,
- relationship diagrams to demonstrate function calling, global referencing, and include file trees,
- demographic analysis to provide an overall assessment of code quality against industry benchmarks.
- cross-module analysis capability, delivering function recursion, and various global identifier issues.

QA C can perform analysis in three phases. Primary analysis is

provided by the inbuilt tool, which delivers the majority of language analysis in the product. Secondary analysis is an optional user-configurable add-on, and usually contains industry, company or standards-specific checks. Cross-module analysis covers a range of checks that needs to operate across translation units. QA C is supplied with a default module of CMA analysis, and additional modules can add to or replace this module.

By suitable configuration of the tool and addition of suitable secondary and CMA analysis it is often possible to apply a high degree of automatic enforcement to a coding standard.

The flow chart on the next page shows an overview of the analysis process and the various output files that are generated.



CHAPTER 2

Projects

In QA C, a project reflects the structure of a development project. Projects contain one or more folders, source files, and configuration information. Project files are saved with a `.prj` file extension.

The properties of each folder are made up of the default source path, the output path, and a set of configuration files known as personalities. While each folder in the project tree can have a unique set of properties, usually all folders in a project share the same output path and personalities.

Note:

If you have same-named source files in different project folders, then these folders should not share the same output path since the successive analysis of these files will overwrite each other.

Projects also contain a single project configuration file, which can contain items of configuration that are common across all folders, or additional configuration required by product customisation. See Chapter 3: Project Configuration File for more details.

There are two ways to create a new project:

- Automatically, by choosing Auto-Create Project from the File menu. A folder structure is created to match your directory structure.
- Manually, by choosing New Project from the File menu. Source files and folders must be added manually.

Creating a Project Automatically

When you create a new project automatically, QA C will add all sub-directories and files starting from a specified directory. The resulting folder structure will reflect the structure of the source code directories.

To create a new project automatically:

1. Choose Auto-Create Project from the File menu.
2. Enter the Root Folder Name. This is the name of your project's root folder. Other folders will be named automatically with the name of the corresponding source directory.
3. Enter the Starting Directory. This is the source code directory associated with the root folder of your project.
4. Enter an Output File Path. This is the directory to which QA C generates output files during analysis. By default, this path is the `temp` directory within your local QA C installation, but it is better practice to use a dedicated directory associated with your project.
5. Click Replicate source tree structure in output paths to create a sub-directory structure for output file path(s). Choose Parallel to Source Structure to create a directory structure in parallel with the source code. Choose Sub-path to each source location to embed your specified output path subdirectories beneath each source directory. Either of these strategies will avoid same-named files from different source directories overwriting their analysis output files.
6. Select the File Extensions of the files that you want added to your project. QA C will add all sub-folders of your starting directory that contain files with the specified file extensions.

Usually you will want to select only your source code files (`.c`). QA C will only add folders to your project if they contain source files.

If you have include (`.h`) files within your directory structure, it will not generally be appropriate to select them into your project. The location of included files will be specified later.

7. Select the personalities for the folder. You may want to use the default set as a starting point. Personalities can be edited later by choosing them from the Configuration menu, see Chapter 3: Configuring QA C.

8. Click OK to create your project.

Your project structure is now created and should contain your source files and sub-folders.

9. Save your project. Projects are saved with the `.prj` file extension.

Note:

You can also auto-create a folder structure into an existing project, using menu option Edit | Auto-create Sub-Folders, matching the procedure above.

Creating a Project Manually

When you create a new project manually, you build it by adding folders and selecting the files yourself.

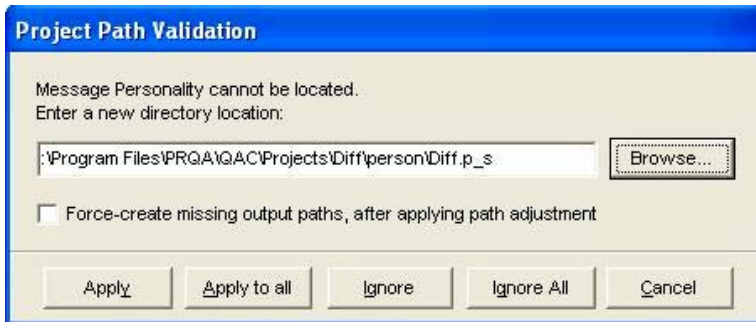
10. Choose New Project from the File menu. The New Project Parameters dialog box appears.
11. Enter the Root Folder Name.
12. Enter the Default Source Path for your project's initial folder. This path can be changed for each sub-folder.
13. In Output File Path enter the directory to which QA C generates output files during analysis.
14. Select personalities for the project.
15. Click OK to create your project. Your project is now configured with a single folder to which you can add files and sub-folders.
16. Save your project.

Reopening Projects

With the File | Reopen menu item, you can select from up to ten recently opened projects.. Any projects no longer accessible will be

greyed out. For such cases, there is a menu item, Clean-up, which will remove all such unavailable history.

When reopening projects, file and directory locations are checked for existence. If an entry does not exist, the following dialog is shown. There is a process by which corrections can automatically be applied.



1. First, make a correction to the entry, by either using the Browse button or entering a corrected location into the edit box.
2. Click Apply to make this change for this entry only.
3. Click Apply to All to attempt the same path substitution for any further incorrect entries. When this correction does not result in a valid path, this same dialog will be presented. However, the same automatic path substitution will be attempted on succeeding entries for the most recent Apply to All instruction.
4. When the Force-create missing output paths, after applying path adjustment checkbox is turned on, after making any path adjustment for output paths these are force-created. This can be useful for projects extracted from a Source Control system, where the project output including directory was not saved, and will therefore not be present on project extraction.
5. Click Ignore to continue project load without making a change. Ignore All carries that instruction forward to all further incorrect paths. Cancel will halt the project load operation.

Making File Selections

Visually, the project is portrayed by a hierarchy of folders on the left, and, based on selection within this listing, the list of constituent files on the right.

All selections in the Browse and Reports menu items operate according to the current selection of files. The basis of selection is as follows:

- a) if a single or group of files are selected, these form the selection,
- b) otherwise the current selected folder, plus all of its sub-folders, form the selection. This means all constituent files in these folders.

From within all browsers, it is possible to alter this starting selection of files, using **Select Files...** on the File menu.

Relative Path and Environment Variable Support

QA C supports the use of relative paths and environment variables in project files and associated personality files. This path reduction operation involves the following principles:

- Relative paths are applied from the location of the project file, termed the Root Path.
- Relative paths and environment variables cannot be mixed in a single entry.
- The default and preferred means of applying path reduction is during save operation on a project.
- When applying path reduction preference is given to relative paths over environment variables.
- Application of path reduction can extend to associated personalities. An “associated” personality refers to a personality entry

within a project configuration that is subject to relative path reduction.

- In all analysis and display processes, any path reductions in projects are converted to their fully qualified paths, so that there is no dependence on relative paths or environment variables in these external processes through configuration files such as `settings.via` and `filelist.lst`.

Note:

In order to optimise your use of relative paths, locate the project file at the root of project components, including the source tree, project headers, project personality files, and analysis output locations.

Root Path

The key variable for application of relative paths is the location of the project file. This Root Path is displayed on the main GUI window in a toolbar entry. When a project is opened, it is automatically set to display the project file location and will change only if the project file is saved to a different location.

When editing personalities, any use of relative path entries will be resolved using the Root Path, so that browse and edit operations will work correctly.

Note:

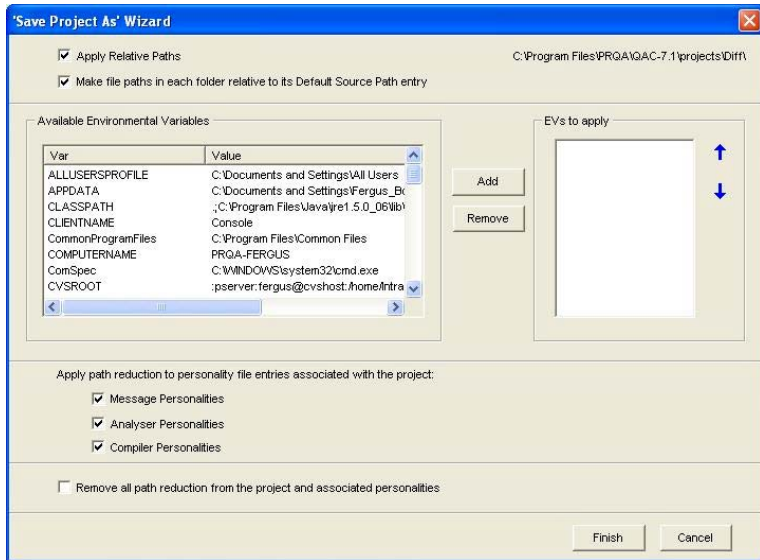
Although possible to apply relative path entries manually to existing (saved) projects, it is recommended to only apply relative paths and environment variables through project-saving operation. This step will automatically include the adjustment to associated personalities.

Applying Relative Path and Environment Variables

The standard way to apply relative paths and environment variables is through an automatic mechanism during the File | Save As... menu operation.

A dialog window presents various options for application of relative

paths and environment variables.



From this dialog, options for application of path reduction are:

6. Select the Apply Relative Paths option to apply relative path reduction to all file entries. The root path entry at top right, representing the save location of the project file, determines whether a path is suitable for relative path reduction.
7. Select Make file paths in each folder relative to its Default Source Path entry, if you want to apply a special pseudo-environment variable representing the default source path of each folder to other entries for that folder. This setting applies only to entries in project files.
8. Select from the list of Available Environment Variables, and add these to the EVs to Apply listing on the right, to apply these possible substitutions, in the order given. This substitution will only happen for path entries in the project file or associated personalities that are not subject to relative path reduction.
9. Select Apply path reduction to personality file entries associated

with the project, in order to continue applying relative paths and environment variables to path-based entries within personalities defined with relative paths in the project file.

10. Select Remove all path reduction from the project and associated personalities to undo all relative paths and environment variables from the project and associated personalities, reverting to fully qualified paths in all cases.

Note:

A project already containing relative paths cannot be re-saved to a different location, and the GUI will disallow this. It must be resaved as a full-path project in the same location and then saved again with relative paths into its new location. This is especially necessary because of the difficulties of managing associated personality files.

As an example, a reconstructed Diff project is shown below with full relative path implementation:

Project File Entries	Meaning
VersionTag45	project file version
StartProjectMarker	Root Folder:
FolderName=Diff	
SourcePath=src\	%SOURCEPATH%
OutputPath=output\	
SubsPers=person\Diff.p_s	“Associated”
AnalPers=person\Diff.p_a	personalities, can
CompPers=person\Diff.p_c	have path reduction
EndContainedFilesMarker	
StartSubProjectMarker	Sub-Folders
StartProjectMarker	1st sub-folder
FolderName=Opt	
SourcePath=src\	Redefined
OutputPath=output\	%SOURCEPATH%
SubsPers=person\Diff.p_s	“Associated”
AnalPers=person\Diff.p_a	personalities, can
CompPers=person\Diff.p_c	have path reduction
%SOURCEPATH%\Ifdef.c	
%SOURCEPATH%\Getopt.c	Use of redefined
%SOURCEPATH%\Ed.c	%SOURCEPATH%
EndContainedFilesMarker	
StartProjectMarker	2 nd sub-folder
FolderName=Analyse	
SourcePath=src\	Redefined
OutputPath=output\	%SOURCEPATH%
SubsPers=person\Diff.p_s	“Associated”
AnalPers=person\Diff.p_a	personalities, can
CompPers=person\Diff.p_c	have path reduction
%SOURCEPATH%\Dir.c	
%SOURCEPATH%\Diff.c	
%SOURCEPATH%\Context.c	Use of redefined
%SOURCEPATH%\Analyze.c	%SOURCEPATH%
%SOURCEPATH%\Alloca.c	
EndContainedFilesMarker	
EndSubProjectMarker	Project end marker

CHAPTER 3

Configuring QA C

One of the strengths of QA C is its high degree of configurability. Project configuration is controlled by personalities applied to each folder and a project configuration file. A personality is a group of configuration options that specifies how QA C analyses source code and displays analysis data. There can be a single project configuration file for each project, and its use is explained later in this chapter.

Personalities can be defined differently for each folder in each project. QA C analyses your source files using the personalities of each folder.

The personality configuration for each folder consists of:

- **Compiler Personality**, which defines options that reflect the configuration of your compiler,
- **Analysis Personality**, which defines analysis options that are associated with your project,
- **Message Personality**, which defines which messages are enabled and how output is displayed.

QA C is supplied with a number of personality configuration files of each type, but you may prefer to adapt these for your company standard, software project, or individual use.

To view the set of personalities in your installation along with those defined in your project, choose from Message Personalities, Analyser Personalities or Compiler Personalities under Configuration. For each of these, you will be presented with the complete list of available personalities, including those residing in added-on compliance modules. If you have a project loaded, you will also see a tree of the project folders and the personalities assigned to each one.

In this window, you can create, copy, or delete personalities. You can view the contents of any of your project or generic personalities. You can also assign personalities to different folders of your project. Whenever you alter and save a generic personality and a project is

loaded, you will be presented with the opportunity to apply the changes to folders in the current project.

Once you have created personalities, you can set them to be used as the default personalities for all projects by setting them in Configuration>Options>Default Personalities.

This chapter details the most common personality settings. For more information, see Appendix A: Personalities.

Configuring Your Compiler Personality

The Compiler Personality defines options that configure QA C to match the behaviour of your compiler. See Appendix A: Personalities for a list of all the Compiler Personality settings.

Note:

When amending **Compiler Personality** information, make sure that you are altering the file currently being used in your project folder settings, rather than one located in the general personality location. To ensure that you are editing the correct file, use the edit button within the folder parameters dialog box.

Setting System Header Files

In System Header Includes on the System Headers tab, set your system header include paths.

You can also suppress the output from these header files by clicking Suppress Output. When you suppress output from header files, analysis data from the specified header files or paths will not be generated to the .err or .met files. This greatly reduces the size of these files and decreases analysis time. See Suppressing Analysis Output from Header Files of Chapter 4: Analysing Source Code.

You can manually enter a path that is relative to the current project location, although it is recommended to make full path selections and then apply any chosen path reduction during the project save operation. See Applying Relative Path and Environment Variables for

more details.

Note:

When you create a new **Compiler Personality**, it will include, by default, the path to some header files that were written to match the standard ISO C header files. They are located in the `include\ansi` directory.

Setting System Macros

In System Macro Defines on the Project Macros tab, set up macros that are associated with your compiler or development environment. Macros can be defined in either the Compiler Personality or the Analyser Personality.

Setting Implementation Defined Types

In the header files associated with the standard ISO C library, type definitions exist for three characteristics of a C compiler which are “implementation defined”. These are:

<code>size_t</code>	An unsigned integral type which reflects the type returned by the <code>sizeof</code> operator.
<code>ptrdiff_t</code>	A signed integral type which reflects the type resulting from the subtraction of two pointers.
<code>wchar_t</code>	An integral type which reflects the type of wide character literals and wide character strings.

Intrinsic Types on the Data Types tab controls the way these types are implemented and needs to be configured to match your compiler environment. Any corresponding typedef statements contained in header files (e.g. `stddef.h`, `stdio.h` etc) must reflect the intrinsic types configured within QA C. If they are not consistent, QA C will generate a level 9 warning. Examine your header files, if necessary, to determine appropriate settings for these options.

As described above, a set of standard library header files are shipped with QA C. If you make use of these stub headers, you will not need to be concerned about intrinsic type settings, because the header files

contain `typedef` statements for `ptrdiff_t`, `size_t` and `wchar_t` which automatically reflect the intrinsic type settings. The `typedef` statements make use of three macro definitions, `PRQA_PTRDIFF_T`, `PRQA_SIZE_T`, and `PRQA_WCHAR_T`, which are implicitly defined within the QA C environment and correspond to the intrinsic type settings in the compiler personality.

Configuring Compiler Extensions

Many compiler manufacturers implement extensions to the ISO C language definition to take advantage of a particular hardware environment. This is particularly true in embedded software environments where speed and “tight” code are important.

The danger of using language extensions is that they compromise portability and source code becomes dependent on the compiler and hardware environment.

QA C is able to parse a wide variety of language variations and extensions, but is not usually able to interpret the extensions semantically. Usually it is a matter of configuring the tool so that non-standard keywords are simply ignored.

There are a number of ways in which QA C can be configured to do this. See the `-extensions` section of Appendix B for more details.

Standard extensions

In Extension Classes, you can enable the more common extensions to the C language. These include recognition of keywords such as `near`, `far`, and `huge`, the use of `$` in identifiers, and the handling of inline assembler blocks.

`#define` of extended keyword

Compiler manufacturers frequently introduce keywords such as `idata`, `tiny`, and `xhuge` to specify memory models and to allocate memory data. These keywords are often applied like storage class qualifiers in object declarations and can be ignored during analysis by

introducing a `#define` which defines away the keyword. For example, the keyword `idata` could be ignored by defining the system macro `idata=` in the System Macro Defines.

`_ignore`

The `_ignore` macro is an instruction to QA C to ignore not just a keyword but a sequence of tokens starting with a specified keyword. It specifies a macro definition that defines the keyword in one of a number of ways.

For example:

Macro definition	Tokens ignored
<code>using=_ignore</code>	using and the next token
<code>eseg=_ignore_semi</code>	<code>eseg</code> and everything up to and including the next semi-colon.
<code>asm=_ignore_paren</code>	<code>asm</code> and the contents of a block delimited by <code>()</code> , <code>{ }</code> or <code>[]</code>
<code>interrupt=_ignore_3</code>	<code>interrupt</code> and the following 3 tokens
<code>tiny=_ignore_at</code>	The sequence “ <code>@tiny</code> ” is ignored.

Note:

The macro `_munch` has been deprecated in favour of `_ignore`.

Redefinition of extended data types

Sometimes extended data types are introduced into the language to address memory mapped I/O ports or to perform efficient bit addressing. These data types can usually be redefined to a standard data type during analysis by introducing a macro or a `typedef`. For example, access to special function registers is often provided through data types such as `sfr` and `sbit`.

```
#define sfr unsigned char
typedef unsigned char sfr
```

Operators

The standard ISO C operators `^` and `.` are used in a non-standard way

for bit selection in conjunction with the special function register data types. If the data types have been `typedef`'d or redefined as suggested above, the `^` operator will be parsed successfully and interpreted as a "bitwise exclusive or" operator. The `.` operator which is normally the structure/union member selection operator is also recognised by QA C when used as a bit selection operator but warning 3664 is generated. This warning can be suppressed if you wish to use the `.` operator in this way.

Force include files

Macros can be defined by incorporating them into a personality. If you want to define `typedef`'s, they will need to be specified within your source code. However, rather than introducing them directly into your source code or header files, it is usually better to create a special header file and "force-include" this file during analysis. This is done in Additional Include File on the Extensions tab.

Configuring Your Analyser Personality

The Analyser Personality controls project settings such as your project's include paths, macro definitions, and code layout. See Appendix A: Personalities for a list of the Analyser Personality settings.

Note:

When amending **Analyser Personality** information, make sure that you are altering the file currently being used in your project folder settings, rather than one located in the general personality location.

Setting Project Header Files

In Header Includes on the Project Headers tab, set your project's header include paths.

You can manually enter a path that is relative to the current project location, although it is recommended to make full path selections and then apply any chosen path reduction during the project save operation. See Applying Relative Path and Environment Variables for more details.

Once your header files have been thoroughly tested and the remaining warning messages are of little importance, you may want to suppress their warning messages. See Suppressing Messages from Header Files of Chapter 4: Analysing Source Code.

Setting Project Macros

In Project Macro Defines on the Project Macros tab, set up macros associated with your project. Macros can be defined in either the Analyser Personality or the Compiler Personality.

Setting Brace Style and Tab Spacing

On the Style tab, choose a Brace Style and specify your tab spacing in Source Code Tab Spacing. Tab spacing is not the same as indentation. It defines the number of character positions associated with a single tab character.

Configuring Your Message Personality

Because of the large number of diagnostic checks performed by QA C, annotated source code may be generated with a large amount of messages, some of which may not be applicable to your project and environment. It is therefore important to generate annotated source code with only a subset of relevant messages.

As a starting point, restrict the subset of messages to those that highlight the most serious problems within your code such as configuration and syntax errors. Normally, your source code should not contain any of these messages. If they are generated, it is usually symptomatic of incorrect configuration or syntax errors that your compiler ignores.

By choosing this limited subset of messages, you will highlight the areas of your project that are not configured correctly.

Note:

When amending **Message Personality** information, make sure that you are altering the file currently being used in your project folder

settings, rather than one located in the general personality location.

Choosing a Message Subset

The easiest way to restrict the subset of enabled messages is to suppress levels on the Warning Messages tab in the Message Personality. Initially suppress all message levels except level 9 in Message Groups on the Warning Messages tab, by choosing Switch Message Level Off from the right-click popup menu.

Note:

Messages in message level 9 describe configuration, syntax and constraint errors.

Suppressed message levels will appear with a red **X** and no messages from these levels will appear in your annotated source code.

Note:

It is not necessary to re-analyse files in order to view the changes to the configuration of the display of output.

You can expand this subset by enabling more message levels. See Appendix A: Personalities for a list of the Message Personality settings.

Standard Message File

The standard message system in QA C is organised into levels and groups. There are 10 levels supplied in the default message set, and within each of these levels are groups representing an unlimited further division of message output. Following is an explanation for each level in the standard message file.

Level 0: Information

Information level warnings with the lowest level of importance.

Annotations	Syntax problems in annotations (used in code-based suppressions).
-------------	---

CMA information	Setup problems with CMA analysis.
Recovery	Informational messages generated in the course of recovering from a parsing error or an internal dataflow analysis failure.
Sub-Messages	Additional context sub-messages giving additional clarification to a primary message.

Level 1: Unused

Level 2: Minor

Messages which identify issues which may infringe coding standard requirements but which are not necessarily serious errors. The message groups are associated with different aspects of the C language.

Level 3: Major

Messages which are likely to identify a significant coding error, anomaly or problem. These messages (like Levels 7, 8 and 9) are generally considered too important to ignore without good justification. The message groups are associated with different aspects of the C language.

Level 4: Local Standards

Identifies features that do not conform to your programming standard. By convention, when a QA C compliance module is installed, additional message groups are added to Level 4 which will contain the messages corresponding to specific rules in the coding standard.

Local Standards	Local coding standard preparatory location.
-----------------	---

Level 5: Unused

Level 6: Portability

ISO-C90 Conformance Limits	Messages which identify violations of the translation limits defined in the C90 standard. Code which exceeds these limits may not be portable to all conforming C90 implementations.
ISO-C99 Conformance Limits	Messages which identify violations of the translation limits defined in the C99 standard. Code which exceeds these limits may not be portable to all conforming C99 implementations.
Implementation defined	Code constructs which will behave in ways which may vary between different compilers.
Language extensions	Code constructs which QA C is able to parse but which do not conform to the C90 or C99 standards.
ISO C99 Language features	C99 code constructs that are not part of the C90 language definition.

Level 7: Undefined Behaviour

CMA Undefined Behaviour	Constructs whose behaviour is explicitly described as undefined in the language standard and which are identified using CMA analysis.
Explicitly undefined	Constructs whose behaviour is explicitly described as undefined in the language standard.

Implicitly undefined	Constructs whose behaviour is not described in the language standard and whose behaviour is therefore undefined – but only by implication.
----------------------	--

Level 8: Language Constraints

These messages identify constructs which violate the language standard.

Level 9: Errors

These errors cannot be suppressed in output. Analysis (if any) produced by QA C will be incomplete and should not be relied upon.

QAC configuration	Messages highlighting problems with the configuration passed to the tools.
Syntax errors	Invalid syntax in the source code.

Message Suppression

QA C contains a rich comment-based suppression capability. This is more powerful, flexible, and extensible than the existing `#pragma` message suppression. There is a simple and intuitive syntax for entering suppressions. In the implementation, generation of diagnostics is separated from generation of suppressions, allowing for viewing tools to apply suppressions across all analysis phases. There is also extensive reporting on suppression application across a project.

Code-based Suppression Syntax

Annotations in code can define locations, message sets, and suppressions. Location tags are useful to tag specific locations that may be the subject of suppressions defined elsewhere. Re-use of a location tag can be used to group locations. Suppression entries are entered in C or C++ comment forms, and are prefixed with the string

PRQA.

There is a full explanation of the entire syntax in Appendix H: Code-basedSuppressions, but the following are some typical entries and their effects.

Suppress message 100 on the current line:

```
int i; // PRQA S 100
```

Suppress message 100 for 2 following lines (suppresses i and j declarations):

```
// PRQA S 100 2
int i;
int j;
int k;
```

Same equivalent suppression using a tag:

```
// PRQA S 100 L1
int i;
int j;    // PRQA L:L1
int k;
```

Suppress message 100 to end of file (suppresses j and k declarations):

```
int i;
int j;    // PRQA S 100 EOF
int k;    // all following messages 100 suppressed
```

Suppression of Warnings with #pragma

QA C recognises several #pragma identifiers in source code which can be used to control the display of messages.

For example:

```
#pragma PRQA_MESSAGES_OFF 1234, 2443, 1256-1258
```

will suppress the specified messages for the remainder of the current translation unit. They will be suppressed until they are re-enabled. For example:

```
#pragma PRQA_MESSAGES_ON 1234
```

will re-enable message 1234 but will not affect the status of other messages.

Notes:

If you do not specify any message numbers in the `#pragma`, all messages will be switched on or off as appropriate. You can choose to display these suppressed warning messages later by choosing Message Personality>Display>Show Suppressed Warnings.

You will not be able to suppress messages that belong to level 9, as these errors affect the quality and scope of analysis, and need to be rectified prior to examining messages at other levels.

In addition to these suppressions that apply at their points in code, there is a special `#pragma` directive which caters for suppression of macro expansions:

```
#pragma PRQA_MACRO_MESSAGES_OFF "mymacro"
```

will suppress all messages in any expansion of `mymacro`.

```
#pragma PRQA_MACRO_MESSAGES_OFF "mymacro" 3344
```

will suppress message 3344 in any expansion of `mymacro`.

Finally, there are two additional `#pragma` directives which, although not controlling suppression, do provide assistance for certain aspects of analysis.

```
#pragma PRQA_NO_RETURN foo
```

identifies that function `foo` does not return. This functionality is used to assist control flow analysis and value analysis.

```
#pragma PRQA_NO_SIDE_EFFECTS foo
```

identifies that function `foo` can be treated as “free of side effects”. By default, QA C considers execution of a function call to be a source of side effects. Use of this `#pragma` may significantly affect the generation of certain messages relating to side effects (e.g. 3415, 3416, and 3446).

Project Configuration File

A single project configuration file can be defined for each project, and is specified through the Project Properties menu item. This file is intended to contain specialized settings that are relevant for various analysis and reporting processes. The formats of settings within this

text file are determined by the needs of various analysis and reporting functions, including custom-supplied components.

The standard installation of QA C does not yet require this additional configuration file, but it is possible to use this file to make project-wide configuration entries, which will then override any specific configuration in personality files. This would have to be accomplished by manually creating the appropriate entries in the project configuration file.

When QA C is extended to match a customer's process or compliance environment, this project-based configuration mechanism can support the delivery of such items as:

- Message suppression according to complex location or parameter rules,
- Custom reporting commencement from nominated paths, functions, or project trees,
- Project-specific analysis directives.

Personalities and Analysis

The Compiler and Analyser personalities contain configuration options that determine the content of the analysis output that is written to the `.err` and `.met` files. If you change a configuration option in either of these personalities, you will need to re-analyse your source file to generate updated `.err` and `.met` files.

When you analyse a file, QA C will generate messages in the `.err` file regardless of whether they are suppressed in the message personality. Suppression of messages in the Message personality only affects the display of the warning messages in annotated source code.

As a result, if you alter message filtering or display options in the message personality, you will not need to re-analyse your source file when you are generating annotated source code. However, if you are viewing the Warning Listing report, you will need to re-analyse in order to force QA C to update the HTML annotated source code files.

Application Options

In Options from the Configuration menu you can set various QA C application options. These apply to all projects.

Choosing Your Editor and Browser

The installation process will attempt to configure QA C so that source code is inspected and edited using your usual text editor and HTML reports are viewed in your default browser. These settings can be adjusted manually on the Editor Preferences tab.

If you are specifying an external text viewer, and the text editor/viewer permits it, you can use the following format specifiers in the Additional Parameters field:

- %L: the line of the source file to which the viewer will navigate,
- %F: the name of the source file.

The addition of line number formatting in text viewing enables automatic navigation to individual lines of your source file from the various source structures and metrics browsers in the product, for example showing the point of definition of named identifiers.

With the HTML browser, you can select Use DDE Communication in order to make HTML display requests use an available open browser window. This option should only be used with Internet Explorer, which supports the facility. Deselecting this option will launch a new browser window for each HTML annotated source request.

Changing Your Annotated Source Settings

Generation of Annotated Source Code

When you inspect annotated source code, QA C will generate an appropriate file (.txt or .html) in your output directory as required. However, these options allow you to generate annotated source listings automatically during analysis. This may consume a lot of disk space but can be useful to:

- provide a complete record of warning messages in context,
- generate the HTML links which are available in the Warning Listing report.

Selecting “Regenerate annotated source on each view request” will always regenerate the annotated source code when you choose to display annotated source code. If you deselect it, QA C will compare the `.err` file and the message personality file with the annotated source code file. If the annotated source code file is older than either the `.err` file or the message personality file, it will be regenerated before it is displayed. If it is newer, QA C will not regenerate the file.

Annotated Source Display

These options represent the four alternative responses when you double-click or press enter with a file selected. The default and recommended option is View Message Browser, as this component will display messages for a complete project in an intuitive and interactive display. If you select View Annotated Source or View HTML Annotated Source the annotated source code will be regenerated if you have selected “Regenerate annotated source on each view request”.

Setting the Housekeeping Options

QA C's housekeeping options determine the time period for which your output files are kept. The housekeeping options are defined by the File Deletion Settings on the Project File Options tab. To change the housekeeping options:

1. In Annotated Source Files, set the time period after which annotated source files (`.txt` and `.html`) are deleted.
2. In Analysis Output Files, set the time period after which output files (`.err`, `.met`, and `.i`) are deleted.

Note:

The Clean Up Now button will delete all annotated source and analysis output files that are older than the time period specified in Delete Beyond.

3. In Run File Deletion, set the time when file deletion runs. File deletion can be run when a project opens or closes.

Note:

You can also delete output files manually through the menu item Remove Analysis Output, available by right clicking on a source file selection or from the Analyse menu.

There is a 3-second allowance made for network timestamp differences in deciding when to delete output.

4. In Project Autosave Settings, set the time period after which QA C++ will automatically save changes to your project.
5. In License Auto-release, choose to force application closedown in the event of 30 minutes of system idleness. When the timer counts down to zero, the license is released, and a dialog is presented to allow you to reclaim a license if available.

Note:

The Message Browser also contains a 30-minute idle application closedown, which is triggered on application actions.

CHAPTER 4

Analysing Source Code

You can analyse a project, folder, or selection of files. The Analysis Status window displays the progress of each file. Analysis operates on single files, through primary and secondary analysis, and on the whole project, through CMA analysis.

You can choose to Pause or Cancel the analysis. When you Pause, the analysis of the active source file will be suspended. When you Resume, analysis will continue from that point. If you Cancel, analysis of the current file is aborted. If you then Resume, analysis will start on the next file.

If you wish to Cancel a queue of analysis files, there is a menu item and associated button, Clear Queue, for this. You can also Pause or Cancel the current file analysis prior to or after this action.

When a new set of source files are presented for analysis, previously successfully completed analysis entries are automatically removed from the Analysis Status window list. This does not apply for CMA analysis. Files that terminated without a successful “Completed” status, which will usually happen because of configuration problems or some other system failure, remain in the list and must be removed manually. See Appendix E: Program Return Codes for a list of these return codes and text entries.

When a file has been analysed successfully, a red tick in the main QA C window accompanies the name of the file. The red tick will continue to appear as long as the `.err` file, produced during analysis, is newer than the source file.

The ticked status of files represents the logic that:

```
output files are newer than source
and
output files are newer than folder personality files
```

The menu item Refresh on the Edit menu list will manually refresh the ticked status of all files in the selected folder. When you select a new

folder, this action is automatically performed.

Commencing Analysis

You can analyse single files, sets of selected files, folder branches, or the complete project (equivalent to folder analysis from the root folder).

There are three menu actions within the Analyse menu list, covering each of these choices. An additional selection determines whether CMA analysis is run at the end of the analysis of all files in the project. See Cross-Module Analysis later in this chapter.

The Analyse speed button will perform analysis according to the current file or folder selection. If one or more files are selected in the file listing pane, it will perform the menu action Analyse Selected Files. If any folder other than the root folder is selected it will perform Analyse Folder. If the root folder is selected it will perform Analyse Project. The “hover help” on this speed button will also adjust based on the file/folder selection.

Analysis Output Files

During analysis, QA C creates a number of output files. These files have the same name as the source file but with the addition of further file extensions. For example, the `.met` file for `source.c` is generated as `source.c.met`.

QA C always outputs the following files:

<code>.err</code>	detected warnings
<code>.met</code>	parsing data

QA C outputs the following files, depending on your configuration:

<code>.i</code>	preprocessed source code
<code>.txt</code> or <code>.html</code>	annotated source code

It is good practice to delete out-of-date output files by setting the

housekeeping options in Options>Project File Options. You can manually remove all output files for a file or folder by selecting the file or folder and choosing Remove Analysis Output from the Analyse menu. You can also manually apply housekeeping checks on output files for a file or folder by selecting the file or folder and choosing Check Analysis Output from the Analyse menu. For both of these actions, folder selections will also ripple to sub-folders.

Note:

The project output files, generated from cross-module analysis (see later in this chapter), are deleted whenever you commence file analysis, taking a cautious approach to the validity of their contents.

.err Files

The `.err` file contains a record of all warnings that were detected in the source code and included header files. However, when you display annotated source code, you will only see the subset of messages that you have chosen in the Message Personality. These warnings are displayed as a listing of source code interspersed with warning messages which is referred to as annotated source code.

Further warnings may be added to the `.err` file through secondary analysis. This is described in Configuring Secondary Analysis later in this chapter.

.met Files

The `.met` file contains data from the parsing process as described in Appendix F: Metric Output File. The data from this file can be used in reports and with the various browsers.

You can inspect the contents of a `.met` by choosing Metric Output File from either the View menu or the pop-up menu.

Note:

`.met` files may be automatically deleted according to your File Deletion Settings. These settings cause output files to be deleted after a specified period in order to conserve disk space.

.i Files

The `.i` file contains preprocessed source code. This file will only be generated if you have enabled Preprocessed Output in the Analyser Personality. Preprocessed source code can be viewed by choosing Preprocessed Source from the View menu or by clicking Preprocessed Source button.

Preprocessed source code can be useful when investigating warnings generated in header files or in the expansion of macros. Annotated source code will only show you warnings in the context of your source file, which may not provide sufficient information to diagnose the root of the problem.

One way of doing this is to generate a preprocessed file and then to analyse the preprocessed file itself by adding it temporarily to your project. You will then be able to see all the code from header files, and all macros will be fully expanded.

Comments are always stripped from preprocessed code. However, QA C has a feature (Include filename and line numbers in preprocessed listing) whereby new comments are introduced in the preprocessed file which show the source or header file name and line numbers. This allows you to see the origin of the code. See Analysis Processing of Appendix A: Personalities.

For example:

```
#include "demo.h"
int step1(int x)
{
    int r = 0;
    if (glob > 10)
    {
        r = x * x;
    }
    return(r);
}
```

Preprocessed code without comments:

```
extern int glob;
extern int step1(int x);
extern int lesser(int y);
int step1 ( int x )
```

```
{
    int r = 0 ;
    if ( glob > 10 )
    {
        r = x * x ;
    }
    return ( r ) ;
}
```

Preprocessed code with comments:

```
/*
 * C:\Program Files\PRQA\QAC\demo\demo.h
 */
/*      1 */ extern int glob;
/*      2 */ extern int step1(int x);
/*      3 */ extern int lesser(int y);
/*
 * C:\Program Files\PRQA\QAC\demo\prep.cpp
 */
/*      2 */ int step1 ( int x )
/*      3 */ {
/*      4 */ int r = 0 ;
/*      5 */ if ( glob > 10 )
/*      6 */ {
/*      7 */ r = x * x ;
/*      8 */ }
/*      9 */ return ( r ) ;
/*     10 */ }
```

.txt and .html files

When you display annotated source code, a .txt or .html file is used. The file will be generated when required but is usually generated automatically during analysis. If you want to use the Warning Listing report, it is necessary for .html files to be present in advance in order for the link references to work. If you want to use this report or to generate annotated listings as a record of the analysis, set the necessary options to generate the output files automatically. See Changing Your Annotated Source Settings of Chapter 3: Configuring QA C.

Managing Output Files

When you analyse many source files, the volume of data written to the output files can become very large. You should take care to control the amount of data generated in these files.

Automatic Generation of Annotated Source Files

QA C can generate an HTML annotated source file whenever you analyse. This file is then used by the Warning Listing report. If you do not intend to use this report, you can disable the automatic generation of these files to conserve disk space. See Changing Your Annotated Source Settings of Chapter 3: Configuring QA C.

Suppressing Analysis Output from Header Files

Often much of the data in `.err` and `.met` files is associated with header files and may be of marginal importance. If the header file has been thoroughly tested, the warning messages will no longer be of interest and the analysis data is usually only relevant if you are performing additional secondary or CMA analysis checks.

By choosing not to generate analysis output from library header files, you can often greatly reduce the size of the `.err` and `.met` files. To do this, choose Suppress Output when you specify a system include path in the Analyser or Compiler Personality. If you choose to suppress analysis output from a header file, you will only see level 9 (Errors) warning messages for this file in annotated source code.

If the header file is directly associated with source, you may want to generate messages from the header file and then choose to suppress the output in the annotated source. This is described below.

Suppressing Messages from Header Files

There are two ways of suppressing the display of header file warning messages. The first way is to suppress the output data from being written to the `.err` file, as described above, and should only be used when you are satisfied that the warning messages from the header files are of little importance.

The second method is to suppress the display of header file warning messages in the annotated source code. This is recommended for your project headers, if you are satisfied that they are stable and free from major problems.

To do this, ensure that you have not chosen Suppress Output for your header file paths in Analyser Personality>Project Headers>Header Includes. All header files included in these paths will be analysed and the messages will appear in the annotated source code.

The display of header file warnings is controlled by Message Personality>Display>Display Header Warnings. By default, QA C displays header file warnings in annotated source code, since some analysis failures have their origins in incorrect parsing or navigation through associated header files.

Secondary Analysis

Secondary analysis is the process by which additional checks may be introduced to supplement the QA C file analysis. Typically, a script or program can make use of the parsing information in the `.met` file to generate additional warnings.

Configuring Secondary Analysis

If you have a secondary analysis program that you wish to run:

1. Open your Message Personality.
2. On the Advanced Settings tab, click Setup in the Secondary Analysis Command String.
3. In the Secondary Analysis Settings dialog box, click Enable Secondary Analysis Processing.
4. Enter, or browse for, your secondary analysis program.
5. If relevant, enter, or browse for, your secondary analysis script.
6. In Command Line Parameters, specify the argument string to be supplied to the secondary analysis program.

Parameter specifiers can be used to pass relevant information to the secondary analysis executable. These parameters are described below. The resulting command line string will appear below in the Combined

Command String box.

%Q Specifies the product identifier (QAC). This is frequently used by product utilities such as the `errwrt` utility. See Writing Messages to the Error File of Chapter 10: Advanced Topics.

%P[+] `[-via] settings.via`

Creates and passes a temporary configuration file, containing a combined set of all the selected folder's personality settings.

%F The source filename.

%S The script filename, if supplied in the Script or Naming Rule Configuration File box.

%N[+] `[-nrf] <name rule file>`, if supplied in the Script or Naming Rule Configuration File box. See Naming Convention Checking below.

7. If your Shell environment requires that the file paths use forward slashes (Unix style) as separators, click Convert directory paths in parameters to Unix format.

Note:

QA C processes return codes from secondary analysis programs. Successful return codes are 0 or 1. Any other return code will be displayed in the status column.

Naming Convention Checking

There is a special secondary analysis program that is supplied in the product to deliver checking of identifier names. Appendix I: Naming Convention Checking provides full details on how to set up your naming convention rules and its behaviour.

To run naming convention checking, define your secondary analysis program as follows:

1. From the Message Personality, Advanced Settings tab, click Setup in Secondary Analysis Command String, and click Enable

Secondary Analysis Processing.

2. Select `pal.exe` (from the product bin directory) as your secondary analysis program.
3. In the Script or Naming Rule Configuration File, enter the rule file for your naming convention.
4. In Command Line Parameters, specify the argument string as:
`%Q %N+ %F`

Running Multiple Secondary Analysis

If there is more than one secondary analysis program to be run through the GUI, then a special strategy is needed.

The GUI must invoke a script file that runs each Secondary Analysis task in sequence. This script must be passed all parameters for each Secondary Analysis task, and then execute the analysis supplying the correct parameters. The script should also check the return code of each analysis and take appropriate action if there is a problem. Finally if there are no problems with any of the analyses then the script should exit with a zero.

The following is an example batch process for this. There can be many variations on this, depending on the project – this is intended as a suitable starting point, which can then be customised to suit the individual projects. Indeed, it is permissible to write a C or C++ executable to start the Secondary Analysis tasks.

The batch file has to be run from cmd.exe. The personality settings in the Secondary Analysis Settings dialog are:

Program executable:

cmd.exe (typically c:\windows\system32\cmd.exe)

Script or Naming Check Configuration File:

Select the script described below.

Command Line Parameters:

/c "%S %Q %F"

The expanded command is displayed in the 'Combined Command String', and will look something like:

```
"C:\WINDOWS\system32\cmd.exe" /c  
""C:\work\sec_anal.bat" QAC "<filename>""
```

Note the double quotes enclosing the entire string argument to /c. This tells cmd.exe to run the command specified in the string and then to terminate.

The batch file (in this example C:\work\sec_anal.bat) will look something like:

```
@ECHO OFF  
set RESULT=0
```

```
"C:\Program Files\PRQA\QAC\bin\qacuser.exe" %2
if ERRORLEVEL NEQ 0 (
    set RESULT=%ERRORLEVEL%
)
"C:\Program Files\PRQA\QAC\bin\pal.exe" %1 %2
if ERRORLEVEL NEQ 0 (
    set RESULT=%ERRORLEVEL%
)
exit %RESULT%
```

Cross-Module Analysis

Certain aspects of source code analysis require complete project information. For example, an accurate analysis of indirect function recursion can only be performed across all source files containing function definitions. Cross-Module Analysis (CMA) allows you to execute this type of additional analysis on all the source files within your project.

CMA is usually performed on the collection of `.met` files generated for all source files in a project. Analysis output from CMA is normally generated into a special project `.met` or `.err` file. For example:

- Certain metric calculations that depend for their calculation on the collected individual file output can be saved into the project `.met` file.
- Function recursion that results from indirect calls across modules can be identified and recorded in the project `.err` file.

Configuring Cross-Module Analysis

Cross-Module Analysis can be configured by either choosing Options>Cross-Module Analysis from the Configuration menu or Cross-Module Analysis>Configure from the Analyse menu. To configure Cross-Module Analysis:

1. In the Cross-Module Analysis dialog box, click Add.
2. In Program Executable, browse for your project analysis program.

3. In Additional Parameters, specify the arguments to be supplied. The default expansion parameters are:

```
%Q %P+ %L+
```

The final command string will appear in Combined Command String and, in the above case, will expand to:

```
program QAC -via <personality_files> -list
<filelist.lst>
```

Original entries above expand out as, and correspond to:

Program Your CMA program.

QAC A fixed label representing the application that generated the analysis data, in this case QA C.

%P[+] [-via] <personality_files>

The `-via` option passes the complete set of root folder personality files to the program executable, as well as the project configuration file, if defined.

Note: Because of this root-folder restriction, it is unwise for CMA analysis to depend on personality settings specific to sub-folders in the project.

%L[+] [-list] <filelist.lst>

The `-list` option passes a list of files to be read by the program executable. Use of this option will create a file listing in the temp directory, and specifies a list of all source files in your project with their corresponding output directories. See Appendix B: Configuration Options.

Note: This file listing always includes the special entry
`-cmef <root_folder>`
 which specifies the output files, into which CMA err and met information are generated (see below).

CMA Output Files

The output of CMA analysis is placed into two predefined filenames, for both message records (`.err`) and metrics and additional semantic

information (`.met`).

The filename is formed from a special entry supplied to the CMA program

```
-cmaf <CMA_Output_File>
```

This entry is supplied along with the `filelist.lst` entry in the GUI (see above).

The parameter to `-cmaf` is constructed by the GUI as the name and output path of the root folder. Message records generated by CMA analysis are created into a file formed from this base filename with a `.err` extension, and metric and other textual information into an equivalent file using the `.met` extension (e.g. `root_folder_name.met`).

Running Cross-Module Analysis

To perform CMA analysis, choose Cross-Module Analysis>Run from the Analyse menu.

You can also set CMA to run automatically after analysis of all source files in a project. To configure this option, select Cross-Module Analysis>Include CMA in Project Analysis from the Analyse menu. With this menu option ticked, when you then run Analyse Project from the Analyse menu, it will include CMA analysis as a last step.

When running each step of CMA, the output window will show its progress, and will provide separate completion status for each step defined. You can also Pause or Halt each of these CMA steps individually.

Note:

The project output files, generated from CMA, are deleted whenever you commence file analysis, taking a cautious approach to the validity of their contents.

Default-supplied CMA Settings

QA C is shipped with a CMA program named `pal.exe` (PAL stands for Post-Analysis Launcher). This program includes various items of cross-module analysis, including:

- **Unused External Identifiers:** A check is performed on all globally visible declarations and definitions across your project to identify any that are never used, or only referenced within a single module and therefore do not need to have external linkage.
- **Function Recursion:** Some coding standards prohibit function recursion because of the dangers of stack overflow. Indirect recursion, across a series of function calls in your project, is identified by this check.
- **Declaration Cross Checks:** There are a variety of declaration checks which are in line with coding standard advice. These cover conflicting typedefs, namespace function, object and type declarations or definitions.

After running this CMA analysis module, the project `.met` and `.err` files are created or replaced. They will contain new message records and certain metric values.

These files are used by various browsers; the `.met` file when displaying metrics, and the `.err` file when displaying analysis messages in the **Message Browser**.

Analysis Log

You can save a log of analysis results from the Analysis Status Window. From the menu, select **Save Analysis Log**. In the following dialog, choose a save location. It will create a `.csv` file to save the log, and will include the status of all files listed in this window.

CHAPTER 5

Viewing Analysis Results

Annotated source code is a listing of source code interspersed with QA C warning messages. The format of the listing can be controlled in a number of ways but the primary function is always to present the warning messages in context. A ^ symbol directly underneath the source line indicates the symbol or identifier to which the warning applies.

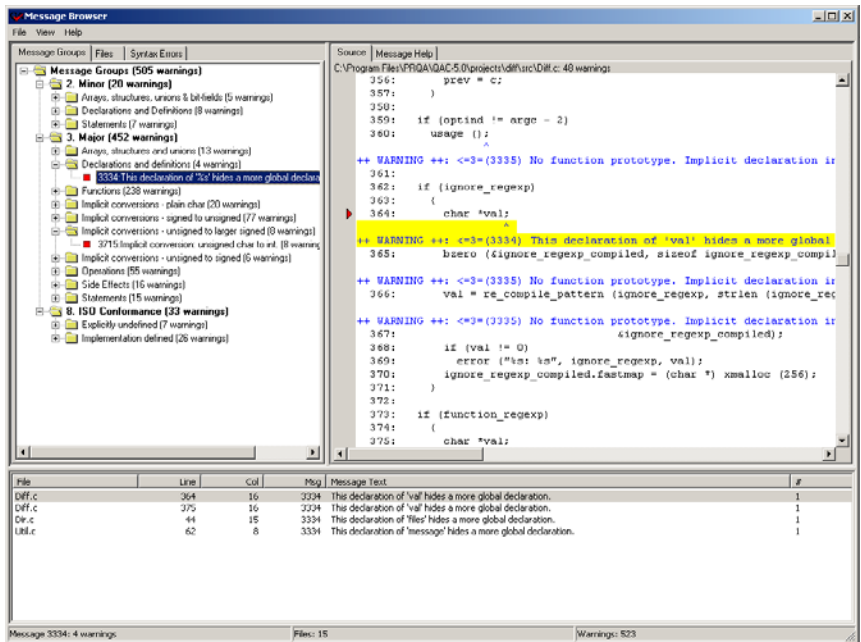
There are several ways to view analysis results from QA C. The Message Browser is a standalone component that captures and organises analysis results across a complete project. A simpler annotated source display on a file-by-file basis can also be obtained in conjunction with your own editor or browser.

The Message Browser

The Message Browser displays annotated source code for one or more source files, a folder, or an entire project. For each file that you have chosen, the Message Browser reads the contents of its `.err` file and displays its warning messages according to the settings of the Message Personality.

If you are launching the Message Browser for a project, or for a folder with sub-folders, the annotated source code will be displayed in accordance with the options specified in the Message Personality of the head folder of the branch.

The Message Browser includes the project `.err` file in all activation from the GUI, whether single files, groups of files, folders, or root folder (entire project).



The Message Browser can be launched in two ways (see also Chapter 2: Making File Selections):

- by selecting one or more files and choosing Messages from the Browse menu or Message Browser from the pop-up menu,
- by selecting a folder and choosing Messages from the Browse menu or Message Browser from the pop-up menu (messages will be displayed for the selected folder and all sub-folders).

Messages will only appear if they have been enabled in the Message Personality. If a message level has been enabled but no warning messages from that level were detected, the message level will not be displayed. To change the selection of enabled messages or the formatting of displayed messages, you will need to close the Message Browser and then re-launch it with the revised message subset.

If you choose to re-analyse files which are on display in the Message Browser, you will need to choose Refresh Warnings from the Message Browser's File menu to load the contents of the new `.err` files.

Note:

If you have syntax errors in your source code, the Message Browser will warn about their presence when it opens. Normally it is advisable to resolve these issues before looking at other warning messages.

The message browser consists of three panes:

- Source View, the top right pane,
- Warning Summary View, top left pane,
- Warning List View, the bottom pane.

The Source View

The Source View is the main pane of the Message Browser. This pane consists of a tab containing annotated source code and a tab containing help text for the selected warning message. The name of the source file is displayed at the top of the Source tab.

When you select a warning message in either of the other two panes (Summary View or Warning List View), the annotated source code will

be positioned at the location of the warning. The message will appear highlighted with a caret (^) indicating the symbol or identifier to which the warning applies.

Only warning messages from the message subset of your Message Personality will appear in the Source View. To expand your message subset, you will need to enable additional messages in the Message Personality. Re-launching the Message Browser will display the new message subset.

If you have re-analysed the files included in the Message Browser, you can choose Refresh Warnings from the File menu to load the new set of warning messages.

Messages originated from CMA Analysis will also be displayed in the Message Browser. These messages make extensive use of the Context Message feature mentioned below.

Context Messages

Some analysis messages are caused not only by the line of source code that they point to, but also by earlier code. For example, a hiding declaration occurs because an identifier is re-declared later on in the code, and to understand the issue fully, both the hiding and the hidden declarations should be pointed out. There is a notion of a subordinate message, which provides additional detail for the main message. Such a context message can point to a different location in the code (location-based), or provide more specific information than the main message and have the same location (information-based).

Because information context messages have the same location as their parent message, they will be displayed below the parent message in annotated source code produced by the Message Browser.

Clicking on (selecting) a location message will move the focus of the annotated source to that location. Use the Back button (see below) to revert to the parent message.

The menu item View | Fold Context Messages toggles the default state of messages with context messages between folded or unfolded. There is also a Fold All/Unfold All toggle menu option to

expand/collapse the messages.

Navigating Through Messages

You can navigate through your source code by right-clicking a warning and choosing Show Previous Warning or Show Next Warning from the pop-up menu. These same commands are available through the back (left) and forward (right) arrow buttons at the top of the Source View, and also through F6/F7 keys.

Navigations are remembered, which enables the user to click back through the history of messages selected. New clicks are inserted into the history location, enabling the user to click forward after going back and then visiting a different location. Refreshing the warnings will remove all click history.

If header warnings are enabled, diagnostics associated with a header file may be repeated for a number of different translation units in which the header file is included. These diagnostics are merged and the number of duplicates is shown in the Warning List View, the pane at the bottom of the message browser.

Displaying Message Help

Select a message and click the Message Help tab or double-click a warning message. In the Source View, the second tab will be brought to the front and the additional explanation text and examples from the message html will be displayed.

Saving Annotated Source Code

Choose Save as Text or Save as HTML from the File menu. The file currently on display in the Source View will be saved in the specified format.

Displaying Line Numbers

Choose Display Line Numbers from the View menu.

Warning Summary View

The Warning Summary View appears to the left of the Source View and displays a list of all detected warning messages for each message group or file. There are four possible tabs to display in the Warning Summary View:

Message Groups

The Message Groups tab lists all detected messages according to message level and group. The number of diagnostics for each message group and level is displayed alongside in brackets.

To go to the first occurrence of a warning message, click a message group or level. The annotated source code in the Source View will jump to the first message occurrence, and all message occurrences will be listed in the Warning List View.

Files

The Files tab lists all diagnostics, grouped by file. The number of diagnostics in each file is displayed in brackets beside the file name. Files selected for display but with no detected warning messages will not be included.

The listing is split into two headings: Source Files and Header Files. Header file warnings will only appear if you have selected Display Header Warnings in your Message Personality and you have not suppressed warnings from the relevant include paths in the Compiler and Analyser personalities.

To go to the first diagnostic in a file, click the file. The annotated source code for that file will appear in the Source View and a listing of all diagnostics for that file will appear in the Warning List View.

CMA Results

The CMA Results tab contains all messages from cross-module analysis in one place, as a convenience for discovering the issues associated with this phase of analysis. This will include multi-homed messages as well as those directly associated with a source file.

Syntax Errors

The Syntax Errors tab will contain any syntax errors that were encountered during analysis. It is normally advisable to resolve all syntax errors before attempting to address other messages.

Warning List View

The Warning List View is displayed across the bottom of the Message Browser. It shows a listing of all diagnostics corresponding to the file or message group selected in the Warning Summary View.

If you have selected a message group on the Message Groups tab in the Warning Summary View, the set of diagnostics for the selected group is listed in the Warning List View. If you have selected a file on the Files tab, the set of diagnostics for the selected file is listed.

When you select each diagnostic, the annotated source code in the Source View will reposition to the line on which the warning was detected.

Viewing Annotated Source Code

Annotated source code can be viewed in text format or as HTML. The HTML version includes links to expanded explanations of the warning messages and cross-references to related warnings.

You can view annotated source code by selecting a source file and using either:

- a toolbar button,
- the appropriate option on the View menu,

- the Messages option on the Browse menu,
- the pop-up menu available from a right mouse button click.

You can also choose to view annotated source code using a double click or the enter key on a selected source file. This behaviour is controlled by Annotated Source Settings of Configuration>Options. You can choose to view the original source code file, one of the two annotated source code formats, or launch the Message Browser.

QA C also provides a Warning Listing report to view a summary of all the warning messages in your source code. See Warning Listing in Chapter 6: Reports.

Context Messages in Annotated Source

Annotated source output will also display *information-style* context messages (see Context Messages section above). Due to the in-built limitations of this form of display, *location* context messages have limited display capability, showing just the other location's filename and line number along with the message text.

This limitation does not affect the Message Browser, and location messages will be displayed below their parent message.

Expand Multi-homed Messages

An option exists to generate an instance of a multi-homed message in each of its sub-locations, rather than generating a single message at the primary location.

The purpose of this option is to highlight, directly in code annotated source, the problems (or the contributing incompatibilities) associated with that source file. The list of messages so defined is configured in the Message Personality, Advanced Settings Tab, Message Expansion List.

Changing the Format of Annotated Source

During analysis, QA C generates a `.err` file containing a record of all

warnings detected in the source code. The Message Personality allows you to suppress the display of individual messages or complete message levels and to specify the format in which the warnings are presented. If you change an option in the Message Personality it is not necessary to re-analyse the source file.

There are three tabs:

Warning Messages	Messages can be suppressed individually or by message level.
Advanced Settings	A user message file can be specified to introduce your own messages.
Display	Control of the display format.

See Appendix A: Personalities for more information on configuring personalities.

Using Annotated Source Code

In the following section we discuss a small C program. First the annotated source code listing is reproduced with some extra comments added in *italics*, to expand on the meaning of the warning messages. Then we show a fully commented source file with all these issues corrected.

Sample Annotated Source Code Listing

The following is the original source code with annotations:

```
1:  #include<stdio.h>
2:  #include<string.h>
3:  #include<stdlib.h>
4:
5:
6:  main(int arg,char *argc[])
7:  {
    ^
++ WARNING ++: <=2=(3114) 'main()' has been declared
without an explicit return type and therefore should end
with an explicit return statement with an 'int' value.
```

main() is implicitly declared with type int but lacks an explicit return(expression) or exit(expression) statement at the end of the function.

```

      ^
++ WARNING ++: <=3=(2050) The 'int' type specifier should
not be omitted from declarations.

```

The int specifier should be present when declaring the main function. It is an important requirement and good programming practice as this is the return type of the main function. It would be required if the code was compiled with a C++ compiler.

```

8:      char      filename[30];
      ^
++ WARNING ++: <=2=(3132) Hard coded 'magic' number '30'
used to define the size of an array.

```

Specifying numbers within the code is generally bad practice from a maintenance point of view, particularly if they are widely used within the code. In this example however, it is only used once. If 'gets(filename)' on line 16 was replaced by a safer input function, that function would require the size of 'filename'. The 'magic' value would then be used in two places.

```

9:      unsigned int class;
      ^
++ WARNING ++: <=2=(1300) 'class' is a keyword in C++.

```

An obvious potential problem for C++ programmers, not so obvious for a C programmer porting to C++.

```

10:     unsigned int c2,entry,spaces,lines,count;
      ^
++ WARNING ++: <=2=(3615) 'entry' was a keyword in K&R C.
      ^

```

A not so obvious variable identifier which is a keyword in K&R C.

```

++ WARNING ++: <=2=(3205) The variable 'count' is not used
and could be removed.

```

The variable "count" is not used within the code. It would be appropriate to remove it as it is not needed.

```

                                ^
++ WARNING ++: <=2=(2211) 'entry' is not aligned with the
previously declared identifier.

```

Variable declarations are badly aligned. This is a stylistic issue, but if identifiers are aligned properly, it makes your code easier to read.

```

11:     int Miscchar[255];
12:     FILEE *fp;
                                ^
++ WARNING ++: <=3=(3112) This statement has no side-effect
- it can be removed.
++ ERROR ++: <=9=(0434) [C] 'FILEE' is not declared.
++ ERROR ++: <=9=(0434) [C] 'fp' is not declared.

```

A typing error. 'FILEE' should be 'FILE'. This small problem invalidates the whole declaration.

```

13:
14:     printf("Word Counting Program V1.0 \n");
                                ^
++ WARNING ++: <=2=(2201) This indentation is not
consistent with previous indentation in this file.

```

The indentation is not consistent with the previous indentation in the source. It is good practice, and often part of programming standards or guidelines to keep indentation consistent, as it makes it easier to read the code and follow the logic.

```

15:     if(arg==1) {
                                ^
++ WARNING ++: <=2=(2209) This brace style is not
consistent with 'exdented' style.

```

Exdented bracing has been selected in the tool (Analyser Personality) configuration. The code layout does not follow this rule.

```

16:     printf("\nPlease type in text file name : ");
gets(filename); }
                                ^
++ WARNING ++: <=3=(2010) The function 'gets()' must not be
called.

```

'gets()' does not allow the size of the buffer, in this case 30, to be given and will write past the memory allocated for 'filename' if the

string entered is longer than the buffer. It should only be used where the string entered is guaranteed not to exceed the size of the given buffer. This warning is generated by declaring function gets in the warncalls section of the analyser personality.

```
++ WARNING ++: <=2=(2205) More than one declaration or
statement on the same line.
```

There is more than one statement on the same line. This is considered to be bad practice and prohibited by many programming standards.

```
^
++ WARNING ++: <=2=(2203) This closing brace is not aligned
appropriately with the matching opening brace.
++ WARNING ++: <=2=(2205) More than one declaration or
statement on the same line.
    17:  else
    18:      strcpy(filename,argc[1]);
        ^
++ WARNING ++: <=2=(2212) Body of control statement is not
enclosed within braces.
    19:  printf("\n");
        ^
++ WARNING ++: <=2=(2201) This indentation is not
consistent with previous indentation in this file.
    20:  fp=fopen(filename,"r");
        ^
++ ERROR ++: <=9=(0557) [C] Right operand of assignment
should have arithmetic types.
    21:  if(fp==0) { printf("Can't open : %s",filename);
    22:                  exit(1); }
    23:  c2=-1;
++ WARNING ++: <=3=(3760) Implicit cast: int to unsigned
int.
```

An implicit cast is being performed. -1 is being assigned to an unsigned integer, which is not what the programmer intended.

```
^
++ WARNING ++: <=2=(3501) This could be read as an old-
style assignment operator. Spaces should be used to make it
more readable.
```

A programmer reading this line might interpret this incorrectly, since older compilers take this to be the same as the '-=' operator. Spaces would be advisable.

```
24:  spaces=0;
```

```

25:  lines=0;
26:  while(class = getc(fp) !=EOF)
++ WARNING ++: <=2=(3314) This control expression is an
assignment. It would be better practice to test the result
against zero.

```

The compiler reads the expression as (class = (getc(fp) != EOF)). The programmer intended to say ((class = getc(fp)) != EOF)).

```

++ WARNING ++: <=2=(3416) This boolean expression contains
side effects.

```

This expression has side-effects. The assignment should be performed outside of the comparison. Side-effects like storing the value returned by 'getc(fp)' in 'class' do not have to take place at the same time that expressions are evaluated. This can be dangerous programming. In this case the value returned by the assignment is the value to be assigned to 'class', as intended.

```

                ^
++ ERROR ++: <=9=(0432) [C] Argument should be compatible
pointer type.

```

```

27:  {
28:      if (class < 0 )
                ^
++ WARNING ++: <=3=(3316) Dangerous comparison of unsigned
data with zero.

```

Evaluating whether an unsigned integer is less than zero is meaningless, it never can be. The logic needs to be reexamined.

```

29:      break;
                ^
++ WARNING ++: <=2=(2212) Braces should be used in control
statements, even when the body has only one statement.

```

Again, no braces as part of a control expression.

```

++ WARNING ++: <=2=(3333) A break statement shall only be
used at the end of all the statements in a switch case
block.

```

```

30:      if(class<=47 && class>=0 || class<=126 &&
class>=123)
++ WARNING ++: <=2=(3401) Possible precedence confusion:

```


extra parentheses are recommended here.

Lack of parentheses in a complex expression, may result in the expression being evaluated in way that the programmer did not intend.

```

++ WARNING ++: <=3=(3324) An unsigned value is always
greater than or equal to zero - this test is always true.

```

A redundant test.

```

31:    if(c2>=48 && c2<=122)
32:        spaces++;
33:    if(class>=128 || class<=31)
34:        { entry++; Miscchar[class]++; }
++ WARNING ++: <=3=(3321) The variable 'entry' may be unset
at this point.
++ WARNING ++: <=3=(3321) The variable 'Miscchar' may be
unset at this point.

```

Variables 'entry' and 'Miscchar' were not initialised with a value, hence their values were initially undetermined. The compiler will have just assigned memory locations for them, and the contents of these memory locations will be indeterminate.

```

35:    if(class==10){ lines++; entry--;}
36:    c2=class;
37: }
38:
39: printf("File : %s\n",filename);
40: printf("Length of file : %u
bytes\n",_filelength(_fileno(fp)));
41: printf("Number of lines in file : %q\n",lines );
++ WARNING ++: <=8=(0160) Using unsupported conversion
specifier number 1
++ WARNING ++: <=8=(0185) Call contains more arguments than
conversion
specifiers
42: printf("Number of words in file : %i\n",spaces);
43:
44: if(misc > 0)
++ ERROR ++: <=9=(0434) [C] 'misc' is not declared.
45: { printf("Number of miscellaneous characters in
file : %i\n",entry);
46:     for(c2=0;c2<=31;c2++);

```

```
++ WARNING ++: <=2=(3109) Empty statement (';' on its own)
- if this is deliberate, it is best to put ';' on a line by
  itself.
```

Not what the programmer had intended. Causes an empty 'for' loop to be executed and the following statement to be executed only once.

```
47:     printf("Character %c counted %i times.\n",c2,
           Miscchar[c2]);
48:     for(c2=128;c2<=255;c2++)
49:         printf("Character %c counted %i times.\n",c2,
           Miscchar[c2]);
           ^
++ WARNING ++: <=2=(2212) Braces should be used in control
statements, even when the body has only one statement.
++ WARNING ++: <=3=(3682) Index may take a value greater
than number of elements.
```

The control loop variable will have a value that causes an access outside the bounds of the array Miscchar.

```
50: }
51: fclose(fp);
52:
53: }

++ WARNING ++: <=2=(2006) 'main()' has more than one
'return' path.
```

The exit(1) statement on line 22 is considered equivalent to a return statement.

```
++ WARNING ++: <=0=(5001) This function has a significant
number of decisions and may be difficult to understand -
'main() : STCYC = 12' .
```

The Cyclomatic Complexity metric of this function is rather high. This is a measure of the number of decisions being made. It might be wise to re-evaluate the logic of the code.

```
++ WARNING ++: <=0=(5002) This function has a high path
count and may be difficult to test - 'main() : STPTH = 500'
```

The computed value of the Static Path Count metric has exceeded the specified threshold. (For this function, the Estimated Static Path Count was measured at 500)

The Corrected Source File

This is the sample program after making appropriate corrections.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

#define SPACESTART 48
#define SPACESTOP 122

#define MISCSTART 128
#define MISCSTOP 31

#define LINEFEED 10

#define MARKER1START 47
#define MARKER1STOP 0
#define MARKER2START 126
#define MARKER2STOP 123

#define MEMALLOC 255

/*
   Specifying "magic numbers" by macro or as 'const' helps
   the maintenance process, and is good practice.
*/

int main(int arg,char *argc[])
/*
   Use 'int' return type, since ISO declaration of main
   expects it.
*/
{
    char *   filename;
    int      classed=0;
    /*
       'classed' has replaced 'class', to avoid C++ keyword
    */
    int      c2=(-1);
    int      misc=0;
    /*
       variable identifier 'entry' has been replaced with 'misc'
       so there is no confusion over K&R C.
    */
    int      Miscchar[MEMALLOC] = {0};
    /*
       Initialise the array with all 0's first, through
       default initialisation.
    */
    int      spaces=0;
```

```
int      lines=0;
FILE *   fp;
/*
  Typing error fixed on 'FILE *fp'.
  Variables are now initialised and aligned properly.
  'unsigned int' is removed as it causes implicit
  conversion.'count' has been removed as it is redundant.
*/
filename = (char *) malloc(MEMALLOC);
/*
  Better to use constants for numeric values, as it
  ensures maintenance issues are easier to spot.
*/

printf("Word Counting Program V1.0 \n");
/*
  Alignment of the code is now consistent
*/
if(arg==1)
{
    printf("\nPlease type in text file name : ");
    gets(filename);
    /*
      Portability of this function is likely to be a problem.
      It behaves in a different manner on different
      platforms.
    */
}
else
{
    strcpy(filename, argc[1]);
}
/*
  Safer to put braces around the control expression
  Could have been overlooked by someone amending the code.
*/

printf("\n");
fp=fopen(filename,"r");
if(fp==0)
{
    printf("Can't open : %s",filename);
    return 1;
}

classed = getc(fp);

while( classed !=EOF)
/*
  Assignment comparison has now been removed
*/
{
    if(((classed<=MARKER1START) && (classed>=MARKER1STOP))
```

```

    || ((classed<=MARKER2START) && (classed>=MARKER2STOP)))
/*
Numbers have been removed and replaced with #defines,
which aids maintenance.
*/

/*
More braces have been added to remove precedence
dependence.
*/
{
    if((c2>=SPACESTART) && (c2<=SPACESTOP))
    {
        spaces++;
        /*
        Safer to enclose in braces even if there is only one
        statement.
        */
    }
    if((classed>=MISCSTART) ||
        (classed<=MISCSTOP) )
    {
        misc++;
        Miscchar[classed]++;
        /*
        Values have already been initialised
        (previously not)
        */
    }
}
if(classed==LINEFEED)
{
    lines++;
    misc--;
}
c2=classed;
classed = getc(fp);
}

printf("File : %s\n", filename);
printf("Length of file : %u bytes\n",
        _filelength(_fileno(fp)));
printf("Number of lines in file : %i\n",lines );
/*
Conversion specifier type fixed.
*/

printf("Number of words in file : %i\n",spaces);

if(misc > 0)
{
    printf("Number of miscellaneous characters in file :
           %i\n",misc);
}

```

```
for (c2=0; c2<=MISCSTOP; c2++)
/*
    Typo error (semi-colon on for statement line) removed.
    Also added braces to enclose the statement.
*/
{
    printf("Character %c counted %i times.\n", c2,
        Miscchar[c2]);
}
for (c2=MISCSTART; c2<MEMALLOC; c2++)
{
    printf("Character %c counted %i times.\n", c2,
        Miscchar[c2]);
}
}
fclose(fp);
free (filename);

return 0;
/*
    main expects a specified integer.
    Return statement now provides this.
*/
}
/*
    Some of the logic has been changed which has reduced
    both the complexity and estimated static path count
*/
```

CHAPTER 6

Reports

QA C produces reports on data that has been collected during source code analysis. These reports provide statistics on specific aspects of your source code.

QA C generates the following reports:

- Project Warning Summary

This report displays, by message level, a summary of all the messages that were detected in your project.

- Warning Listing

This report displays all message occurrences in the selected folder or files of your project.

- Identifier Declarations

This report displays, with the filename and line number, the identifiers that have been declared in your project.

- Close Name Analysis

This report displays a listing of all identifiers that differ by only one character.

- External Name Cross-Reference

This report displays all the external definitions and declarations that were called in your project.

- COCOMO Cost Estimation Model

This report displays the estimated development time and cost to complete your project. This is calculated using QA C's COCOMO metrics.

- Project Metrics

This report displays a number of project-based metrics, which are not suitable for display in any of the metrics browsers, due to the single values that are generated per project.

Notes:

You can also create reports of your own through Reports>Custom Reports>Configure. See Writing Custom Reports of Chapter 10: Advanced Topics.

For the file selection mechanism when opening Reports, see Chapter 2: Making File Selections.

Project Warning Summary

The Project Warning Summary report presents the total number of diagnostics encountered according to the enabled message set, among all the source files and associated header files in the project. The output lists all analysed source files and associated header files, and details the number of detected warnings for each message level from 0 to 9. The report also provides a total for each warning level, and a total for each file, and the overall number of warnings in the entire project.

If you configure your personalities to suppress specific messages or message levels, these messages will not be counted. If you suppress an entire message level, an asterisk will appear in the appropriate column (level) and that level will not be included in the file and project totals.

Execution of this report will result in a permanent file being created in the output location of the root folder. It will have a filename constructed from the root folder appended with `_sum.txt`.

This report is used to locate aspects of the source code that have a large number of errors at a particular warning level. It also provides a snapshot of enabled message levels.

Warning Summary Message Count

The Warning Summary report produces a message count that exactly matches the Message Browser utility. There are two areas of particular emphasis:

- Messages generated from shared header files will be de-duplicated in this report. This will be true even if the navigation through each header, due to different `#define` logic, is different. All that matters is that the `.err` file from each TU has the same message at the same location (in the same shared header file).
- Output from the CMA analysis phase is included in this report. Diagnostics that are tagged to a specific source file will be included in that file's count. "Multi-homed" diagnostics, which generally note inconsistencies between source files, are listed at the bottom of the report in a separate category.

Warning Listing

The Warning Listing report is a summary of all the warning messages in the selected folder or files. It is useful if you want to locate all occurrences of a message. The report only displays warning messages associated with files that have been analysed and does not initiate any analysis.

The number of occurrences of each warning message is listed by file for each message number. The message numbers are ordered according to their message group.

Links are provided to annotated source code and to further explanatory help if the link files exist. If you want to take full advantage of these links, you will need to create the necessary HTML annotated source files. See [Changing Your Annotated Source Settings of Chapter 2: Projects](#).

Warning Listing Message Count

The Warning Listing report is a roll-up of diagnostic counts from each translation unit, and will include links to the annotated source when this exists. Because of the separate execution model involved, there are two areas of particular emphasis:

- There will be no de-duplication of diagnostics from shared header files. The individual annotated source output will contain each

instance of a shared header warning. Thus, the primary output html from this report will count each instance in its counts.

- There is no capability to display CMA output. Neither individual source nor “mulit-homed” diagnostics will be counted in the report. There will thus be a mismatch between this report and the summary at the bottom of each annotated source output.

The Warning Listing report is thus deprecated, and there are alternative and more efficient means of viewing summary and browsing message output.

Identifier Declarations

The Identifier Declarations report details all identifiers that have been declared within the source code.

Listed alphabetically by identifier, the report shows the file name of the identifier, the line number of the file on which the identifier was declared, and the type information of the identifier.

Close Name Analysis

The Close Name Analysis report details identifiers that have similar names. Listed by file, the report displays identifier names that differ by only one character.

For each file, a summary gives the number of pairs of close names, the total number of identifiers, and the closeness metric. The closeness metric is a QA C metric that statistically measures the number of close names in the file.

The Close Name Analysis report can be used to identify those areas of your source code where the closeness of the names of variables may cause difficulty in reading the source code.

External Name Cross-Reference

The External Name Cross-Reference report details the external definitions and declarations called by each source file.

Listed by source file, the reports show the external definitions and declarations and the line number on which the definition or declaration is located.

This report is used as a reference that details all external declarations and definitions. This listing helps you to track all external declarations and definitions for the source files.

COCOMO Cost Model

The QA C COCOMO report displays software development cost and time estimates based on the Constructive Cost Model (COCOMO) by Boehm¹. These estimates are calculated during analysis as QA C's six COCOMO metrics.

The calculation of these estimates is based on measuring the size of the delivered code factored by values for the type of system used to develop the code. Boehm has classified systems into the following three classifications:

- Organic, an environment where developers have few external interface or hardware constraints. Organic systems tend to use databases and focus on transactions and data retrieval (e.g. a banking or accounting system).
- Embedded, an environment where the software can be described as highly dependent on its environment. These systems tend to contain real-time software that is an integral part of a larger hardware-based system (e.g. a missile guidance system).
- Semi-detached, an environment where the development is of a larger product that has complex interfaces between components (e.g. applications such as database or Windows applications).

¹ Boehm, B, (1981) *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey.

The COCOMO report displays, for each system classification, totals for the programmer time, the development time, the number of programmers needed to complete on time, and the estimated cost to finish the project. These totals are calculated for three different levels of tool usage.

Programmer Time

The COCOMO report displays the software development cost as Total Programmer Time. This value is based on the following Boehm equation for programming effort:

$$E = a * S^b$$

where E is programmer time or effort, measured in person months, and S is the size of delivered code, measured in thousands of delivered source instructions (KDSI). The size of delivered code is calculated as the metric STTPP.

The values of a and b are dependent on the following system classifications:

Programming Mode	a	b	Metric
Organic	2.4	1.05	STBMO
Semi-detached	3.0	1.12	STBMS
Embedded	3.6	1.20	STBME

For example, the Total Programmer Time for an organic system is calculated (based on the above formula for programming effort) as the STBMO metric as follows:

$$STBMO = 2.4 * (STTPP / 1000)^{1.05}$$

Development Time

The COCOMO development time estimate is displayed as Elapsed Time for Development and is based on the following Boehm equation for project duration:

$$D = a * E^b$$

where D is the estimate of optimal time to complete the project for the programming effort, E , that was calculated above.

The value of a and b are dependent on the following system classifications:

Programming Mode	a	b	Metric
Organic	2.5	0.38	STTDO
Semi-detached	2.5	0.35	STTDS
Embedded	2.5	0.32	STTDE

For example, the Elapsed Time for Development for an organic system is calculated (based on the above formula for project duration) as the STTDO metric as follows:

$$STTDO = 2.5 * STBMO^{0.38}$$

Tool Usage

QA C calculates the above estimates for three Capability Maturity Model (CMM) levels of software development by applying the factor, F , to the formula for programming effort (Total Programmer Time). For example:

$$E = F (a * S^b)$$

The value of F depends on the following levels of tool usage:

Tool Usage	F
Few Tools (CMM Level 1)	1.24
Some Tools (CMM Level 2)	1.00
Efficient Tool Use	0.83

Note that the factor F is not applied to the calculation of the project duration (Elapsed Time for Development) since the factored value of programming effort is used to calculate project duration.

The report uses the above tool usage modes to calculate the possible savings to be gained by having efficient tool usage. For all cost estimates in the COCOMO Cost Model, the currency is U.S. Dollars

or Equivalent Currency Unit (ECU).

Project Metrics

The Project Metrics report displays a number of metrics that are calculated with single values over a complete project. The calculations are made from CMA analysis, and are stored in the project met file. See Project-Wide Metrics of Appendix D for further details on metrics displayed.

Execution of this report will result in a permanent file being created in the output location of the root folder, and then displayed through the default web browser. It will have a filename constructed from the root folder appended with `_met.html`.

CHAPTER 7

Code Structure

In addition to reports, QA C also provides a visualisation of the interfaces and internal structure of source code. QA C displays code structure in two ways:

- as a relationship graph of aspects of source code, including file inclusion, function calling, and external references.
- as a diagram of the control flow within a function.

Both views are displayed for the current folder or file selection. Selections can be made from files within a folder, or else for a folder plus its sub-folders.

To generate these diagrams choose Relationships or Function Structure from the Browse menu or click the respective toolbar button.

You can toggle between each of these views and also between Metrics Browser and Demographics Browser using the Window menu item.

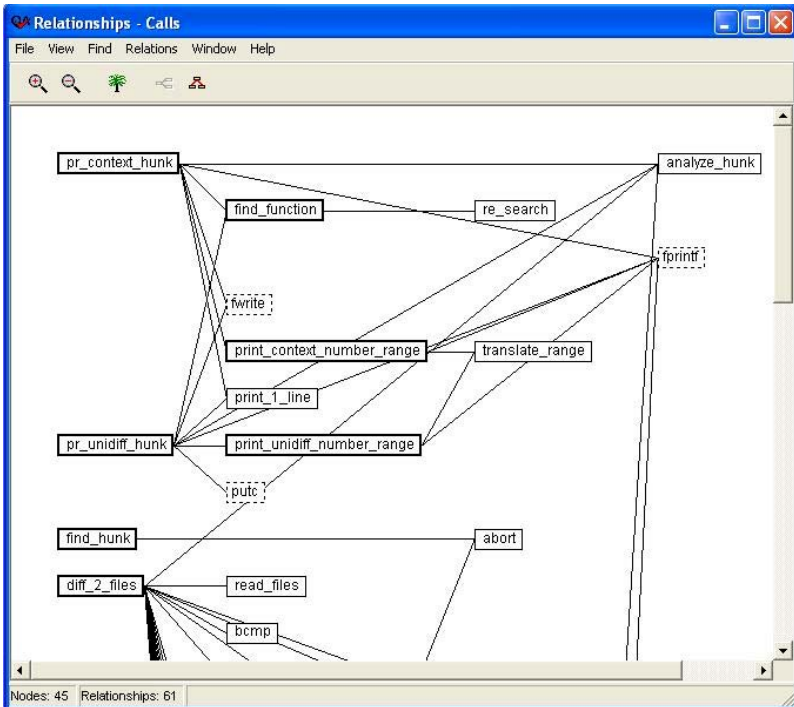
Relationships

The relationship graph displays the following three types of relationships:

- Include relationships that show the files that are included in each of the source files.
- Call relationships that show all function calls, both internal and external (but excluding functions called through pointers), within your source files.
- Refers-to relationships that show all external identifiers, including variables and functions, to which your source files refer.

To generate the relationship graph, select one or more files, a folder, or your project and choose Relationships from the Browse menu or click the toolbar button.

These relationships are displayed as left to right or top to bottom linkages. Each relationship node on the graph details the relationship's data type and parameter types.



Relationship nodes are drawn in three ways:

- with thick lines if the node is a definition,
- with thin lines if the node is a declaration only,
- with dotted lines if the declaration is in a suppressed file. See Suppressing Analysis Output from Header Files of Chapter 4: Analysing Source Code.

Note:

The relationship nodes for includes relationships are all drawn with thin lines.

Choosing Relationships

Choose a relationship type from the Relations menu.

Searching for Relationships

Choose Find Relationship Node from the Find menu. Select a relationship from the list and click Return To Node. QA C returns to the graph, highlighting the relationship.

Displaying Parent and Child Nodes

Select a node and choose Simple Fan In/Out or Full Fan In/Out from the View menu. Simple Fan In/Out displays only the immediate parent and child nodes. Full Fan In/Out displays all parent and child nodes. To draw the entire tree again, choose Complete Tree from the View menu or click the Complete Tree toolbar button.

Starting the Graph from an Individual Node

Select a node and choose Start Tree Here from the View menu.

Selecting Additional Files

Choose Select Files from the File menu. Select or de-select files to be included in the relationships tree. The graph will be redrawn accordingly. The project .met file is always included in the selection, if available.

Printing the Relationship Graph

Choose Print from the File menu. The image will be printed on your default printer. Before printing the image, enlarge the image by several magnifications. When the image is printed, the pages are printed vertically. If the image spans the page horizontally, those pages will be printed after the first vertical column.

Saving the Relationship Graph

Choose Save Image from the File menu. The image can be saved as an either enhanced metafile (.emf), a windows metafile (.wmf), or a bitmap (.bmp).

Viewing Source Code

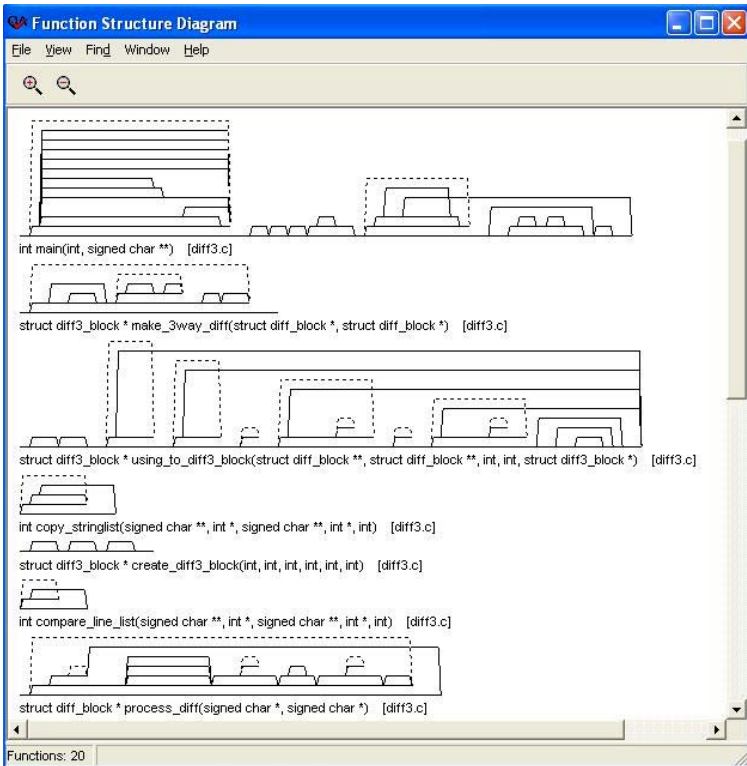
Select a node and choose Source Code from the View menu. You can also highlight two related nodes and choose Relationship Source Code from the View menu. This displays the source code at the point of definition of the relationship.

Drawing the Graph Vertically

The graph is drawn horizontally by default, but you can switch it to a vertical display by menu option View>Vertical.

Function Structure

The function structure diagram displays a visualisation of the control flow of a function. See Appendix C: The Components of Function Structure for information on function structure.



For each function, the diagram describes its:

- structure,
- name,
- data type,
- parameter types,
- source file name.

Searching for Functions

Choose Find Function from the Find menu. Select the function and click Return to Function. QA C returns to the structure diagram at the function you selected.

Display Simple Functions

Choose Simple Functions from the View menu. Simple functions are functions that have a straight path through the function. These functions have no decision complexity and have a Cyclomatic Complexity of 1.

Selecting Additional Files

Choose Select Files from the File menu. Select or de-select files to be included in the structure diagram. The diagram will be redrawn accordingly. The project `.met` file is always included in the selection, if available.

Printing the Structure Diagram

Choose Print from the File menu. The image will be printed on your default printer. Before printing, it may be necessary to enlarge the image by several magnifications. If the image is too large, it will be printed on multiple pages.

Saving the Structure Diagram

Choose Save Image from the File menu. The image can be saved in enhanced metafile (`.emf`), windows metafile (`.wmf`), or bitmap (`.bmp`) format.

Viewing Function Structure Source Code

You can select a point within the structure diagram and display the source code at that point within the function. Clicking within a structure diagram will amend the display to make one of the join points in bold (selected). Clicking to the left will make the top of the function selected. Then choose Source Code from either the View menu or the pop-up menu. This displays the source code at that point of that function.

CHAPTER 8

Metrics

A metric is a measure of some quantifiable attribute of source code. No single metric will provide a comprehensive indicator of code quality. Each provides a different view of aspects such as complexity, readability, residual bugs, or development effort.

QA C calculates 69 different metrics, of three types:

- function-based metrics,
- file-based metrics,
- project-based metrics.

Metric values are calculated automatically during the analysis process and are recorded in the `.met` file. See Appendix D: The Calculation of Metrics for a detailed description of how they are derived.

You can inspect metric values by selecting one or more files, a folder, or your project and choosing one of the following menu options:

- Browse>Demographics
- Browse>Function Metrics
- Browse>File Metrics

The Browse menu options are also available as buttons on the toolbar.

Project metrics are not usefully presented through any of these visualisations, and it is only possible to display project metrics as a set of single measurements, see Project Metrics Report.

Metric Thresholds in Analysis

Most metrics can be directly linked to warning messages. By applying a metric threshold configuration entry, any resultant metric value falling outside these limits will then generate an analysis warning.

Thresholds can be set as part of configuring a coding standard.

These thresholds can be set in Analyser Personality>Metrics>Metrics Thresholds as a relational expression in the format:

```
metric op value[:message]
```

where:

<i>metric</i>	The metric name as it appears in the metrics records of the .met file.
<i>op</i>	A relational operator (<, <=, ==, >, >=).
<i>value</i>	The integer value for which QA C will generate a warning message if the threshold is broken.
<i>message</i>	The warning message that is generated. This message number should refer to a warning message that you have added through your user message file. See The User Message File in Chapter 10: Advanced Topics.

If you do not specify a message number for a threshold, QA C will issue warning 4700. For example:

```
STCYC>30
```

will generate the following warning message if the cyclomatic complexity is 35:

```
++ WARNING ++: <=0=(4700) Metric value out of threshold  
range: function() : STCYC = 35.
```

QA C cannot enforce thresholds for metrics expressed as decimal fractions. This applies to the following metrics: STDEV, STDIF, STKDN, STMOB, and the COCOMO metrics (STBME, STBMO, STBMS, STTDE, STTDO, STTDS). QA C also does not directly enforce the setting of thresholds of project metrics, and any such desired thresholds would need to be manually created in a phase of Cross-Module Analysis.

A threshold can however be specified for the comment density metric, STCDN. Since this metric is calculated as a percentage and expressed as a value from 0 to 100, you must enter the threshold accordingly. For example:

```
STCDN<60:5001  
STCDN>80:5002
```

These thresholds will enforce a lower comment density limit of 60% and a higher limit of 80%.

Threshold Order

Once a threshold is entered, it appears in the list of thresholds. During analysis, QA C will compare metric values with the ordered list of metric thresholds. If the relational expression is true, it will issue a warning and then cancel further lookup in the threshold list for that value.

As a result, the order in which thresholds appear in the list is important. If you want QA C to issue different warnings for increasing degrees of severity, you should ensure that your thresholds appear in decreasing order of severity in the list. For example:

```
STCYC>=30:5003  
STCYC>=20:5004  
STCYC>=10:5005
```

These thresholds, in this order, could be used to enforce three levels of Cyclomatic Complexity (STCYC). If the metric value is 35, warning 5003 is generated. If the metric value is 25, warning message 5004 will be generated. If the metric value is 15, warning message 5005 will be generated.

If you are entering more than one set of upper and lower thresholds, you will also need to arrange these limits in descending degrees of severity. For example:

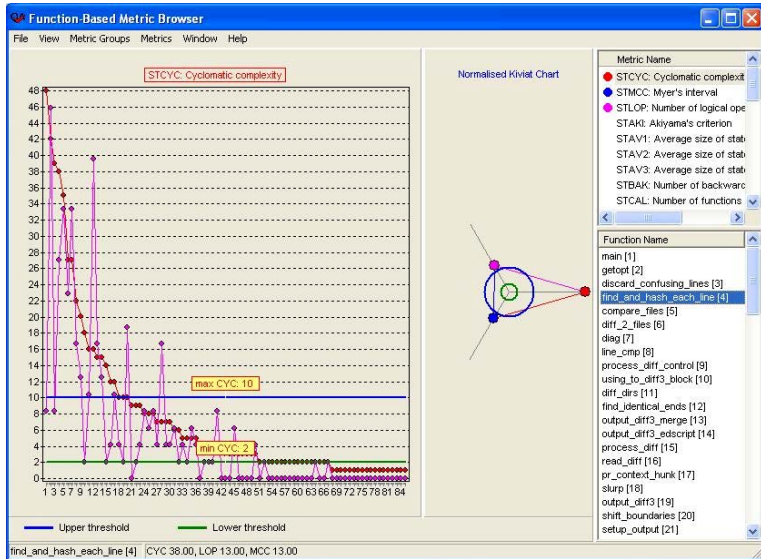
```
STCDN>95:5006  
STCDN<40:5007  
STCDN>90:5008  
STCDN<60:5009
```

For a given value of STCDN, these thresholds will issue warning message (in order):

```
5006 when above 95%  
5007 when below 40%  
5008 when between 90% and 95%  
5009 when between 40% and 60%
```

The Metrics Browser

The Metrics Browser is used to display function and file metrics.



Metric Name Listing Panel

The metric name listing panel contains a list of the available function-based or file-based metrics as appropriate. From this listing, you can select up to ten metrics to be plotted on the graph by double clicking. Alternatively they can be selected from the Metrics menu.

The colour of the metric line, as plotted on the graph, is displayed beside the metric name in the metrics listing. As you select a metric to be plotted, it is moved to the top of the listing with the other plotted metrics. To de-select the metric from the metrics listing, double click the metric name or de-select it on the Metrics menu.

Name Listing Panel

The name listing panel displays a list of all the functions or files

currently displayed in the browser. An item is only displayed when selected. You can change the selection list by choosing **Select Files** from the **File** menu of the browser. See **Chapter 2: Making File Selections** for details on opening the Metrics Browser with a filter selection.

The position of the name in the listing is reflected in its position on the x-axis of the metrics graph and is identified by an index number. You can change the order in which the list is sorted by selecting **Sort by Name** or **Sort by Metric** from the **View Menu**. If you sort by name, the list will be in alphabetical order. If you sort by metric, the list will be sorted according to the magnitude of the metric that is currently in focus, i.e. the metric selected in the metric name panel.

If you sort by **Metric**, you can also choose to sort in either ascending or descending order by right clicking on the graph or on the selected metric in the **Metric Name Listing** panel.

If you click an entry in the listing, it will be highlighted and its values for all selected metrics will be shown on the display line at the bottom of the metrics graph.

The Metrics Graph

The metrics graph plots the metrics that you have chosen to display for each function or file. A different coloured line corresponding to the colour in the metric name listing identifies each metric.

If you want to know the metric values for a specific item, move the mouse pointer over an appropriate node on the graph. The values are displayed on a line just below the x-axis.

The metrics graph will display up to ten different metrics but at any one time, there will be one metric that is currently “in focus”. You can switch the focus to a particular metric using the **Metric Name Listing** panel or by moving the mouse pointer over an appropriate node as described above. The vertical axis is always shown with a scale appropriate to the focussed metric.

Also displayed on the graph are two horizontal lines that reflect lower and upper limits for the focussed metric. The values of these limits are

loaded from the text file, `qacmet.txt`. You can find this in the QA C `bin` directory. If you wish to configure the limits, you will need to edit the contents of this file using a text editor. There is no mechanism to change these limits from within the GUI.

Note:

The metric limits displayed in the browser are quite distinct from any thresholds used to generate warning messages. The browser limits are derived from the `qacmet.txt` file and are not part of a project or personality configuration.

Selecting Additional Files

Choose Select Files from the File menu. Select or de-select files to be included in the metrics browser. The project `.met` file is always included in the selection, if available.

Reloading Metrics Data

If you have re-analysed your source files, choose Reload Data from the File menu to load the new analysis data.

Filtering Metrics

Choose Filter Metrics... from the View menu to display a filter dialog window. Filters can be applied in consecutive (logical AND) or additive (logical OR) modes. Lower and upper value exclusion filters can be applied for any combination of selected metrics.

On the right hand side of this dialog are two buttons. To remove all filters, select Clear All (or alternatively remove single entries from the chart). To apply the upper and lower limits from the `qacmet.txt` file select Apply Min/Max. Upper and lower limit values of 0 or 1 are excluded from this operation, so that some metric upper and lower limits will not be applied automatically.

Click OK to apply filters and redraw the Metric Browser window. Filters for selected metrics will continue to apply when adding and removing metrics, although de-selecting and re-selecting a metric will

cause its filter to be removed. All remaining active filters will be displayed on successive display of the filter dialog.

Changing the Metric Group

Choose a metric group from the Message Groups menu. The metric group information will be loaded into the Metrics Browser for your selected files.

Magnifying the graph

Select an area by clicking and dragging downwards and to the right to create a box. Drag left and upwards to display the full graph image again.

Exporting Metrics

Metrics values can be exported by choosing Export Data from the File menu. Metrics values will be written for all files that are displayed in the Metrics Browser. This data is saved as a `.csv` file suitable for importing to a spreadsheet.

Creating a Demograph file

Select a metric and choose Create Demograph file from the File menu. A demograph file contains the metric values grouped into ten percentile groupings that represent ten degrees of code quality and will appear in the list of available demographics in the Demographics Browser.

Note:

When you create a demograph file, the values of the selected metric will be saved in a `.prm` file in the `demographics` directory. The Demographics Browser will only display demograph files that are located in this directory.

Printing the graph

Choose Print from the File menu. The main graph (not including the

kiviat segment) will be printed on your default printer.

Saving the metrics graph

Choose Save Image from the File menu. The graph can be saved in enhanced metafile (.emf), windows metafile (.wmf), or bitmap (.bmp) format.

Displaying a Kiviat Diagram

A Kiviat Digram can be shown as an additional pane of the Metrics Browser by choosing Kiviat Diagram from the View menu. Each metric that is displayed in the metrics graph will also be drawn in the Kiviat Diagram for a selected file or function. See The Kiviat Diagram later in this chapter for a detailed explanation.

Displaying Source Code

Select an item from the name listing and choose Source Code from either the View menu or the pop-up menu, to view the definition of the function or the file.

Dismplying Function Structure

Select a function from the function listing and choose Structure Diagram from the View menu. You can only display the structure diagram for functions.

The Demographics Browser

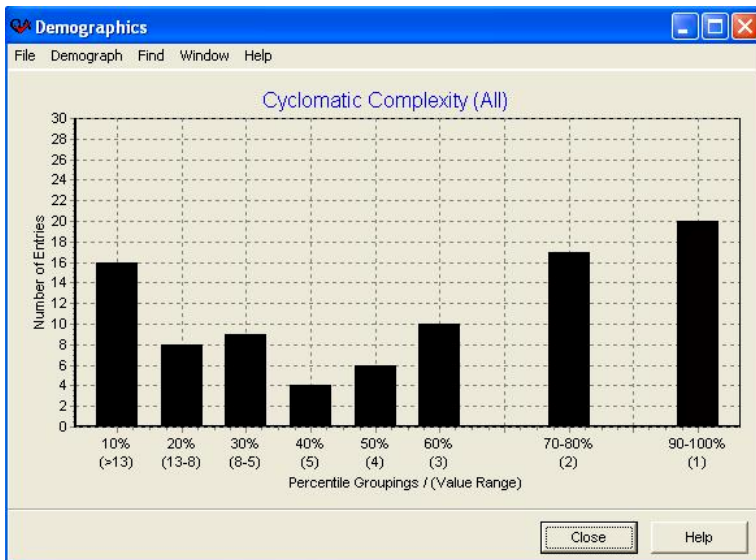
The Demographics Browser is activated by selecting Demographics on the Browse menu or by clicking the Demographics toolbar button. Three function metrics are supplied with various demograph data to display in the Demographics Browser:

- Depth of nesting,
- Cyclomatic Complexity,
- Static Path Count.

A Demograph displays a bar chart of metric values plotted against some industry averages. Programming Research has analysed a large body of industry code and has grouped the results into ten percentile groupings for each metric.

The height of each bar in the graph represents the number of functions that fall within a percentile grouping. These groupings represent ten degrees of quality from least to best. Metric values from your project are plotted against these standards to measure the quality of your source code. The value range for each percentile is displayed in brackets below the percentile grouping.

High quality source code will have a high proportion of its functions falling in the higher percentiles. An even mix among the percentile groupings suggests that the project has an average level of demographic quality.



For some metrics we have supplied up to four industry types that address different development modes:

- Min,
- Public Tools,
- X11 Apps,
- All - a total of the three types.

You can choose one of the available combinations of metric and industry type from the Demograph menu. Cyclomatic Complexity (All) is displayed by default. If you have created a demographic file from within the Metrics Browser, it will appear in the Demograph menu.

You can also create demograph files for any metric based on your own population of metric data from within the Metrics Browser, and then display them in the Demographics Browser. See Creating a demograph file in The Metrics Browser in this chapter.

The Demographics Listing

The height of each bar within the demograph represents the number of functions whose metric value falls within the relevant quality percentile. You can see which functions contribute to a particular bar by right clicking on the bar to bring up a Metrics Listing window.

If you select the Find menu item, you will be presented with a list of all selected functions.

The Demographics Listing window has a search facility to enable you to find a particular function name. It also contains options to enable you to generate a Kiviat diagram for the selected function, or to view the source code.

Selecting Additional Files

Choose Select Files from the File menu. Select or de-select files to be included in the demographics browser. The project `.met` file is always included in the selection, if available.

Reloading Metrics Data

If you have re-analysed your source files, choose Reload Data from the File menu to load the new analysis data.

Saving the Demographics Graph

Choose Save Image from the File menu. The graph can be saved as an enhanced metafile (.emf), a windows metafile (.wmf), or a bitmap (.bmp).

Printing the graph

Choose Print from the File menu. The graph will be printed on your default printer.

Viewing Source Code

Select an entry from the Demographics Listing and choose Source Code.

Displaying Function Structure

Select a function from the Demographics Listing and click Structure. This option is only available from function-based metrics.

Displaying a Kiviat Diagram

Select an entry from the Demographics Listing and select Kiviat Diagram. The diagram will be drawn for all available metrics of the given type.

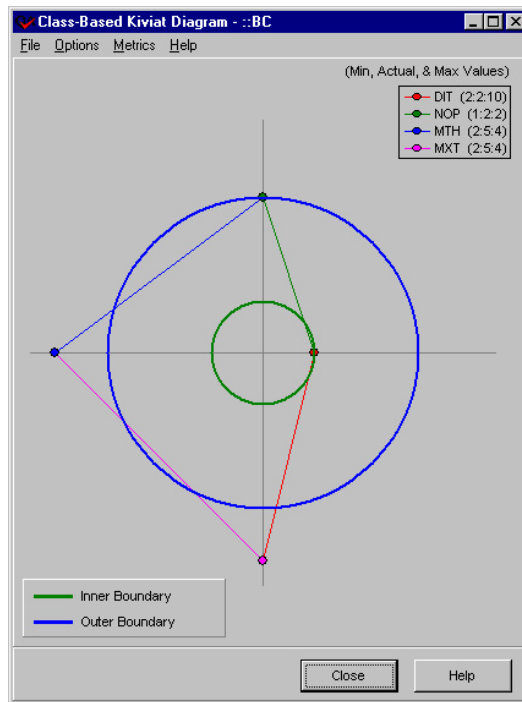
Exporting Metrics

Metrics values can be exported from within the Metrics Browser. Choose Export Data from the File menu. Metrics values will be written for all files that have been selected. This data is saved as a .csv file

suitable for importing to a spreadsheet.

The Kiviati Diagram

From the demographics browser, you can generate a standalone Kiviati diagram. The Kiviati diagram allows for a single normalised plot of a range of metrics for a single file or function.



Two concentric circles represent the high and low limits for the displayed metrics. The values of these limits are loaded from the text file, `qacmet.txt`.

Each metric is represented by one of the coloured nodes and the length of the radial line represents its value from the node to the centre of the circles. The metric values are normalised so that they can be plotted on a scale related to the size of the circles. Values are logarithmic above the outer circle, for further display ease. A node that lies between the

inner and outer circles will represent a metric value that lies within acceptable limits.

The values of the displayed metrics and their respective lower and upper limit values are displayed in the legend box at the top.

By comparing the metrics to their high and low boundaries, you can see the parts of your source code that do not conform to your programming standards or represent 'outlying' values. For example, a function with moderate cyclomatic complexity and very high path count suggests a lot of sequential logic that could be split into sub-functions.

The following features are available only on the standalone Kiviat window.

Changing the display

From the Options menu several display items can be toggled on/off:

- Show Labels for values positioned beside each metric node
- Show Legend for legend of min/actual/max
- Show Circles for concentric blue and green circles representing normalised minimum and maximum values for each metric.

Selecting Metrics

Choose individual metrics from the Metrics menu to toggle on/off their inclusion in the chart. Select All and Select None are also included in the menu item.

Printing the graph

Choose Print from the File menu. The kiviatic graph will be printed on your default printer.

Saving the metrics graph

Choose Save Image from the File menu. The graph can be saved as an enhanced metafile (.emf), a windows metafile (.wmf), or a bitmap (.bmp).

CHAPTER 9

Running QA C on the Command Line

Running QA C on the command line offers the same analysis capabilities as through the GUI although it does not offer code visualisation features such as reports and browsers. The principal programs are:

- `qac.exe` which analyses your files,
- `viewer.exe` which launches the Message Browser,

In addition, there are some utility programs:

- `errsum.exe` which generates a summary of diagnostics across a project or set of source files,
- `errdsp.exe` which generates annotated source code,
- `prjdsp.exe` which generates the Warning Listing report.

These executables, which are all launched from within the GUI, can be run from a command line batch or make file. It is often convenient to configure your own development environment so that source code can be analysed directly without opening the GUI.

QA C's command line environment relies on setting a number of environment variables and configuration options.

Environment Variables

QA C references the following environment variables:

`QACBIN`

The location of QA C's system files such as the message file, `qac.msg`, and the configuration file, `qac.cfg`. This should be set to the `bin` directory.

`QACHELPPFILES`

The location of the HTML help pages, explaining the rationale behind each generated

message. This should be set to the `help` directory, and help files located in `\messages` sub-directory.

`QACOUTPATH`

The location in which output files are generated. This is usually part of project configuration, and, in the GUI, is set in the folder parameters.

You should review the batch files `QACCONF.BAT` and `QACRUN.BAT` for an example of how to configure these environment variables.

Configuration Options

Configuration options pass information to QA C's executables and can be supplied directly on the command line. For example the command:

```
qac -i c:\QAC\include\ansi mysource.c
```

will instruct QA C to analyse the file `mysource.c` and to search for include files in the specified path.

Option Syntax

Some configuration options apply to `qac.exe`, and some to Message Browser and other viewing utilities (`errdsp.exe` and `prjdsp.exe`). Options are processed sequentially in the order in which they are encountered and a supplied option may override a previous setting. If you do not supply an option, the default value is used.

Options are preceded by a dash and take arguments following the option name. Names are not case sensitive and most options can be abbreviated. For example:

```
-define ident=1  
-DEFINE ident=1  
-d ident=1
```

will all set the macro `ident=1` equivalent to `#define ident 1`.

Appendix B: Configuration Options provides a detailed list of options and their abbreviations.

Toggled Options

Options can be turned on or off by using the toggle characters, + and -. The toggle character should follow the option with no intervening space. For example:

```
-html+
```

will generate annotated source code with HTML links.

Options with Arguments

Options and arguments are usually separated by a space. For example:

```
-threshold STCYC>2:5001
```

Most options will override previous settings, but some add to previous settings. For example:

```
-only 2000,2001,2002,2003,2004,2005  
-only 4055,4056,4057,4058,4059,4060
```

If an option (e.g. `-i` or `-q`) has an argument that contains a path name with embedded spaces, the path must be enclosed in quotation marks.

The `-via` option

It is usually convenient to supply configuration options grouped in a file. This is usually a text file that contains a number of configuration options. These option group files are specified on the command line by using the `-via` option. For example the command:

```
qac -via c:\QAC\personalities\start.p_c mysource.c
```

will instruct QA C to read the configuration options contained in the `start.p_c` compiler personality.

Personality Files

Personality files are option files that reflect different aspects of configuration (i.e. messages, compiler and analysis options). The classification of options in this way is arbitrary, although the message personality contains options that apply principally to `errdsp.exe` and the compiler and analyser personalities contain options mainly for `qac.exe`, all options are read by both programs. Options that are not

relevant are ignored.

The programs will therefore operate no differently if all options are supplied in one file and, in fact, this is precisely what happens. When a source file is analysed from the GUI, options from the three personality files are concatenated and saved in a temporary file in \QAC\temp. Analysis is then run using the command:

```
qac -via C:\QAC\temp\settings.via mysource.c
```

Note:

Although this temporary file is labelled `settings.via`, it is actually formed as a unique filename. At the end of the process, it is renamed back to `settings.via` as a convenience.

The `-via` option can be nested. You might achieve the same result on the command line by creating a file that contains three nested `-via` settings. For example, suppose the file `myconfig` contains the following:

```
-via C:\myproj\person\analyser.p_a  
-via C:\myproj\person\compiler.p_c  
-via C:\myproj\person\message.p_s
```

Analysis would then be run with the command:

```
qac -via C:\myproj\person\myconfig mysource.c
```

This is the recommended approach if you want to analyse on the command line using the personalities that you created within the GUI.

-forgetall

If you are introducing option files with `-via` on the command line, you may wish to ignore options previously set by using the `-forgetall` option.

This option is especially useful with options that add to existing entries. While most configuration options override previous settings, some, such as `-i`, will add entries to the initial setting. For example, the command:

```
qac -forgetall i -i C:\QAC\include\ansi mysource.c
```

will instruct QA C to ignore all previously set include paths and only search for include files in `C:\QAC\include\ansi`.

Analysing Source

Configuring Primary Analysis

Files are analysed on the command line by entering:

```
qac [option] source.c
```

where:

option any combination of configuration options.

source.c the source file to be analysed

Output files will be generated in the directory specified by `QACOUTPATH` unless you specify another directory with the `-op` option.

Common Analyser Options

<code>-i path</code>	Specifies the paths of header files.
<code>-q path</code>	Suppresses messages from header files in path.
<code>-d ident[=[value]]</code>	Defines macros.
<code>-op path</code>	Specifies the path to which output files are generated.
<code>-tabstop columns</code>	Specifies the number of spaces represented by a tab character.

Analyser Return Codes

When `qac.exe` has been run successfully, the program will exit with a return code of 0. A non-zero exit code will be generated in response to severe problems such as:

- Hard errors were encountered during analysis or, if the `-maxerr` option is set to a non-zero value, the number of hard errors has been exceeded. A hard error is triggered when QA C is not configured correctly or if it encounters code that it is unable to

parse e.g. when there are illegal characters in the source code.

- A `#error` preprocessor directive has been processed.
- A licensing problem has occurred.
- An internal failure has occurred within `qac.exe`.

A list of analyser return codes is included in Appendix E: Analyser Return Codes.

Running Secondary Analysis Checks

The principles of how to write a secondary analysis program are discussed in Chapter 10, Advanced Topics. A secondary analysis program will be run after an analysis by `qac.exe` and will typically use the relevant `.met` file to perform further customised checks.

With secondary analysis programs supplied by Programming Research, any resultant diagnostics will be automatically written to the relevant `.err` file. For external user-created secondary analysis, the resulting diagnostics can be written to the `.err` file using `errwrt.exe`.

As a typical example, the command line to run a set of naming convention rules through secondary analysis is:

```
pal.exe QAC [option] -nrf naming.nrf source.c
```

where:

<i>QAC</i>	the product label for internal usage
<i>option</i>	any combination of configuration options
<i>naming.nrf</i>	the set of naming rule configuration entries (See Appendix I, Naming Convention Checking for more details)
<i>source.c</i>	the source file to be analysed

Note:

Secondary analysis configuration through the GUI is handled in a

special way, and will not be available when running analysis on command line. In the GUI, the secondary analysis program is picked up from special `-rem` settings in the message personality and run automatically. From the command line, it must be run explicitly, even if a message personality containing secondary analysis settings is passed to `qac.exe` using a `-via` option. Normally, command line users will control and launch the program using a batch script.

Running CMA Checks

A CMA analysis program will be run after all file-based analysis is complete, and will typically use the set of all `.met` files to perform further customised checks.

As a typical example, the command line to run the default supplied CMA program is:

```
pal.exe QAC [option] -list filelist.lst
```

where:

<i>QAC</i>	the product label for internal usage
<i>option</i>	any combination of configuration options
<i>filelist.lst</i>	the full set of project source files (see Appendix B: Configuration Options for formatting details)

Viewing Diagnostic Output

Launching the Message Browser

The Message Browser is launched in a separate window on the command line by running `viewer.exe`. It gathers, sorts, and displays all message numbers that are contained in the `.err` files for every source file in its input list. `viewer.exe` will look for the `.err` files in the path set by `QACOUTPATH` or the `-outputpath` option.

To launch the Message Browser for one or more files enter:

```
viewer.exe QAC [option] source1.c source2.c source3.c
```

To launch the Message Browser with a list of files, enter:

```
viewer QAC [option] -list filelist.lst
```

where:

<i>QAC</i>	the product label for internal usage
<i>option</i>	any combination of configuration options
<i>source.c</i>	the source file(s) to load and display
<i>filelist.lst</i>	the full set of project source files (see Appendix B: Configuration Options for formatting details)

Note:

Although this temporary file is labelled *filelist.lst*, it is actually formed as a unique filename. At the end of the process, it is renamed back to *filelist.lst* as a convenience. See Appendix B: Configuration Options.

Summary Diagnostic Output

The analysis summary report in the GUI, Reports | Project Warning Summary, can be run on command line, using *errsum.exe*.

The command line to run this program is:

```
errsum.exe QAC [option] -list filelist.lst  
-file output.txt
```

where:

<i>QAC</i>	the product label for internal usage
<i>option</i>	any combination of configuration options
<i>filelist.lst</i>	the full set of project source files (see Appendix B: Configuration Options for formatting details)

summary.txt represents the output file to which a summary of diagnostics will be generated

The errsum program output is formatted in plain text, arranged to present columnar information representing the de-duplicated set of diagnostics captured from all phases of project analysis (primary, secondary, and CMA), with the horizontal axis organised into source and header files.

Annotated Source Generation

On the command line, annotated source code is generated to standard output by entering:

```
errdsp.exe QAC [option] source.c
```

where:

<i>QAC</i>	the product label for internal usage
<i>option</i>	any combination of configuration options.
<i>source.c</i>	the source file for which annotated source code will be generated.

When you run `errdsp.exe`, it will look for the `.err` file that corresponds to the specified source file e.g. `source.c.err`. It expects to locate this file in the directory specified by `QACOUTPATH` or the `-op` option. After `errdsp.exe` locates the `.err` file, it will use the message numbers contained in the `.err` file to merge message text from `qac.msg` with the contents of your source file. The annotated source code can be piped to a file or you can use the `-file` option.

Configuration options are passed to `errdsp.exe` in the same way as they are passed to `qac.exe`. Options that affect `errdsp.exe` will usually be contained in the same configuration files used by `qac.exe`. Options that are not relevant to `errdsp.exe` are ignored.

Common errdsp.exe Configuration Options

<code>-file filename</code>	Outputs annotated source code to <i>filename</i> . If the full path of the file is not given, this file will be created in the path specified by the QACOUTPATH environment variable or <code>-op</code> . If you are generating HTML annotated source code that will be used in the Warning Listing report, this file must be called <i>filename.c.html</i> .
<code>-format string</code>	Determines the format of warning messages. See The Warning Message Format of Chapter 10: Advanced Topics.
<code>-hdrsuppress path</code>	Suppresses messages from all header files in <i>path</i> .
<code>-html+</code>	Generates source code annotated with HTML links. HTML annotated source code is needed for the Warning Listing report.
<code>-references+</code>	Includes references with warning messages.
<code>-summary+</code>	Instead of annotated source code, generates a summary of the number of occurrences of each warning at each message level.

A full list of configuration options is included in Appendix B: Configuration Options.

Warning Listing Report

The Warning Listing report can be run on command line, using `prjdsp.exe`. It gathers, sorts, and displays all message numbers that are contained in the `.err` files for every source file in its input list. `prjdsp.exe` will look for the `.err` files in the path set by QACOUTPATH or the `-outputpath` option. The report can be redirected to a file or you can

use the `-file` option.

Note:

The Warning Listing report does not include CMA output, nor does it take account of comment-based diagnostic suppressions. It is thus deprecated in favour of Message Browser based summaries of diagnostics across a complete project.

In the GUI, the Warning Listing report is generated in HTML format. On the command line the report is generated in plain text format by default. You can generate the report in HTML format by setting the `-html+` option. To use the full functionality of the HTML report, you should make sure that the `QACHELPFILES` environment variable is set to the `help` directory so that explanatory links are generated for each warning message.

To generate the Warning Listing report for one or more files, enter:

```
prjdsp QAC [option] source1.c source2.c source3.c
```

To generate the Warning Listing report for a list of files, enter:

```
prjdsp QAC [option] -list <filelist.lst>
```

where:

<i>QAC</i>	the product label for internal usage
<i>option</i>	any combination of configuration options. The Warning Listing report will accept most of the configuration options used by <code>errdsp.exe</code> .
<i>source.c</i>	the source file(s) for which the report will be generated.
<i>filelist.lst</i>	the full set of project source files (see Appendix B: Configuration Options for formatting details)

CHAPTER 10

Advanced Topics

This chapter discusses some topics that are helpful in the efficient administration of QA C. Most users will want to integrate QA C into an existing development environment. This will usually involve developing some guidelines for the location of directories and customising the tool to enforce a coding standard.

In order to enforce a programming standard, you will need to develop a message personality to include a suitable subset of the standard messages. You can also redesign the message system, add secondary and CMA analysis checks, create custom reports, or configure a layout standard.

Customising the Message System

QA C's message system consists of two files that control the grouping and content of messages that are displayed in annotated source code. If you are using QA C to enforce a coding standard, you may want to reorganise the standard library of messages by regrouping or rewording them.

The two files used by the message system are:

- the message file, `qac.msg`, which contains the library of QA C's messages and defines the message levels and groups,
- an optional user message file, `qac.usr.xxx`, which can define new or reworded messages, message levels, and message groups.

The Message File (`qac.msg`)

The message file, `qac.msg`, contains:

- message levels,
- message groups,

- warning messages.

Message Level Records

Each warning message is assigned to a message group and each group is assigned to a message level in the range 0 to 9. In the standard message file, there are 7 `#levelname` statements that are defined as follows:

```
#levelname 0 Information
#levelname 2 Minor
#levelname 3 Major
#levelname 6 Portability
#levelname 7 Undefined Behaviour
#levelname 8 Language Constraints
#levelname 9 Errors
```

You can add or reword level names by including `#levelname` statements in a user message file. See User Message Files later in this chapter.

Message Group Records

Each message level can contain a number of message groups. Usually, a message is assigned to a single group, although some messages are present in groups in two different levels (See Duplicate Warning Messages later in this section). There are ten message levels (0-9) and each level can contain any number of message groups.

The name of a message group, its description, and the level to which it belongs is defined in a set of `#define` and `#levelname` statements such as:

```
#levelname 3 Major
#define MAJ_Decl      3 Declarations and definitions
#define Maj_Ops       3 Operations
#levelname 8 ISO Conformance
#define Constraint     8 Constraint violations
```

Message Records

A message record consists of:

- the message number,
- the message group to which the warning belongs,
- the message text,
- optional reference text.

The message text can contain standard escape sequences and other constructs including:

<code>\n</code>	new line
<code>\t</code>	tab-stop
<code>\</code>	line continuation
<code>\\</code>	this is used to indicate the start of the optional verbose message text. You can choose to suppress this text when displaying the warning.

For example:

```
3305  Maj_Ops  Pointer cast to stricter alignment.\\  
REFERENCE - PH-1.8_ptrtypes
```

Duplicate Warning Messages

The same message number may have two entries in the message file associated with different message groups. The entries may have different text. The Display Highest Warning Level checkbox in the Message Personality determines which message entry is displayed. See Appendix A: Personalities.

User Message Files

You can customise the message system by introducing a user message file that can override a single message or the entire message system. User message files have the same format as the standard message file and are saved as `qac.usr.x` where `.x` is a user defined file extension. This allows you to customise messages without changing the standard message file, which may be replaced in future releases of QA C.

An easy way to create a user message file is to edit the sample file that is shipped with QA C, `qac.usr.ex`, and save it with a new file

extension. You can then remove the example messages and add new messages as required. Messages do not need to be entered in order but it may be easier to manage.

To include the user message file when displaying messages, you need to specify it in User Message Files of Message Personality>Advanced Settings. See Message Personality of Appendix A: Personalities.

Rewording Messages

To reword an existing message, add the reworded version using the same message number and group as the original message.

To reword the reference part of a message record, you will first need to copy, into your user message file, all parts of the original message record (message number, group, message text, and reference text). The display of reference message text is controlled by Message Personality>Display>Message Format.

Moving a Message to a New Group

To reword a message and give it a different group, add the reworded message with its new group.

Duplicating a Message Number at a Different Level

To add a new version of a message at a different level while retaining the original message and level, add both message entries. You will need to copy the original message entry to the user message file so that it is not superseded by the new entry.

Changing the Level or Description of a Message Group

Add a new `#define` record specifying the original group name with the new level number and/or description.

Renaming Message Levels

To rename a message level, add a `#levelname` statement that specifies the new message level. You can have a maximum of 10 message levels, numbered 0 to 9. Message levels must be added in the following format:

```
#levelname levelnumber levelname
```

where:

levelnumber a number used to organise the message levels.

levelname the name of the message level.

Message level names in the user message file will override those in the message file with the same level number.

Note:

Level 9 messages, which represent syntax or configuration errors, should not be altered (have their level or text changed), since they are used to report analysis failures.

Adding a New Message

You may wish to add new messages if you have specified Warning Calls or Metric Thresholds, or are applying secondary analysis checks. New messages should always be numbered from 5000 so as not to conflict with messages in the standard message library.

Note:

The `errwrt.exe` utility, which adds additional user messages to the `.err` file, automatically adds 5000 to the message number, enforcing this restriction.

Formatting Message Output

The format of warning messages is controlled on the command line by setting the `-format` option and on the GUI by Message Personality>Display>Message Format.

Message Display

The `-format` option accepts the following display specifiers:

<code>%f</code>	file name (as passed to <code>qac.exe</code>)
<code>%F</code>	file name (absolute, including path)
<code>%B</code>	base filename
<code>%l</code>	line number
<code>%c</code>	column number
<code>%C</code>	column number less 1 (left column is 0)
<code>%g</code>	message level
<code>%n</code>	message number (raw integer format)
<code>%N</code>	message number (zero-padded to 4 digits)
<code>%t</code>	message text
<code>%v</code>	verbose/reference text
<code>%u</code>	context message depth
<code>%^</code>	if the column is not zero and the message is not a submessage for different line or filename, update the last location, show the caret and insert a newline. It is equivalent to the following sequence (see below for additional explanation):
<pre>%?c>0% (?u==0% (%q%R (%C,) ^\n% : %?L% (% : %q%R (%C,) ^\n%) %)</pre>	

A basic message format that displayed the line number, message level, message number, and message text would look like:

```
Line:%l Level:%g Msg:%n(%t)
```

and would display message 3272 as:

```
Line:1 Level:5 Msg:3272(This expression is always true)
```

Conditional Formatting

Message formatting can also be configured so as to apply only in certain conditions. This allows you to specialise message display, for example according to the message level. You can enter a conditional

statement in the format:

```
%?<condition> %(<true condition> [%:<false condition>] %)
```

The false branch is optional and if it is not supplied, nothing is displayed. The *condition* field consists of a letter, and optionally a conditional operator and a value.

The following conditional variables can hold any legitimate value, and must be tested using a relational operator:

l	line number
c	column number
g	message level
n	message number
u	context message depth

For example, to display `Err` for level 9 messages and `Msg` for all others, the conditional statement would look like:

```
%?g>=9%(Err%:Msg%)
```

All warnings in message level 9 or higher will be generated with `Err` in the message prefix. All other levels will be generated with `Msg`.

Primary messages have a context message depth (`u`) of zero, and context messages have level 1 or greater. Thus, typical conditional processing of context messages is:

```
%?u==0%( primary message %: context message %)
```

The following conditional variables can only be truth-tested for a change in their value:

C	file name, line and column number
L	file name and line number
F	file name

For example, a simple check for a file name is:

```
%?F%(\n%f%)
```

In this example, if the file name has changed (`F`), the new file name will be displayed on a new line (`\n%f`).

The conditional variable C can be used to determine the positioning of the warning location caret (^). In the default message format, the caret is omitted for messages with the same location as the previous message. To replicate this, use the following conditional statement:

```
%?C% (%^%)
```

This statement will compare the file name, line and column number (C) of the current message with the previous message. If the location is different, the statement is considered true and the caret is displayed, followed by a new line, which is added implicitly.

Finally, properties of a message can be queried with these boolean conditional variables:

- h is message a hard error (level 9)?
- v is verbose text present?

For example, the following statement:

```
%t%?v% (\n%v)
```

will print the message text (%t) followed by the verbose text on a new line (\n%v) , if it exists (%?v).

Message Text Control

There is another set of format specifiers that control how text is handled within warning messages. These should appear at the beginning of the format string.

- %s (n) limit message size: text is truncated to n characters
- %w (n) character wrap: message text wraps at column n.
- %W (n) word wrap: message text wraps after the last word before column n.
- %i (str) indent wrapped text: when text is wrapped, it is indented with <str>.
- %q save the location for the next message (used for %F, %L, %C tests).
- %Q reset the location for the next message (%F, %L, %C will

yield true when formatting the next message).

Note: %Q takes precedence over %q.

%R(num, char) print num copies of char.

To limit the size of message output to 80 characters per line without word breaks, include the specifier %W(80) in the format string.

The %q and %Q entries can be used to override the default behaviour of %^, or to implement similar functionality from scratch.

For example:

```
%?c>0%(%?C%(%R(%C,.)^\\n%)%)%
```

will cause the following display of carat lines:

```
.....^
```

An Example Message Format

The following is a typical message format that displays the most relevant information about a warning message:

```
%W(80)%i( )%?C%(%^%)%f(%l) ++ %?h%(ERROR%:WARNING%) ++:
<=%g=(%n) %t
```

The components of this string are:

%W(80)	Message text wraps after the last word before column 80.
%i()	Each hanging indent is indented by three columns.
%?C%(%^%)	A caret is displayed if the location of the error is different than the previous error.
%f(%l) ++	The file name and line number (in brackets) are displayed, followed by '++'.
%?h%(ERROR%:WARNING%) ++:	If the message is a hard error, ERROR is displayed, otherwise WARNING is displayed, followed by '++:'.
<=%g=(%n)	The message level is displayed (preceded

by '<=') and followed by the message number (preceded by '=').

```
%t
```

The message text is displayed.

A warning message with the above format would look like:

```
int i;  
  ^  
test.c (1) ++ WARNING ++: <=2=(3408) 'i' is externally  
      visible. Functions and variables with external linkage  
      should be declared in a header file.
```

Environment Variables

You can extend QA C by adding programs to perform secondary or CMA analysis checks or generate custom reports. These programs will need to reference the following environment variables:

QACBIN

The `bin` directory, containing QA C's system files.

QACHELPPFILES

The location of the HTML help pages, explaining the rationale behind each generated message. This should be set to the `help` directory, and help files located in `\messages` sub-directory.

QACOUTPATH

The location of project output files as specified in your Folder Parameters. The default is `<QAC>\temp`.

Writing Secondary Analysis Checks

Library Usage

Some programming standards prohibit the use of certain standard library headers. The `.met` file contains a record of all `#include` statements. You can generate a warning for all instances of `#include` records that refer to these headers.

Writing Messages to the Error File

When creating a secondary analysis check, you will need to associate it with a new message number and create a message in a user message file. See The User Message File in this chapter.

When your secondary analysis program needs to generate a warning message, it should run the `errwrt.exe` utility, which will write a new message record in the `.err` file. This message will appear in your annotated source code alongside the standard QA C warnings. To run the `errwrt.exe` utility, enter:

```
errwrt prod file line [hfile] [hline] col errnum [ident]
```

`prod` QAC (`errwrt` is a utility which is also used in other PRL products.)

`file` The filename of the source file for which the error should be generated.

`line` The line number within the source file on which the error should be flagged. If the warning occurs within a header file, this will be the line number of the `#include` statement.

`hfile` The name of the header file, if any, in which the error occurs.

`hline` The line number within the header file

`col` The character position on the line where the error appears.

`errnum` Insert a number 5000 less than the intended message number as expressed in the user message file.

Message numbers below 5000 are reserved for in-built messages. `Errwrt` will automatically add 5000 to `errnum` when appending to the `.err` file.

`ident` This optional parameter will be substituted in the message text if a format string “%s” is encountered. You can supply several %s strings in the following format: %1s, %2s, %3s.

Additional parameters that are accepted by the `errwrt` utility are:

`-op` Override the QACOUTPATH environment variable.

`-hm+|-` Specify whether the message should be hidden.

Batch usage of `errwrt`

As well as the previous forms of usage, `errwrt` can be called as:

```
errwrt prod <batch_filename> -op <output_path>
```

where

`<product>` is QAC or QACPP

`<batch_filename>` names a text file containing records of the form:

`<filename> <lineno> <column> <errnum> {<message>}`

or

`<filename> <lineno> <header> <hdrline> <column> <errnum> {<message>}`

`<output_path>` is the location of the output path for all source files involved in batch error record generation.

Notes:

The QACOUTPATH environment variable is not used for this setting.

All files must share the same output path for batch operation.

Writing Custom Reports

The Reports menu contains an entry, Custom Reports, where you can add new user-defined reports that you have developed. The setup and run configuration of these reports are stored in Custom Reports of Configuration>Options.

Custom reports will usually rely on a program which processes the analysis and metrics data stored in `.met` files. They are assumed to operate only on files in the currently selected project folder and its sub-folders.

To create a custom report you will need to provide:

- a report name which will appear in the Custom Reports menu,
- a custom report executable,
- any required parameters.

In the Parameters box, you can use the following format specifiers:

`%Q` Specifies the product identifier (`QAC` or `QACPP`) that is required for many of the product components, e.g. Warning Listing report (`prjdisp.exe`) or Message Browser (`viewer.exe`).

`%P[+]` Formats and passes the personality settings of the root folder, `settings.via`.

With the `+` postfix, it prepends `-via` to the entry.

Note that when running a custom report for a folder branch, only the personality entries for the parent folder will be used.

`%L[+]` `[-list] <filelist.lst>`

The `-list` option passes a list of files to be read by the program executable. This option will create `<filelist.lst>` in the temp directory, and specifies a list of all source files under the selected folder in your project with their corresponding output directories. See Appendix B: Configuration Options.

`%R[+]` `[-file] <TEMPDIR\<exe_name>.html>`

Formats an output file, located in the product temp directory, made up of the basename (`<exe_name>`) of the report executable with a `.html` extension.

There are two possible modes of operation for custom reports:

1. Run as a batch process, generate an output file, and display through an editor or browser. For example, `prjdsp.exe` operates in this manner.
2. Run as an interactive program. For example, the Message Browser (`viewer.exe`) operates in this manner.

The decision on which mode will apply is based on whether there is a `-file` entry in the final command line of the custom report. If this parameter is detected, it signifies that an output file will be generated, and thus, that this output file will be displayed as the report result. If this parameter entry is not found, then it is assumed that the external report program has its own display mechanism, as is the case with the Message Browser.

Note:

When running a custom report for a folder branch, only the personality entries for the parent folder will be used.

APPENDIX A

Personalities

A personality is a file containing a group of configuration options that define how QA C analyses source code and displays annotated source code. Each folder of your project can have different personalities, which are specified in the folder parameters.

Underlying the presentation of the personality configuration in the GUI is a set of command line configuration options, stored in the personality files. In the following descriptions of the personalities, reference is made to these command line options which are documented in Appendix B: Configuration Options. However the correspondence between a personality setting and a configuration option is not always direct and obvious. An understanding of options is only necessary for users who need to configure QA C for command line operation.

Message Personality

The Message Personality controls the display of warning messages in annotated source code.

Each warning message is contained in a message group according to the type of warning detected. Each message group is assigned to a message level. Message levels are numbered from zero to nine.

QA C is shipped with seven message levels. You can add new message level names or reword existing levels by using a user message file. See User Message Files in Chapter 10: Advanced Topics.

The default personality displays all messages detected by analysis. You can, however, choose to suppress the display of any combination of message levels or warning messages. This allows you to focus on important issues.

You can suppress messages by:

- warning level by selecting the level folder and pressing ctrl-enter or right-clicking and choosing Switch Message Level Off,
- individual messages by double-clicking the message.

If you toggle a message group, the individual messages will be toggled. If you toggle a message level, the level itself is toggled.

Note:

If you change the settings of the **Message Personality**, you are only changing the options that affect the display of messages. As a result, you will not need to re-analyse your source code.

The standard message level configuration is described below. The allocation of messages within these levels is in some cases arbitrary but they provide a useful framework to distinguish messages on the basis of their relative importance and application.

Information (0)

Information level warnings with the lowest level of importance.

Minor (2)

These messages usually draw attention to issues of good practice rather than constructs that are necessarily dangerous.

Major (3)

Source code constructs that could be dangerous such as references to uninitialised variables, calls to undeclared functions, or dangerous implicit casts.

Portability (6)

Identifies issues relating to conformance limits and implementation-defined aspects of, and language extensions to, ISO C.

Undefined Behaviour (7)

Identifies constructs that are either explicitly or implicitly undefined in the ISO C standard.

Language Constraints (8)

Identifies constructs that are syntactically correct but dangerous because their behaviour is not fully defined by the ISO C standard.

Errors (9)

These messages indicate conditions in which QA C is unable to parse. This may be the result of a configuration problem or simply that the code contains syntax errors or some language variation which QA C does not recognize.

Messages from the higher warning levels are the most severe and it is usually a matter of priority to resolve these issues first. It is unwise to pay too much attention to other warnings without resolving all level 9 messages.

You can change the message structure of QA C by using a user message file. See Customising the Message System of Chapter 10, Advanced Topics.

Personality Setting	Option
Warning Messages	
Message Groups	
Information	-su 0
Minor	-su 2
Major	-su 3
Portability	-su 6
Undefined Behaviour	-su 7
Language Constraints	-su 8
Errors	-su 9
Advanced Settings	
User Message Files	
User Message File	-up -usr
Secondary Analysis Command String	N/A
Message Expansion List	-emhm
Save Message Settings By	
Suppressed Messages	-n
Selected Messages	-o

Display	
Message Display	
Annotated source display format	
Messages Only	-m
Source Lines with Warnings	-one
Full Source with Warnings	-m-,-one-
Message Filtering	
Create Analysis Summary	-summ
Display Header Warnings	-hdr
Show Suppressed Warnings	-hw
Display Line Number	-l
Display Highest Warning Level	-st
Maximum Message Occurrence	-max
Message Format	-format

Analyser Personality

Personality Setting	Option
Project Headers	
Header Includes	-i
Suppress Output	-q
Project Macros	
Project Macro Defines	-d
Style	
Coding Style	
Brace Style	-sty
Source Code Tab Spacing	-t
Code Indent Level	-il
Maximum Line Length	-mll
Analysis Processing	
Processing Options	
Encoding	-en
Maximum No of Hard Errors to Abort	-maxerr
Preprocessed Output	
Include filename and line number in preprocessed listing	-ppf

Implicit Conversion Message Options	
Generate unsigned to larger signed messages for wider types	-ss
Generate narrowing messages for same-width types	-sr
Warning Calls	
Warning Calls	-wc
Pragma Blocks	
Pragma Block Ignore	-spragma
Metrics	
Metrics Thresholds	-thresh
Comment Count	-co
Output Options	
Display Metrics	-met
Metrics after Preprocessing	-ppm
K&R Compatibility	
K&R to ANSI Portability	-k+r

Compiler Personality

Personality Setting	Option
System Headers	
System Header Includes	-i
Suppress Output	-q
System Macros	
System Macro Defines	-d
Data Types	
Data Type Interpretation	
Right Shift Behaviour	-ar
Bit Field Interpretation	-bits
Char Interpretation	-u
Fold Plain char to signed/unsigned type	-fpc
Intrinsic Types	
size_t	-it size_t
ptrdiff_t	-it ptrdiff_t
wchar_t	-it wchar_t

Type Size and Alignment	
Size	-s
Alignment	-a
Identifiers	
Identifiers	
Internal Identifier Significant Length	-na
External Identifier Significant Length	-xn
Ignore External Identifier Case	-xc
Extensions	
Extension Classes	-ex
Additional Include File	-fi
Miscellaneous	
Ignore whitespace between '\' and new line	-sl

APPENDIX B

Configuration Options

Most options can be abbreviated and in the descriptions below, an alternative abbreviated version is also shown. Option names are not case sensitive. For example:

```
-align
-ALIGN
-a
```

are all equivalent.

Also shown is the syntax of any arguments and the context in which the option is set within the GUI. Syntax within the GUI is the same except that the option name is not included.

-a, -align

Syntax:	<code>-align [type]=[bytes]</code>
Default:	See chart below.
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Data Types> Type Size and Alignment>Alignment
Function:	Determines the alignment requirement of each primitive C type.

The alignment of datatypes may be constrained by your hardware and compiler. For example, your environment may allocate a `char` at any memory address but may require an `int` to be allocated on a four-byte boundary. Such constraints impose restrictions on the validity of pointer conversions. For example, it might be valid to cast an `int *` pointer to a `char *` pointer but not the reverse. QA C will generate warnings to identify dangerous pointer casts.

In the GUI, alignment can be set using a trackbar and editbox. The available size and alignment values are constrained by the ISO rules

on integral size ordering.

QA C also uses type alignment values to calculate the size of structures. Alignment is often related to the size of a data type, see the `-size` option.

Type	Size (bits)		Size Rules	Alignment Default (bytes)
	Min	Default		
char	8	8	<code><=short</code>	1
short	16	16	<code><=int</code>	2
int	16	32	<code><=long, <=8 bytes</code>	4
long	32	32	<code><= long long, <=16 bytes</code>	4
Long long	32	64	<code><=16 bytes</code>	8
float	32	32	<code><=16 bytes</code>	4
double	64	64	<code><=16 bytes</code>	8
ldouble	64	64	<code><=16 bytes</code>	8
codeptr		32	e.g. <code>int (*) ()</code>	4
dataptr		32	e.g. <code>char *</code>	4

-ar, -arithrsh

Syntax:	<code>-arithrsh {+ -}</code>
Default:	<code>-arithrsh-</code>
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Data Types> Data Type Interpretation>Right Shift Behaviour
Function:	<p>Defines whether the right shift operator <code>>></code> performs an arithmetic (<code>-arithrsh+</code>) or logical (<code>-arithrsh-</code>) shift.</p> <p>The action of the right shift operator (<code>>></code>) is one of the features of ISO C that is classed as “implementation-defined”. It can be interpreted as either an arithmetic shift or a logical shift. This distinction is only significant when applied to signed data types.</p> <p>The expression <code>(-1)>>1</code> has the value <code>-1</code> when the</p>

	shift is arithmetic but a large positive value when the shift is logical.
--	---

-bits, -bitsigned

Syntax:	<code>-bitsigned {+ -}</code>
Default:	<code>-bitsigned-</code>
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Data Types> Data Type Interpretation>Bit-Field Interpretation
Function:	Defines whether plain integer bit-fields are treated as signed int (<code>-bitsigned+</code>) or unsigned int (<code>-bitsigned-</code>).

The ISO C standard states that a bit-field has a type that is either `int`, `unsigned int`, or `signed int`. The high-order bit of an `int` bit-field may or may not be treated as a sign bit. This is “implementation-defined”, although many compilers treat plain `int` bit-fields as `signed int`, especially when the compiler does not support the `signed` keyword. In the following declaration:

```
struct
{
    signed int sbit : 3;
        /* holds values between -4 and +3 */
    unsigned int ubit : 5;
        /* holds values between 0 and 31 */
    int bit : 4;
        /* may hold values between -8 and +7
           or between 0 and 15 */
} x;
```

`x.bit` can only safely be assumed to hold values between 0 and 7.

-co, -comment

Syntax:	<code>-comment {a,n,i}</code>
Default:	<code>-comment n</code>
Program:	<code>qac.exe</code>

GUI:	Analyser Personality>Metrics>Comment Count
Function:	Determines how QA C calculates the comment density metric (STCDN).

To calculate this metric, QA C counts the number of characters in comments. QA C will count characters from:

- all comments, (a),
- all comments except for those from headers, (n),
- inline or internal comments (i). These are comments within functions and comments that annotate a line of code (comments that are on the same line as code at file scope).

-cmaf, -crossmoduleanalysisfile

Syntax:	<code>-cmaf filename</code>
Default:	<code>no entry</code>
Program:	<code>qac.exe, errdsp.exe, pal.exe, errsum.exe</code>
GUI:	N/A
Function:	<p>For generation tools, <code>qac.exe</code> and <code>pal.exe</code>, this option specifies the output files into which project-based <code>err</code> and <code>met</code> information are delivered. For display tools, it specifies the location where this same information is located.</p> <p>When operated through the GUI, the <code>-cmaf</code> entry is constructed from the root folder output path and the name of the root folder.</p>

-d, -define

Syntax:	<code>-d ident [=value]</code>
Default:	<code>no entry</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Project Macros Compiler Personality>System Macros

Function:	Defines macros for use in QA C parsing.
------------------	---

The compilation of source code is often controlled by macros supplied to the compiler through command line options. There are three ways to define a macro:

Configuration Option:	Equivalent Source Code Line:
<code>-d ident</code>	<code>#define ident 1</code>
<code>-d ident=value</code>	<code>#define ident value</code>
<code>-d ident=</code>	<code>#define ident</code>

C source code often includes constructs that are not part of the ISO C standard. Compiler vendors frequently introduce new keywords in order to extend the language. To be able to parse these extensions, you can use the `_ignore` macro, which instructs QA C to ignore a single word or section of code.

There are five ways of doing this:

<code>-d ident=_ignore</code>	ignores the specified identifier and the next token.
<code>-d ident=_ignore_semi</code>	ignores the identifier and everything up to and including the next <code>;</code> .
<code>-d ident=_ignore_paren</code>	ignores the specified identifier and the following block of code. The block may be of any length and may be surrounded by <code>(...)</code> , <code>[...]</code> , or <code>{...}</code> .
<code>-d ident=_ignore_N</code>	ignores the specified identifier and the next <code>N</code> tokens.
<code>-d ident=_ignore_at</code>	when the specified identifier is preceded by an <code>'@'</code> character, both tokens are ignored.

The ignore operation is processed before any other macro substitution takes place.

Note:

The macro `_munch` is a deprecated alternative to `_ignore`.

Passing a `-define` entry through the configuration system will always strip out quotes, and it is thus not possible to define a quoted string entry this way. One way to circumvent this problem is to pass the string entry through a function macro such as:

```
-d STR(x)=#x
-d STRNG=STR(abcd)
```

-e, -echo

Syntax:	<code>-echo text</code>
Default:	no entry
Program:	<code>qac.exe</code> , <code>errdsp.exe</code> , <code>prjdsp.exe</code>
GUI:	N/A – console output is ignored
Function:	Specifies that <i>text</i> is echoed to the console. This option can be used to show the configuration files that are being referenced.

-emhm, -expandmultihomedmessages

Syntax:	<code>-expandmultihomedmessages message_number_list</code>
Default:	no entry
Program:	<code>errdsp.exe</code> , <code>prjdsp.exe</code>
GUI:	Message Personality>Advanced Settings>Message Expansion List
Function:	Expands the selected set of messages, so that an instance of the message will appear in each location represented by its list of sub-messages, rather than generating a single message in the primary location. For each of these message instances, the message takes on a submessage list representing itself and all other sub-messages in the original message.

	<p>The purpose of this option is to highlight, directly in code annotated source, the problems (or the contributing incompatibilities) associated with that source file.</p> <p>There is no parser default, and the default set of candidate messages, as configured in the GUI when the option is turned on, is:</p> <pre>-emhm 1500,1501,1506,1507,1508,1509,1510, 1512,1513,1515,1516,1517,1520,1525,1526, 1527,1528,1529.</pre>
--	---

-en, -encoding

Syntax:	<code>-encoding {ASC , EUC, NEWJ, OLDJ, NECJ, SJ}</code>
Default:	<code>-encoding ASC</code>
Program:	<code>qac.exe, errdsp.exe, prjdsp.exe, viewer.exe</code>
GUI:	Analyser Personality>Analysis Processing>Encoding
Function:	Specifies the character set. The default is ASC, representing ASCII encoding. This option only applies to wide character formatting environments such as Japanese systems.

-ex, -extensions

Syntax:	<code>-extensions {ANSIPC, PC, ASM, C++, DOLLAR, LONGLONG, ALL}</code>
Default:	<code>no entry</code>
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Extensions>Extension Classes
Function:	Specifies the extension class to be used in analysis.

QA C supports a variety of compiler extensions that may be enabled using this option. On command line, you can enable all the extensions by setting the option to `-ex+` or `-ex all`.

QA C recognises the following extension classes:

Extension:	Behaviour:
ansipc	<ul style="list-style-type: none"> • Filenames appearing in <code>#include</code> directives are mapped to lower case. • Backslash (<code>\</code>) is permitted within filepaths. • The keywords <code>__near</code>, <code>__far</code>, <code>__huge</code>, <code>__cdecl</code>, <code>__pascal</code>, and <code>__fortran</code> are recognised and ignored. A warning message is issued.
PC	<ul style="list-style-type: none"> • The keywords <code>near</code>, <code>far</code>, <code>huge</code>, <code>cdecl</code>, <code>pascal</code>, and <code>fortran</code> are recognised and ignored. A warning message is issued.
asm	<ul style="list-style-type: none"> • Recognises the assembler code directives <code>#asm</code>, <code>#endasm</code> and <code>asm("string")</code>; . If assembly language within <code>#asm</code> contains text that cannot be treated as raw C tokens, syntax errors will still occur. This is only likely if single ' or " characters appear.
C++	<ul style="list-style-type: none"> • The reference operator <code>&</code> is recognised in declarations but has no semantic effect • C++ style comments are allowed. Text between <code>//</code> and a new line is treated as a comment.
dollar	<ul style="list-style-type: none"> • <code>\$</code> is legal in identifiers.
LONGLONG	<ul style="list-style-type: none"> • Type <code>"long long"</code> is recognised as an extended data type. Properties of the type and conversion rules will conform to the principles defined in the ISO C99 standard. • If this option is not set, all references to type <code>"long long"</code> result in generation of message 1027 and the type is treated as a synonym for type <code>"long"</code>. • The rules by which the type of a numeric integer constant are determined conform to ISO C99. Unsuffix decimal values which are too large to fit in a signed long are of type <code>"long long"</code> rather than <code>"unsigned long"</code>, if they fit.

-fi, -forceinclude

Syntax:	<code>-forceinclude filename</code>
Default:	no entry
Program:	qac.exe
GUI:	Compiler Personality>Extensions>Additional Include File
Function:	Causes <i>filename</i> to be implicitly included at the beginning of each file. This method of “force-including” a file is sometimes a useful way of defining macros and other project settings.

-file

Syntax:	<code>-file filename</code>
Default:	no entry
Program:	errdsp.exe, prjdsp.exe
GUI:	N/A
Function:	Generates output to <i>filename</i> . If the full path of the file is not given, this file will be created in the path specified by QACOUTPATH or the <code>-op</code> option.

-forget, -forgetall

Syntax:	<code>-forgetall option</code>
Default:	no entry
Program:	qac.exe, errdsp.exe, prjdsp.exe, viewer.exe
GUI:	N/A
Function:	Resets previous configuration settings for <i>option</i> . Some options are cumulative in their action in that successive usage of the option adds extra settings rather than replacing the previous. See <code>-i</code> , <code>-d</code> , <code>-n</code> , <code>-o</code> , <code>-su</code> , <code>-q</code> , and <code>-wc</code> .

For example:

```
qac -forget i -iC:\mypath source.c
```

This command line will cause the analyser to disregard any include paths previously defined and to use only the path C:\mypath.

-format

Syntax:	<code>-format <i>string</i></code>
Default:	no entry
Program:	errdsp.exe, viewer.exe
GUI:	Message Personality>Display>Message Format
Function:	<p>Determines how warning messages will be displayed in annotated source code. When the <code>-format</code> option is used, <code>-ref</code> and <code>-text</code> options are ignored, and usage of the <code>-line</code> option is restricted.</p> <p>See Formatting Message Output of Chapter 10: Advanced Topics for a detailed explanation.</p>

If no `-format` option is specified, certain default formatting will apply:

For the Message Browser (viewer.exe) default is:

```
%?u==0%(?C%(%^%)%)%?u==0%(?h%(Err%:Msg%)%:-->)(%g:%N) %R(%u, )%t
```

For errdsp.exe, default is dependent on other options:

```
format=MessagesOnly ? "" : "%?C%(%^%)";
if (MessagesOnly || Line)
{
    format += "%F(%l,%c): ";
}
format += TextOnly ? "" :
"%?u==0%(?h%(Err%:Msg%)%:-->)(%g:%N) ";
format += "%R(%u, )%t";
format += References ? "%?v%(\n?v%)" : "";
```

-fpc, -foldplainchar

Syntax:	<code>-foldplainchar {+ -}</code>
Default:	<code>-foldplainchar-</code>
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Data Types> Char Interpretation> Fold plain char to unsigned type
Function:	Determines if plain char is treated as signed char (<code>-foldplainchar+</code>) or unsigned char (<code>-foldplainchar-</code>).

Although the `plain char` type may be implemented as signed or unsigned, it is important to understand that it remains a distinct datatype and is strictly a different type from either signed char or unsigned char. This means that even if `plain char` is implemented as a signed type, it is not legitimate to initialise an array of type signed char with a literal character string (e.g. "ABC"). The only way to achieve this would be to apply an explicit cast (`signed char *`) to the literal.

This distinction is often considered to be pedantic and is seldom observed strictly by compilers. If this option is set, a conversion from a "pointer to char" to either "pointer to unsigned char" or "pointer to signed char" will not raise the constraint message 0563 "[C] Right operand of assignment does not have a compatible pointer type."

-h, -help

Syntax:	<code>-help</code>
Default:	<code>no entry</code>
Program:	<code>qac.exe</code> , <code>errdsp.exe</code> , <code>prjdsp.exe</code> , <code>viewer.exe</code> , <code>errsum.exe</code>
GUI:	N/A
Function:	Displays help description for all options.

-hdr, -hdrsuppress

Syntax:	<code>-hdrsuppress {+ -}</code>
Default:	<code>-hdrsuppress-</code>
Program:	<code>errdsp.exe, prjdsp.exe, viewer.exe</code>
GUI:	Message Personality>Display> Display Header Warnings
Function:	<p>With <code>-hdrsuppress+</code>, all warnings detected in header files are suppressed, except for level 9 messages (which cannot be suppressed).</p> <p>The default setting will include all messages (not turned off through <code>-n</code>, <code>-o</code>, <code>-q</code> or other diagnostic suppression means) arising from parsing of header files. In annotated source output these will immediately follow the <code>#include</code>.</p>

Often a large number of warnings are generated from source code in header files. These files may be library header files or project headers that generate warnings not considered significant. This option allows you to suppress the messages from your annotated source code.

Note:

The `-q` option also suppresses header warnings. It differs from `-hdr` in that it suppresses not just the display of messages, but also stops the generation of messages to the `.err` file by `qac.exe`. The `-q` option may also be applied selectively to particular paths or files.

-hm, -hiddenmessage

Syntax:	<code>-hiddenmessage {+ -}</code>
Default:	<code>-hiddenmessage-</code>
Program:	<code>errwrt.exe</code>
GUI:	N/A
Function:	With <code>-hiddenmessage+</code> , a flag is set on the created error record to mark the diagnostic as suppressed.

This is equivalent to the behaviour for messages generated under the control of `#pragma PRQA_MESSAGES_OFF`.

Notes:

Hidden messages can be displayed by utilities with the `-hw+` option.

The suppression system does not use this technique to mark warnings as suppressed, this is only used by the `#pragma PRQA_MESSAGES_OFF` alternative.

-html

Syntax:	<code>-html {+ -}</code>
Default:	<code>-html-</code>
Program:	<code>errdsp.exe, prjdsp.exe</code>
GUI:	N/A
Function:	By default, annotated source code is created in text format (<code>-html-</code>). When set to <code>-html+</code> , annotated source code is created in HTML format. This is required in order that the Warning Listing report can form links to individual annotated source output files. The GUI has menu actions to create either annotated or html annotated source code on demand.

-hw, -hiddenwarnings

Syntax:	<code>-hiddenwarnings {+ -}</code>
Default:	<code>-hiddenwarnings-</code>
Program:	<code>errdsp.exe, prjdsp.exe, viewer.exe, errsum.exe</code>
GUI:	Message Personality>Display> Show Suppressed Warnings
Function:	With <code>-hiddenwarnings+</code> , warning messages that have been suppressed through message suppression techniques are displayed. See Message Suppression in Chapter 3, Configuring QA C.

Message diagnostics can be suppressed through several means (suppression entry in code comment, `-hiddenmessage+`, or `#pragma` directives). With the `-hiddenwarnings` option set, all messages suppressed through any of these means will be turned back on. For the Message Browser, these warning diagnostics can be turned off again during usage.

Note:

These diagnostics will still not be displayed if the relevant message numbers are suppressed by either `-n` or `-o` options.

-i, -include

Syntax:	<code>-i path</code>
Default:	<code>no entry</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Project Headers Compiler Personality>System Headers
Function:	Specifies a search path for header files.

-il, -indentlevel

Syntax:	<code>-indentlevel value</code>
Default:	<code>-indentlevel 0</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Style>Code Indent Level
Function:	Specifies the amount of space used to indent code. QA C will issue message 2215 if the code indentation differs from this value. With the value set to zero (default), QA C will not enforce a specific indentation depth but it will identify indentation that is inconsistent and will issue message 2201.

-it, -intrinsictype

Syntax:	<code>-intrinsictype {size_t ptrdiff_t wchar_t}=datatype</code>
Default:	<code>-intrinsictype "size_t=unsigned int" -intrinsictype "ptrdiff_t=long" -intrinsictype "wchar_t=unsigned char"</code>
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Data Types>Intrinsic Types
Function:	<p>Determines the underlying type associated with the <code>sizeof</code> operator (<code>size_t</code>), the result of pointer subtraction (<code>ptrdiff_t</code>), and the wide character data type (<code>wchar_t</code>).</p> <p>QA C will generate a level 9 warning if these are not consistent with any typedef settings encountered in the source code. See Setting Implementation Defined Types of Chapter 3: Configuring QA C.</p>

-k+r

Syntax:	<code>-k+r [sun, a, b, c, e, g, i, l, n, p, s]</code>
Default:	<code>no entry</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>K&R Compatibility
Function:	Defines ISO language features that should be avoided if using a K&R compiler.

If you want your code to be compliant to K&R standards you will need to restrict your usage of the C language to exclude all these features.

Specifying a construct with this option will generate one of the K&R messages when it encounters that construct. If your code does not need to be compatible with pre-ISO standards, do not set this option.

The following arguments can be specified in any order:

Option Value:	Behaviour:
<i>sun</i>	enforces compatibility with the Sun K&R cc compiler under SunOS 4.1 or earlier. This is equivalent to: <code>-k+r acegipstux</code> .
<i>a</i>	initialising locally declared structures and unions.
<i>b</i>	bit-field
<i>c</i>	string concatenation
<i>e</i>	<code>#error</code>
<i>g</i>	glue operator <code>##</code>
<i>i</i>	signed keyword
<i>l</i>	<code>#elif</code>
<i>n</i>	enumerated types
<i>p</i>	function prototypes
<i>s</i>	stringify operator <code>#</code>
<i>t</i>	<code>const</code> and <code>volatile</code>
<i>u</i>	unary <code>+</code>
<i>v</i>	<code>void</code> keyword
<i>x</i>	initialising <code>extern</code> variables.

-l, -line

Syntax:	<code>-line {+ -}</code>
Default:	<code>-line-</code>
Program:	<code>errdsp.exe</code> , <code>viewer.exe</code>
GUI:	Message Personality>Display>Display Line Numbers
Function:	<p>With <code>-line+</code>, the filename, line and column number is displayed with the warning message.</p> <p>If the <code>-format</code> option is specified, the <code>-line</code> option is ignored for message formatting.</p> <p>The Message Browser uses <code>-line+</code> to enable display of line numbers beside the source code.</p>

-list

Syntax:	<code>-list filelist.lst</code>
Default:	<code>no entry</code>
Program:	<code>prjdsp.exe, viewer.exe, pal.exe, errsum.exe</code>
GUI:	N/A
Function:	Passes a list of files to various utilities and external processes, including the Message Browser and the Warning Listing report.

In the GUI, `filelist.lst` is generated by iterating through all selected files, or a selected folder including all of its files and subfolders. It is automatically generated for the Message Browser and the Warning Listing report, and when used with the `%L` expansion key in Custom Reports and CMA Analysis (always performed on the root folder). For each folder, it contains the folder's output path followed by the constituent list of source files. It is typically composed as follows:

```
-op"<output_path1>"
"<source_path_and_file1>"
"<source_path_and_file2>"
"<source_path_and_file3>"
"<source_path_and_file4>"
-op"<output_path2>"
"<source_path_and_file5>"
"<source_path_and_file6>"
"<source_path_and_file7>"
```

The `-outputpath` option is required to locate the `.err` and `.met` files for each of the subsequent source files.

Note:

The file is referred to as `filelist.lst`, although it will in fact be a unique filename in `QAC\temp`. It is later renamed to `filelist.lst` for convenience.

-m, -messagesonly

Syntax:	<code>-messagesonly {+ -}</code>
----------------	----------------------------------

Default:	<code>-messagesonly-</code>
Program:	<code>errdsp.exe</code>
GUI:	Message Personality>Display>Messages Only
Function:	<p>With <code>-messagesonly+</code>, annotated source code is generated showing only detected warning messages. The location of each message is recorded but the source line is not displayed.</p> <p>If the <code>-format</code> option is specified, only warning messages will be displayed but all message formatting will be determined by <code>-format</code>.</p>

-max, -maxcount

Syntax:	<code>-maxcount value</code>
Default:	<code>-maxcount 0</code>
Program:	<code>errdsp.exe, prjdsp.exe</code>
GUI:	Message Personality>Display> Maximum Occurrences of each Message
Function:	Introduces a limit on the number of times that any particular warning message is displayed. The default of zero indicates no limit is applied.

-maxerr, -maxerrors

Syntax:	<code>-maxerrors value</code>
Default:	<code>-maxerrors 0</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Analysis Processing> Maximum No of Hard Errors to Abort
Function:	Determines the number of hard errors that will cause the analysis to stop. The default of zero indicates that the parser will attempt to continue to completion regardless of hard errors encountered.

	The parser will issue return code 1 when any hard errors are encountered. This setting merely decides whether parsing will continue to source completion.
--	---

-mll, -maxlinelength

Syntax:	<code>-maxlinelength value</code>
Default:	<code>-maxlinelength 0</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Style> Maximum Line Length
Function:	<p>Specifies the maximum physical line length in source code, before the line-splicing phase of translation. Permitted values are zero (for no checking of maximum physical line), or any number 50 or greater.</p> <p>Using the GUI spinedit control, any value entered between 1-49 will be automatically converted to 50, the minimum check value.</p>

-met, -metrics

Syntax:	<code>-metrics {+ -}</code>
Default:	<code>-metrics+</code>
Program:	<code>qac.exe, errdsp.exe</code>
GUI:	Analyser Personality>Metrics>Display Metrics
Function:	<p>This option serves two distinct purposes:</p> <p>In analysis (<code>qac.exe</code>),</p> <ol style="list-style-type: none"> With <code>-metrics+</code>, analysis data is written to the metric output file. With <code>-metrics-</code>, a metric output file is generated but will be empty.

	<p>In display of annotated source (errdsp.exe),</p> <ol style="list-style-type: none"> 1. With <code>-summary-</code>, <code>-metrics+</code> adds a table showing the number and density (number per line of code) of warnings, by warning level. 2. With <code>-summary+</code>, <code>-metrics+</code> displays a count of the number of occurrences of each warning message.
--	--

-n, -no, -nomsg

Syntax:	<code>-nomsg message_number_list</code>
Default:	no entry
Program:	errdsp.exe, prjdsp.exe, viewer.exe, errsum.exe
GUI:	Message Personality> Advanced Settings> Save Message Settings By>Suppressed Messages
Function:	Suppresses the display of specified messages.

Message numbers can be specified individually, or as a list, range or a combination of both:

```
-nomsg 2356
-nomsg 1234,1236,1240
-nomsg 2111,2200-2300
```

-na, -namelength

Syntax:	<code>-namelength value</code>
Default:	<code>-namelength 31</code>
Program:	qac.exe
GUI:	Compiler Personality>Identifiers> Internal Identifier Significant Length
Function:	<p>Determines the number of characters in valid internal identifiers.</p> <p>QA C will generate message number 779 for each internal identifier it encounters that is not distinct</p>

	within the number of characters you have specified.
--	---

The ISO C Standard states that compilers should treat at least the first 31 characters of internal identifiers as significant. Some compilers may distinguish identifiers in more than 31 characters but it is not advisable to depend on this extension. Programming standards may choose to restrict the declaration of identifiers so that they are distinct within a smaller number of characters.

-nosort

Syntax:	<code>-nosort {+ -}</code>
Default:	<code>-nosort-</code>
Program:	<code>errdsp.exe</code>
GUI:	N/A
Function:	<p>This option controls the sorting of messages.</p> <p>When annotated source code is generated with the <code>-messagesonly+</code> option, <code>-nosort+</code> will display warning messages as they are encountered during parsing. Messages in the error list of the Analysis Status Window are ordered in this way.</p>

-nrf, -namerulefile

Syntax:	<code>-namerulefile <i>rulefile</i></code>
Default:	<code>no entry</code>
Program:	<code>pal.exe</code>
GUI:	Message Personality>Advanced Settings>Secondary Analysis Setup
Function:	Identifies the file that contains rules for Naming Convention checking in secondary analysis. See Naming Convention Checking of Chapter 4 and Appendix I.

-o, -only

Syntax:	<code>-only message_number_list</code>
Default:	no entry
Program:	errdsp.exe, prjdsp.exe, viewer.exe, errsum.exe
GUI:	Message Personality> Advanced Settings> Save Message Settings By>Selected Messages
Function:	Displays only the specified warning messages. Message numbers can be specified individually, or as a list, range or a combination of both. For example: <ul style="list-style-type: none"> -o 2356 -o 1234,1236,1240 -o 2111,2200-2300

-one, -onelineonly

Syntax:	<code>-onelineonly {+ -}</code>
Default:	-onelineonly-
Program:	errdsp.exe
GUI:	Message Personality>Display> Source Lines with Warnings
Function:	With <code>-onelineonly+</code> , annotated source code is generated that only includes source code lines that have warning messages.

-op, -outputpath

Syntax:	<code>-outputpath path</code>
Default:	%OUTPUTPATH% or no entry
Program:	qac.exe, pal.exe, errdsp.exe, prjdsp.exe, viewer.exe, errsum.exe
GUI:	Folder Parameters>Output File Path

Function:	<p>Determines the location in which output files are generated.</p> <p>During analysis, the output path determines the location to which all output files will be generated. When displaying the Message Browser, annotated source code or generating the Warning Listing report, the output path determines the location of the <code>.err</code> files.</p>
------------------	---

Note:

In command line usage, the output path is usually set by the `QACOUTPATH` environment variable unless overridden by this option.

-ppf, -ppfilename

Syntax:	<code>-ppfilename {+ -}</code>
Default:	<code>-ppfilename-</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Analysis Processing> Include filename and line numbers in preprocessed listing
Function:	With <code>-ppfilename+</code> , comments are introduced into the preprocessed file which indicate the source file and line number from which the code was derived. This option only applies in conjunction with <code>-pplist</code> .

-ppl, -pplist

Syntax:	<code>-pplist {+ -}</code>
Default:	<code>-pplist-</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Analysis Processing> Preprocessed Output
Function:	With <code>-pplist+</code> , a preprocessed file is generated in the output directory with a <code>.i</code> file extension.

-ppm, -ppmetrics

Syntax:	-ppmetrics {+ -}
Default:	-ppmetrics-
Program:	qac.exe
GUI:	Analyser Personality>Metrics> Metrics After Preprocessing
Function:	<p>This option is designed to be used with file metrics, and governs the inclusion of header file information in the calculation of metrics:</p> <p>-ppmetrics+ results in all code/tokens in included header files being used when calculating metrics.</p> <p>-ppmetrics- results in code/tokens in included header files being ignored when calculating metrics.</p>

Note:

The setting of this option is ignored by all function metrics except STLIN. Function metrics, STLIN excepted, always incorporate all code/tokens in included header files.

All metrics, except STLIN and STTPP, ignore code that is disabled on evaluation of preprocessor directives. That is, both the code in the source file and #include'd files inside conditional preprocessor directives is ignored for all metrics except STLIN and STTPP.

For example:

```
#define A 10
#if 0
#include <stdio.h> /* Ignored by all except STLIN, STTPP */
int i;             /* Ignored by all except STLIN, STTPP */
#else
long i;
#endif
```

-q, -quiet

Syntax:	-quiet { <i>filename</i> <i>path</i> }
Default:	no entry

Program:	<code>qac.exe, pal.exe</code>
GUI:	Compiler Personality>System Headers>Suppress Output Analyser Personality>Project Headers>Suppress Output
Function:	Suppresses the generation of output to the <code>.err</code> and <code>.met</code> files from the specified header file or header files in the specified path. This will have the effect of suppressing warning messages as well as significantly reducing the size of output files. See Suppressing Output from Header Files in Chapter 4: Annotated Source Code.

-ref, -references

Syntax:	<code>-references {+ -}</code>
Default:	<code>-references-</code>
Program:	<code>errdsp.exe, prjdsp.exe</code>
GUI:	Message Personality>Display>Show References
Function:	<p>With <code>-references+</code>, the optional verbose/reference text specified in the message file is displayed. Messages may include additional explanation or ISO Standard references, and user message files may contain local programming standard references.</p> <p>If the <code>-format</code> option is specified, the <code>-references</code> option will have no effect. All message formatting will be determined by the <code>-format</code> string, including the <code>%v</code> specifier for displaying verbose/reference text.</p>

-rem, -remark

Syntax:	<code>-remark <i>comment</i></code>
Default:	<code>no entry</code>
Program:	<code>all binaries</code>
GUI:	N/A
Function:	With <code>-remark</code> , comments can be written into configuration files. As an alternative, comments can be

	included in configuration files by prefixing the comments with an asterisk.
--	---

-s, -size

Syntax:	<code>-size [type]=[bits]</code>
Default:	See chart below.
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Data Types> Type Size and Alignment>Size
Function:	Defines the type sizes (measured in bits) implemented in your compiler environment.

The following table lists the minimum and default type sizes, default alignment, and sizing rules for each type. The alignment may be set using the `-align` option.

In the GUI, the size can be set using a trackbar and editbox. The available size and alignment values are constrained by the ISO rules on integral size ordering.

Type	Size (bits)		Size Rules	Alignment Default (bytes)
	Min	Default		
<code>char</code>	8	8	<code><=short</code>	1
<code>short</code>	16	16	<code><=int</code>	2
<code>int</code>	16	32	<code><=long, <=64 bits</code>	4
<code>long</code>	32	32	<code><= long long,</code> <code><=128 bits</code>	4
<code>long long</code>	32	64	<code><=128 bits</code>	8
<code>float</code>	32	32	<code><=128 bits</code>	4
<code>double</code>	64	64	<code><=128 bits</code>	8
<code>ldouble</code>	64	64	<code><=128 bits</code>	8
<code>codeptr</code>	16	32	<code>e.g. int (*) ()</code>	4
<code>dataptr</code>	16	32	<code>e.g. char *</code>	4

-set, -settings

Syntax:	-settings {+ -}
Default:	-settings-
Program:	qac.exe, errdsp.exe, prjdsp.exe
GUI:	N/A
Function:	When -settings+ is supplied, the current values for all configuration options are listed to standard output.

-sl, -slashwhite

Syntax:	-slashwhite {+ -}
Default:	-slashwhite-
Program:	qac.exe
GUI:	Compiler Personality>Extensions>Miscellaneous
Function:	With -slashwhite+, parsing ignores whitespace between \ and the end of a line.

Some source code uses continuation lines that have whitespace between the \ and the end of the line. This can happen when source files are transferred between different operating systems and results in level 9 errors (message 0907) from QA C, unless this option is enabled.

-spragma, -skippragma

Syntax:	-skippragma [<i>pragma_start</i>], [<i>pragma_end</i>]
Default:	no entry
Program:	qac.exe
GUI:	Analyser Personality>Pragma Blocks
Function:	Defines the behaviour of #pragma directives.

#pragma statements are preprocessor directives that are specific to a particular compiler. When QA C encounters a #pragma it will generate warning message 3116 unless you declare it.

You can declare a #pragma or a #pragma block by entering one or more #pragma names, separated by a comma:

Source Code	Begin	End	Action
#pragma ID	ID		#pragma ID is ignored.
#pragma x1 ... #pragma_x2	X1	X2	Ignores both #pragmas and all lines in between.
#pragma ABC1 ... #pragma ABC2 ... #pragma ABC3 ...	ABC*		Ignores all #pragmas that begin with ABC.

Note:
QA C also recognises two specific #pragma directives of its own, PRQA_MESSAGES_OFF and PRQA_MESSAGES_ON. These do not need to be declared. They are used for suppressing and re-enabling warning messages from within the code. Refer to Message Suppression in Chapter 3, Configuring QA C.

-sr, -strictrank

Syntax:	-strictrank {+ -}
Default:	-strictrank-
Program:	qac.exe
GUI:	Analyser Personality>Analysis Processing> Implicit Conversion Message Options
Function:	The -strictrank option affects the behaviour of implicit type conversion messages which report conversions from one integer type to a type of the same

	<p>signedness but lower rank. These messages are referred to in the standard message file as “narrowing” conversions.</p> <p>When the <code>strictrank</code> option is off (<code>-strictrank-</code>), these messages are not generated when converting to a type which is of lower rank but is actually an “equivalent” type.</p>
--	--

The term rank (integer conversion rank) is a term used in the ISO-C99 standard to describe the intrinsic ordering of integer types. In brief, no two integer types have the same rank even if they share the same representation. Type `long` (or `long long`) has the highest rank and type `char` has the lowest rank. The rank of an unsigned type is equal to the rank of the corresponding signed type.

Equivalent type in this context refers to the situation where two integer types are of different integer rank but share the same representation

-ss, -strictsignedness

Syntax:	<code>-strictsignedness {+ -}</code>
Default:	<code>-strictsignedness+</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Analysis Processing> Implicit Conversion Message Options
Function:	<p>With <code>-strictsignedness+</code>, “unsigned to larger signed” messages will always be generated, even if the conversion is to a wider type.</p> <p>On command line, the default setting (on) represents the prior QA C behaviour in being “strict” on unsigned to larger signed conversion messages.</p> <p>In the message personality, the messages under consideration here have been grouped under “Unsigned to Larger Signed”.</p> <p>Note: “Larger” in this context refers to “rank” rather than “implemented size”. Rank refers to the intrinsic</p>

	ordering of the basic types.
--	------------------------------

-st, -standard

Syntax:	<code>-standard {+ -}</code>
Default:	<code>-standard-</code>
Program:	<code>qac.exe, errdsp.exe, prjdsp.exe</code>
GUI:	Message Personality>Display> Display Highest Warning Level
Function:	<p>Displays the highest warning level of a message.</p> <p>When set to <code>-standard+</code>, this option causes annotated source output to display the highest level warning message in annotated source code.</p> <p>When set to <code>-standard-</code> and a detected warning occurs at more than one warning level, the lower level message is displayed.</p>

A typical compliance module will include some messages at Level 4 that are also present at Level 8 in the message file. It is usual to set as `-standard+` so that these messages are displayed as Level 8 warnings.

-sty, -style

Syntax:	<code>-style {k+r, exdented, indented}</code>
Default:	<code>-style exdented</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Style>Brace Style
Function:	Enforces various bracing styles.

The following are the three styles of bracing and indentation most commonly used in C programming:

K&R (Kernighan & Ritchie)

```
if ( ) {
    ...
}
```

Indented

```
if ( )
{
    ...
}
```

Exdented

```
if ( )
{
    ...
}
```

QA C will enforce consistency but will not enforce any particular depth of indentation, unless it is specified using the `-indentlevel` option.

-su, -suppresslvl

Syntax:	<code>-suppresslvl message_level</code>
Default:	<code>no entry</code>
Program:	<code>errdsp.exe, prjdsp.exe, viewer.exe, errsum.exe</code>
GUI:	Message Personality>Warning Messages
Function:	Suppresses all warning messages from a specified message level. To suppress a number of levels, select a range with a dash, for example, <code>-su 4-7</code> , or specify a selection as a comma-seperated list, for example, <code>-su 1, 2, 5</code> .

-summ, -summary

Syntax:	<code>-summary {+ -}</code>
Default:	<code>-summary-</code>
Program:	<code>errdsp.exe</code>
GUI:	Message Personality>Display>Create Analysis Summary

Function:	<p>With <code>-summary+</code>, a summary is produced is displayed instead of annotated source code, which shows the number of messages detected at each level.</p> <p>If the <code>-metrics</code> option is set, the summary will list each warning message, by level, and the number of occurrences of the warning message.</p>
------------------	--

-t, -tab, -tabstop

Syntax:	<code>-tabstop columns</code>
Default:	<code>-tabstop 8</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Style>Source Code Tab Spacing
Function:	<p>Specifies the number of spaces represented by a tab character. Tab spacing is not related to depth of indentation. QA C will warn about inconsistent indentation but will not enforce any particular depth. If this option is set to zero, QA C will issue message 2210 whenever it encounters a tab character in the source code.</p>

-text, -textonly

Syntax:	<code>-textonly {+ -}</code>
Default:	<code>-textonly-</code>
Program:	<code>errdsp.exe, prjdsp.exe</code>
GUI:	N/A
Function:	<p>With <code>-textonly+</code>, the level number and message number that normally prefix the message text are both suppressed.</p> <p>If the <code>-format</code> option is specified, the <code>-textonly</code> option will have no effect. All message formatting will be determined by <code>-format</code>.</p>

-thresh, -threshold

Syntax:	<code>-threshold metric{< <= = > >=}value[:msg_no]</code>
Default:	<code>no entry</code>
Program:	<code>qac.exe</code>
GUI:	Analyser Personality>Metrics>Metric Thresholds
Function:	When specified, message <i>msg_no</i> is output whenever the threshold condition (from the set {< <= = > >=}) for specified <i>metric</i> is exceeded. The metric is specified by the name that appears in the metrics records of the <code>.met</code> file. See Metrics Records of Appendix F: Metric Output File. There are no default values, and if <i>:msg_no</i> is omitted, the default message 4800 is used.

-total

Syntax:	<code>-total {+ -}</code>
Default:	<code>-total-</code>
Program:	<code>errdsp.exe</code>
GUI:	N/A
Function:	When running <code>errdsp.exe</code> on more than one file, and using the <code>-summary</code> or <code>-tablesummary</code> options, <code>-total+</code> will generate an additional record containing the accumulated totals for all files.

-tsumm, -tablesummary

Syntax:	<code>-tablesummary {+ -}</code>
Default:	<code>-tablesummary-</code>
Program:	<code>errdsp.exe</code>
GUI:	N/A

Function:	<p>With <code>-tablesummary-</code>, a summary is produced that shows the number of messages detected at each level. This format is displayed instead of annotated source code. This option is identical to <code>-summary</code> except that the levels are separated by tabs instead of colons.</p> <p>If <code>-metrics+</code> is set, the summary will list each warning message and its number of occurrences, organised by message level.</p>
------------------	--

-u, -unsignedchar

Syntax:	<code>-unsignedchar {+ -}</code>
Default:	<code>-unsignedchar-</code>
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Data Types>Char Interpretation
Function:	<p>With <code>-unsignedchar+</code>, the <code>char</code> datatype is interpreted as unsigned <code>char</code>.</p> <p>With <code>-unsignedchar-</code>, it is interpreted as signed <code>char</code>.</p>

The ISO standard states that a compiler may implement `char` as a signed or unsigned datatype. It is safer not to assume any particular implementation.

You can test whether your source relies on plain `char` being treated in a particular manner by running QA C twice, once, with `char` declared as signed and again with plain `char` declared as unsigned. Messages that differ between the two runs may indicate reliance on the treatment of plain `char`.

-up, -usrpath

Syntax:	<code>-usrpath [path]</code>
Default:	<code>%QACBIN%</code>
Program:	<code>errdsp.exe, prjdsp.exe, viewer.exe</code>

GUI:	Message Personality>Advanced Settings> User Message File
Function:	<p>Specifies the path of the user message file.</p> <p>In the GUI, this option will only be set if you specify a user message file in a directory other than the <code>bin</code> directory.</p> <p>If this option is not specified, viewing tools use the path defined by <code>QACBIN</code>.</p> <p>QA C will also look for message help files in the specified path before looking in the path defined by <code>QACHELPFILES</code>. In both cases, <code>\messages</code> sub-directory is added for file searching.</p>

-usr, -usrfile

Syntax:	<code>-usrfile extension</code>
Default:	<code>no entry</code>
Program:	<code>errdsp.exe</code> , <code>prjdsp.exe</code> , <code>viewer.exe</code>
GUI:	Message Personality>Advanced Settings> User Message File
Function:	Specifies the file extension of a user message file.

QA C has a library of messages that are contained in the `qac.msg` file. You can apply a user message file to add new messages or to change the text of existing messages. The name of the user message file will be `qac.usr.extension`, and it will be located in a directory determined by `-usrpath` or `QACBIN`.

In the GUI, clicking Refresh Message View will display, in the list of messages, the warning messages from your user message file along with those from the message file.

-ver, -version

Syntax:	<code>-version {+ -}</code>
----------------	-----------------------------

Default:	-version-
Program:	qac.exe, errdsp.exe, prjdsp.exe, viewer.exe
GUI:	N/A
Function:	With -version+, the version number for the executable is displayed

-via

Syntax:	-via <i>file</i>
Default:	no entry
Program:	qac.exe, errdsp.exe, prjdsp.exe, viewer.exe
GUI:	N/A
Function:	Specifies a file containing configuration options.

-wc, -warncall

Syntax:	-warncall [<i>function</i> [= <i>msg_no</i>]]
Default:	no entry
Program:	qac.exe
GUI:	Analyser Personality>Warning Calls
Function:	Displays warning message <i>msg_no</i> if QA C encounters a call to <i>function</i> .

If no message number is declared, the following message is displayed:

```
Msg(4:2010) 'func_name' must not be called.
```

-xc, -xcase

Syntax:	-xcase {+ -}
Default:	-xcase-
Program:	qac.exe

GUI:	Compiler Personality>Identifiers> Ignore External Identifier Case
Function:	With <code>-xcase+</code> , analysis implements the ISO standard that specifies that alphabetic case should not be treated as significant in external identifiers. See <code>-xnamelength</code> for implications of external identifier significance.

-xn, -xnamelength

Syntax:	<code>-xnamelength value</code>
Default:	<code>-xnamelength 6</code>
Program:	<code>qac.exe</code>
GUI:	Compiler Personality>Identifiers> External Identifier Significant Length
Function:	Determines the number of significant characters in an external identifier.

The ISO C Standard states that an implementation may restrict the significance of an external name to only 6 characters and may ignore alphabetical case. This is quite severe and many compilers and linkers relax this restriction.

QA C will generate message number 777 for each external identifier it encounters that is not distinct within the number of characters you have specified.

APPENDIX C

The Components of Function Structure

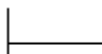
Function structure diagrams show the control flow structures within source code. Decisions (`if`, `switch`, and the condition part of a loop) are displayed as forks in the structure. The corresponding join, further to the right, indicates where the paths rejoin. Backward arcs, caused by loops, are shown as dotted lines.

The function structure diagrams show the following code structures:

- straight code,
- `if`,
- `if else`,
- `switch`,
- `while` loop,
- `for` loop,
- nested structures,
- `break` in a loop,
- `return` in a loop,
- `continue` in a loop,
- unreachable code.

Each component progresses from the left to the right as the control flow would progress through the source code.

Straight code



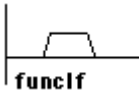
funcStraight

```
static int code = 0;
```

```
void funcStraight (void)
{
    code = 1;
}
```

The x-axis denotes that there is only one path through this particular function.

If

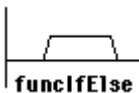


```
static int code = 0;

void funcIf (void)
{
    if (code > 0)
    {
        code = 1;
    }
}
```

This function has two paths. The first path is through the `if` statement and is shown by the raised line. The second path is where the `if` condition is false and is represented by the x-axis.

If-Else



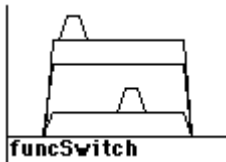
```
static int code = 0;

void funcIfElse (void)
{
    if (code > 0)
    {
        code = 3;
    }
    else
    {
        code = 4;
    }
}
```

This function has two execution paths. The first path is the `if` sub-statement represented by the raised line. The second path is the `else` sub-statement represented by the x-axis.

Note that the body of this structure is longer than the body of the `if` statement. If the two arms of the `if-else` were not straight line code, the body of the `if` branch would appear at the left hand end of the raised line and the body of the `else` branch would appear to the right of the lower line.

Switch



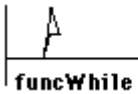
```
static int code = 0;

void funcSwitch (void)
{
    switch (code)
    {
        case 1:
            if (code == 1)
            {
                /* block of code */
            }
            break;
        case 2:
            break;
        case 3:
            if (code == 3)
            {
                /* block of code */
            }
            break;
        default:
            break;
    }
}
```

In the `switch` statement, the x-axis represents the default action, and each case statement is shown by a raised line. The two briefly raised lines represent the `if` statements within case 1 and case 3.

The diagram shows how the `if` statements are staggered. The `if` of case 1 is shown on the left and the `if` of case 3 is shown on the right.

While loop

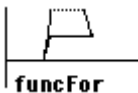


```
static int code = 0;

void funcWhile (void)
{
    while (code > 0)
    {
        --code;
    }
}
```

In the `while` loop, the x-axis represents the path straight through the function as though the `while` loop had not been executed. The solid raised line shows the path of the `while` body. The dotted line shows the loop to the beginning of the `while` statement.

For loop



```
static int code = 0;
void doSomethingWith(int);

void funcFor (void)
{
    for (int i = 0; i > code; ++i)
    {
        doSomethingWith(i);
    }
}
```

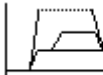
The `for` loop is similar to the `while` loop as `for` loops can be rewritten as `while` loops. For example, `funcFor` in the above example could be written as:

```

void funcFor (void)
{
    int i=0;
    while (i > code)
    {
        /* body */
        ++i;
    }
}

```

Nested Structures



funcWhileIfElse

```

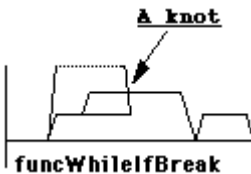
static int code = 0;

void funcWhileIfElse (void)
{
    while (code > 0)
    {
        if (code == 1)
        {
            code = 0;
        }
        else
        {
            --code;
        }
    }
}

```

This is an if-else contained within a while loop. The first solid raised line represents the while loop while the inner raised solid line represents the if-else loop. Other function structure components can be similarly nested.

Break in a loop



funcWhileIfBreak

```

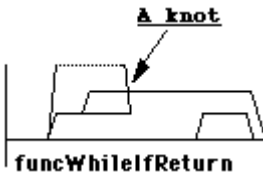
static int code = 0;

void funcWhileIfBreak (void)
{
    while (code > 0)
    {
        if (code == 3)
        {
            break;
        }
        --code;
    }
    if (code == 0)
    {
        code++;
    }
}

```

The `break` jumps to the end of the `while` statement and causes a knot. A knot is where the control flow crosses the boundary of another statement block and indicates unstructured code. In this case, `break` jumps to the end of the `while` statement and the next part of the program, the `if` statement, is executed.

Return in a loop



```

static int code = 0;

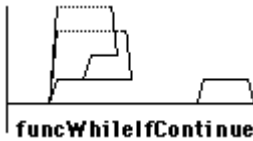
void funcWhileIfReturn (void)
{
    while (code > 0)
    {
        if (code == 3)
        {
            return;
        }
        --code;
    }
    if (code == 0)
    {
        code++;
    }
}

```

}

The `return` statement causes the program to jump to the end of the function. This jump breaks the control flow and causes a knot in the code.

Continue in a loop



```
static int code = 0;

void funcWhileIfContinue (void)
{
    while (code > 0)
    {
        if (code == 3)
        {
            continue;
        }
        --code;
    }
    if (code == 0)
    {
        code++;
    }
}
```

The `continue` statement causes the program to jump back to the beginning of the `while` loop and does not cause a knot.

Unreachable code



```
static void funcUnreach (int i)
{
    if (i)
    {
        i = 1;
    }
}
```

```
goto Bad;

if (i)
{
    i = 2;
}

Bad:
if (i)
{
    i = 3;
}
return;
}
```

The raised section represents code that is unreachable.

The structure and approximate position of the unreachable code is shown. It occurs after the first `if` condition and before the last `if` condition. Its position above the main structure shows that the control flow misses the middle `if` condition.

APPENDIX D

The Calculation of Metrics

QA C calculates metrics in three groups. Function metrics are generated for all functions with a full definition. File metrics are generated for each file analysed. Project metrics are calculated once per complete project, and are generated from CMA analysis.

Function-Based Metrics

These metrics are all calculated for the source code of an individual function. Some metrics are used in the definition of others but there is no obvious sequence and so they are described here in alphabetical order for ease of reference. The complete list is as follows:

STAKI	Akiyama's Criterion
STAV1	Average Size of Function Statements (variant 1)
STAV2	Average Size of Function Statements (variant 2)
STAV3	Average Size of Function Statements (variant 3)
STBAK	Number of Backward Jumps
STCAL	Number of Distinct Function Calls
STCYC	Cyclomatic Complexity
STELF	Number of Dangling Else-If's
STFN1	Number of Function Operator Occurrences
STFN2	Number of Function Operand Occurrences
STGTO	Number of Goto's
STKDN	Knot Density
STKNT	Knot Count
STLCT	Number of Local Variables Declared
STLIN	Number of Maintainable Code Lines
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	Maximum Nesting of Control Structures
STPAR	Number of Function Parameters

STPBG	Path-Based Residual Bug Estimate
STPDN	Path Density
STPTH	Estimated Static Path Count
STRET	Number of Function Return Points
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	Number of Unused and Non-Reused Variables
STXLN	Number of Executable Lines

STAKI Akiyama's Criterion

This metric is the sum of the cyclomatic complexity (STCYC) and the number of function calls (STSUB). Although this is not an independent metric, it is included on account of its use in documented case histories. See Akiyama² and Shooman³ for more details. The metric is calculated as:

$$\text{STAKI} = \text{STCYC} + \text{STSUB}$$

STAVx Average Size of Function Statements

These metrics (STAV1, STAV2, STAV3) measures the average number of operands and operators per statement in the body of the function. It is calculated as follows:

$$\text{STAV}_x = (\text{N1} + \text{N2}) / \text{number of statements in the function}$$

where:

N1 is Halstead's number of operator occurrences,

N2 is Halstead's number of operand occurrences.

² Akiyama, F. (1971) *An Example of Software System Debugging*, Proc. IFIP Congress 1971, Ljubljana, Yugoslavia, American Federation of information Processing Societies, Montvale, New Jersey.

³ Shooman, M.L. (1983) *Software Engineering*, McGraw-Hill, Singapore.

The STAV_x metrics are computed using STST1, STST2 and STST3 to represent the number of statements in a function, hence there are three variants: STAV1, STAV2 and STAV3 relating to the respective statement count metrics.

This metric is used to detect components with long statements. Statements comprising a large number of textual elements (operators and operands) require more effort by the reader in order to understand them. This metric is a good indicator of the program's readability.

Metric values are computed as follows:

$$\text{STAV1} = (\text{STFN1} + \text{STFN2}) / \text{STST1}$$
$$\text{STAV2} = (\text{STFN1} + \text{STFN2}) / \text{STST2}$$
$$\text{STAV3} = (\text{STFN1} + \text{STFN2}) / \text{STST3}$$

STBAK Number of Backward Jumps

Jumps are never recommended and backward jumps are particularly undesirable. If possible, the code should be redesigned to use structured control constructs such as `while` or `for` instead of `goto`.

```
main()
{
    Backward:
    switch(n)
    {
        case 0: printf(stdout, "zero\n");
                break;
        case 1: printf(stdout, "one\n");
                goto Backward;           /* 1 */
                break;
        case 2: printf(stdout, "two\n");
                break;
        default: printf(stdout, "many\n");
                 break;
    }
}
```

The above code sample has a STBAK value of 1.

STCAL Number of Distinct Function Calls

This metric counts the number of function calls in a function.

It differs from the metric STSUB, in that only distinct functions are counted (multiple instances of calls to a particular function are counted as one call), and also that functions called via pointers are not counted.

STCYC Cyclomatic Complexity

Cyclomatic complexity is calculated as the number of decisions plus 1.

High cyclomatic complexity indicates inadequate modularization or too much logic in one function. Software metric research has indicated that functions with a cyclomatic complexity greater than 10 tend to have problems related to their complexity.

McCabe⁴ gives an essential discussion of this issue as well as introducing the metric.

Example 1:

```
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;
    }
}
```

The above code sample has a cyclomatic complexity of 3 as there are two decisions made by the function. Note that *correctly indented* code does not always reflect the *nesting structure* of the code. In particular the use of the construct 'else if' always increases the level of nesting but is conventionally written without additional indentation and so the nesting is not visually apparent.

⁴ McCabe, T. J. (1976) *A Complexity Measure*, IEEE Transactions on Software Engineering, SE-2, pp. 308-320.

Example 2:

```
void how_many(int n)
{
    switch (n)
    {
        case 0: printf('zero');          /* 1 */
                break;
        case 1: printf('one');           /* 2 */
                break;
        case 2: printf('two');           /* 3 */
                break;
        default: printf('many');
                break;
    }
}
```

The above code sample has a cyclomatic complexity of 4, as a switch statement is equivalent to a series of decisions.

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.

STCYC is one of the three standard metrics used by QA C for demographic analysis.

STELF Number of Dangling Else-Ifs

This is the number of `if-else-if` constructs that do not end in an `else` clause. This metric is calculated by counting all `if` statements that do not have a corresponding `else` and QA C issues a warning 2004 for each instance. This statistic provides a quick reference allowing monitoring of these warnings. The code sample below has an STELF value of 1.

```
int divide(int x, int y)
{
    if (y != 0)
    {
        return x/y;
    }
    else if (x == 0)
    {
        return 1;
    }
}
```

STFN1 Number of Function Operator Occurrences

This metric is Halstead's operator count on a function basis (N1). STFN1 is related to STOPT and STM21: all of these metrics count 'operators', the difference is summarised as:

- STFN1 Counts ALL operators in the function body
- STM21 Counts ALL operators in the file
- STOPT Counts DISTINCT operators in the file

See STOPT for the definition of an operator.

STFN2 Number of Function Operand Occurrences

This metric is Halstead's operand count on a function basis (N2). STFN2 is related to STOPN and STM20: all of these metrics count 'operands', the difference is summarised as:

- STFN2 Counts ALL operands in the function body
- STM20 Counts ALL operands in the file
- STOPN Counts DISTINCT operands in the file

See STOPN for the definition of an operand.

STGTO Number of Gotos

Some occurrences of `goto` simplify error handling. However, they should be avoided whenever possible. The Plum Hall Guidelines say that `goto` should not be used.

STKDN Knot Density

This is the number of knots per executable line of code. The metric is calculated as:

$$\text{STKDN} = \text{STKNT} / \text{STXLN}$$

The value is computed as zero when STXLN is zero. QA C issues warnings if the knot density exceeds certain values.

STKNT Knot Count

This is the number of knots in a function. A knot is a crossing of control structures, caused by an explicit jump out of a control structure either by `break`, `continue`, `goto`, or `return`. STKNT is undefined for functions with unreachable code.

This metric measures knots not by counting control structure crossings but by counting the following knot keywords:

- `goto` statements,
- `continue` statements within `loop` statements,
- `break` statements within `loop` or `switch` statements, except those at top `switch` level,
- all `return` statements except those at top function level.

The function below has an STKNT value of 1.

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

STLCT Number of Local Variables Declared

This is the number of local variables declared in a function of storage class `auto`, `register`, or `static`. These are variables that have no linkage. The function below has an STLCT value of 2.

```
int other_result;
extern int result;

int test()
{
    int x;                                     /* 1 */
}
```

```

    int y;                                /* 2 */

    return (x + y + other_result);
}

```

STLIN Number of Maintainable Code Lines

This is the total number of lines, including blank and comment lines, in a function definition between (but excluding) the opening and closing brace of the function body. It is computed on raw code. STLIN is undefined for functions which have `#include`'d code or macros which include braces in their definition.

The function below has an STLIN value of 5.

```

int fn()
{
    int x;                                /* 1 */
    int y;                                /* 2 */
                                        /* 3 */
    return (x + y);                       /* 4 */
    /* Comment Here */                   /* 5 */
}

```

Long functions are difficult to read, as they do not fit on one screen or one listing page, and thus an upper limit of 200 is recommended.

STLOP Number of Logical Operators

This is the total number of logical operators (`&&`, `||`) in the conditions of `do-while`, `for`, `if`, `switch`, or `while` statements in a function. The example function below has a STLOP value of 2.

```

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

```

```
}
```

STM07 Essential Cyclomatic Complexity

The essential cyclomatic complexity is obtained in the same way as the cyclomatic complexity but is based on a 'reduced' control flow graph. The purpose of reducing a graph is to check that the component complies with the rules of structured programming.

A control graph that can be reduced to a graph whose cyclomatic complexity is 1 is said to be structured. Otherwise reduction will show elements of the control graph which do not comply with the rules of structured programming.

The principle of control graph reduction is to simplify the most deeply nested control subgraphs into a single reduced subgraph. A subgraph is a sequence of nodes on the control flow graph which has only one entry and exit point. Four cases are identified by McCabe⁵ which result in an unstructured control graph. These are:

- a branch into a decision structure,
- a branch from inside a decision structure,
- a branch into a loop structure,
- a branch from inside a loop structure.

However, if a component possesses multiple entry or exit points then it can not be reduced. The use of multiple entry and exit points breaks the most fundamental rule of structured programming.

The example below has a STM07 value of 4.

```
void g( int n, int pos, int force ) /* STM07 = 4
                                   Cannot reduce control graph to
                                   Cyclomatic Complexity of 1 */
{
    int nlines = 0;
    while ( --n >= 0 )
    {
        pos = back_line( pos );
    }
}
```

⁵ McCabe, T. J. (1976) *A Complexity Measure*, IEEE Transactions on Software Engineering, SE-2, pp. 308-320.

```
        if ( pos == 0 )
        {
            if ( ! force )
            {
                break;
            }
            ++nlines;
        }
    }
}
```

STM19 Number of Exit Points

This metric is a measure of the number of exit points in a software component and is calculated by counting the number of return statements. A function that has no return statements will have an STM19 value of zero even though it will exit when falling through the last statement. This is regardless of whether the function is declared to have a return value or not (i.e. returns void). Exits by special function calls such as `exit()` or `abort()` are ignored by this metric.

The example below has an STM19 value of 3.

```
void f( int a )
{
    return;          /* 1 */
    if ( a ) return; /* 2 */
    a++;
    return;          /* 3 */
}
```

STM29 Number of Functions Calling this Function

This metric is defined as the number of functions calling the designated function. The number of calls to a function is an indicator of criticality. The more a function is called, the more critical it is and, therefore, the more reliable it should be.

STMCC Myer's Interval

This is an extension to the cyclomatic complexity metric. It is expressed as a pair of numbers, conventionally separated by a colon. 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 QA C STLOP metric which is a measure of the number of logical operators ($\&\&$, $||$) in the conditional expressions of a function. A large value of L indicates that there are a lot of compound decisions, which makes the code more difficult to understand. A Myer's interval of 10 is considered very high.

The example below has a STMCC value of 3:4 because the cyclomatic complexity is 3 and there is one connective ($\&\&$) used in the conditions.

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

Notes:

In the calculation of STMCC, the ternary operator ($? :$) is ignored.

When exporting metric values or displaying in the metrics browser, rather than attempt to display a value pair the value of L is chosen for STMCC.

STMIF Maximum Nesting of Control Structures

This metric is a measure of the maximum control flow nesting in your source code.

You can reduce the value of this metric by turning your nesting into separate functions. This will improve the readability of the code by reducing both the nesting and the average cyclomatic complexity per function.

The code example below has an STMIF value of 3.

```
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;
    }
}
```

STMIF is incremented in switch, do, while, if and for statements. Note that the nesting level of code is not always visually apparent by looking at the indentation. In particular, an 'else if' construct increases the level of nesting in the control flow structure but is conventionally written without additional indentation.

STMIF is one of the standard metrics used by QA C for demographic analysis.

STPAR Number of Function Parameters

This metric counts the number of declared parameters in the function argument list. Note that ellipsis parameters are ignored.

STPBG Path-Based Residual Bug Estimate

Hopkins, in Hatton & Hopkins⁶ investigated software with a known audit history and observed a correlation between Static Path Count (STPTH) and the number of bugs that had been found. This relationship is expressed as STPBG.

$$\text{STPBG} = \log_{10} (\text{STPTH})$$

STPDN Path Density

This is a measure of the number of paths relative to the number of executable lines of code.

$$\text{STPDN} = \text{STPTH} / \text{STXLN}$$

STPDN is computed as zero when STXLN is zero.

STPTH Estimated Static Path Count

This is similar to Nejme⁷'s NPAT^H statistic and gives an upper bound on the number of possible paths in the control flow of a function. It is the number of non-cyclic execution paths in a function.

The NPAT^H value for a sequence of statements at the same nesting level is the product of the NPAT^H values for each statement and for the nested structures. NPAT^H is the product of:

- NPAT^H(sequence of non control statements) = 1
- NPAT^H(if) = NPAT^H(body of then) + NPAT^H(body of else)
- NPAT^H(while) = NPAT^H(body of while) + 1
- NPAT^H(do while) = NPAT^H(body of while) + 1

⁶ Hatton, L., Hopkins, T.R., (1989) *Experiences With Flint, a Software Metrication Tool for Fortran 77*, Symposium on Software Tools, Napier Polytechnic, Edinburgh, Scotland.

⁷ Nejme⁷, B.A. (1988), *NPAT^H: A Measure of Execution Path Complexity and its Applications*, Comm ACM, 31, (2), p. 188-200.

- $\text{NPATH}(\text{for}) = \text{NPATH}(\text{body of for}) + 1$
- $\text{NPATH}(\text{switch}) = \text{Sum}(\text{NPATH}(\text{body of case } 1) \dots \text{NPATH}(\text{body of case } n))$

Note:

else and default are counted whether they are present or not.

In switch statements, multiple case options on the same branch of the switch statement body are counted once for each independent branch only. For example:

```
switch( n )
{
    case 0 : break;      /* NPATH of this branch is 1 */
    case 1 :
    case 2 : break;      /* NPATH for case 1 & case 2 */
                        /* combined is 1 */
    default: break;      /* NPATH for this default is 1 */
}
```

NPATH cannot be computed if there are goto statements in the function.

The following code example has a static path count of 26.

```
int n;
if ( n )
{
} /* block 1, paths 1 */

else if ( n )
{
    if ( n )
    {
    } /* block 2, paths 1 */

    else
    {
    } /* block 3, paths 1 */

    /* block 4, paths block2+block3 = 2 */

    switch ( n )
    {
    case 1 : break;
    case 2 : break;
    case 3 : break;
    case 4 : break;
```

```

        default: break;
        } /* block 5, paths = 5 */

    } /* block 6, paths block4*block5 = 10 */

else
{
    if ( n )
    {
        } /* block 7, paths 1 */

        else
        {
            } /* block 8, paths 1 */
        } /* block 9, paths block7+block8 = 2 */
    } /* block 10, paths block1+block6+block9 = 13 */

    if ( n )
    {
        } /* block 11, paths 1 */

    else
    {
        } /* block 12, paths 1 */
    } /* block 13, paths block11+block12 = 2 */

    /* outer block, paths block10*block13 = 26 */

```

Each condition is treated as disjoint. In other words, no conclusions are drawn about a condition that is tested more than once.

The true path count through a function usually obeys the inequality:
Cyclomatic complexity \leq true path count \leq static path count

Static path count is one of the three standard metrics used by QA C for demographic analysis.

STRET Number of Function Return Points

STRET is the count of the reachable return statements in the function, plus one if there exists a reachable implicit return at the `}` that terminates the function.

The following example shows an implicit return:

```

void foo( int x, int y )
{

```

```

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

```

Structured Programming requires that every function have exactly one entry and one exit. This is indicated by a STRET value of 1. STRET is useful when the programmer wants to concentrate on functions that do not follow the Structured Programming paradigm, for example those with switch statements with returns in many or every branch.

STSTx Number of Statements in Function

These metrics count the number of statements in the function body. There are 3 variants on the metric:

STST1 is the base definition and counts all statements tabulated below.

STST2 is STST1 except block, empty statements and labels are not counted.

STST3 is STST2 except declarations are not counted.

The following chart shows the statements counted by the STST metrics:

Statement Kind	STST1	STST2	STST3
block	yes	ignore	ignore
simple statement followed by ;	yes	yes	yes
empty statement	yes	ignore	ignore
declaration statement	yes	yes	ignore
label	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

The following example shows statements counted by STST1, which for this function yields a value of 10:

```

void stst( void )
{

```

```

    int i;                /* 1 */
    label_1:              /* 2 */
    label_2:              /* 3 */
    switch ( 1 )          /* 4 */
    {                     /* 5 */
        case 0:           /* 6 */
        case 1:           /* 7 */
        case 2:           /* 8 */
        default:          /* 9 */
            break;        /* 10 */
    }
}

```

This metric indicates the maintainability of the function. Number of statements also correlates with most of the metrics defined by Halstead. The greater the number of statements contained in a function, the greater the number of operands and operators, and hence the greater the effort required to understand the function. Functions with high statement counts should be limited. Restructuring into smaller sub-functions is often appropriate.

STSUB Number of Function Calls

The number of function calls within a function. Functions with a large number of function calls are more difficult to understand because their functionality is spread across several components. Note that the calculation of STSUB is based on the number of function calls and not the number of distinct functions that are called.

A large STSUB value may be an indication of poor design, for example a calling tree that spreads too rapidly. See Brandl (1990)⁸ for a discussion of design complexity and how it is highlighted by the shape of the calling tree.

The following code example has an STSUB value of 4.

```

extern dothis(int);
extern dothat(int);
extern dotheother(int);

void test()
{
    int a, b;

```

⁸ Brandl, D.L. (1990), *Quality Measures in Design*, ACM Sigsoft Software Engineering Notes, vol 15, 1.

```
a = 1;
b = 0;

if (a == 1)
{
    dothis(a);                      /* 1 */
}
else
{
    dothat(a);                      /* 2 */
}

if (b == 1)
{
    dothis(b);                      /* 3 */
}
else
{
    dotheother(b);                  /* 4 */
}
}
```

STUNR Number of Unreachable Statements

This metric is the count of all statements within the function body that are guaranteed never to be executed. STUNR uses the same method for identifying statements as metric STST1. Hence STUNR counts the following as statements if unreachable.

Statement Kind Counted

- block
- simple statement followed by ;
- empty statement
- declaration statement
- label
- break
- continue
- do
- for
- goto
- if
- return
- switch
- while

Example yielding STUNR = 4

```
void stunr( unsigned i )
{
    if ( i >= 0 )
    {
        if ( i >= 0 )
        {
            while ( i >= 0 ) return;
        }
        else
            /* unreachable */
            {
                while ( i >= 0 ) return;
            }
    }
    return; /* unreachable */
}
```

STUNV Unused and Non-Reused Variables

An unused variable is one that is defined but never referenced. A non-reused variable is a variable that has a value by assignment but is never subsequently used.

Such variables are generally clutter and are often evidence of “software ageing”, the effect of a number of programmers making changes. The code example below would have an STUNV value of 2.

```
int other_result;
extern int result;

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

    z = 1; /* Non-Reused */

    return (result + other_result);
}
```

STXLN Number of Executable Lines

This is a count of lines in a function body that have code tokens. Comments, braces, and all tokens of declarations are not treated as code tokens. The function below has an STXLN value of 9.


```
void fn( int n )
{
    int x;
    int y;

    if ( x )                                /* 1 */
    {
        x++;                                /* 2 */
        for (;;)                            /* 3 */
            /* Ignore comments */
            /* Ignore braces */
            {
                switch ( n )                /* 4 */
                {
                    case 1 : break;         /* 5 */
                    case 2 :                /* 6 */
                    case 3 :                /* 7 */
                        break               /* 8 */
                    ;                       /* 9 */
                }
            }
    }
}
```

This metric is used in the computation of the STKDN and STPDN metrics.

File-Based Metrics

These metrics are all calculated for the source code in a complete translation unit. They are described here in alphabetical order for ease of reference although because of their derivation from each other, they would not be calculated in this sequence. The complete list is as follows:

STBME	COCOMO Embedded Programmer Months
STBMO	COCOMO Organic Programmer Months
STBMS	COCOMO Semi-detached Programmer Months
STBUG	Residual Bugs (token-based estimate)
STCDN	Comment to Code Ratio
STDEV	Estimated Development Time
STDIF	Program Difficulty
STECT	Number of External Variables
STEFF	Program Effort
STFCO	Estimated Function Coupling
STFNC	Number of Function Definitions
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
STMOB	Code Mobility
STOPN	Halstead Distinct Operands
STOPT	Halstead Distinct Operators
STPRT	Estimated Porting Time
STSCT	Number of Static Variables
STSHN	Shannon Information Content
STTDE	COCOMO Embedded Total Months
STTDO	COCOMO Organic Total Months
STTDS	COCOMO Semi-detached Total Months
STTLN	Total Preprocessed Source Lines
STTOT	Total Number of Tokens
STTPP	Total Unpreprocessed Source Lines
STVAR	Number of Identifiers
STVOL	Program Volume
STZIP	Zipf Prediction of STTOT

STBMO	COCOMO Organic Programmer Months
STBMS	COCOMO Semi-detached Programmer Mths
STBME	COCOMO Embedded Programmer Months

The COCOMO metrics are produced for each source code file. You can display an accurate estimate of development costs for a whole project by choosing the COCOMO Cost Model option from the Reports menu. See Chapter 6: Reports for a discussion of the COCOMO cost model and an explanation of the three programming modes, Organic, Semi-detached, and Embedded.

These metrics are an estimate of the number of programmer months required to create the source code in the respective environments.

$$\text{STBME} = 3.6 * (\text{STTPP} / 1000) 1.20$$

$$\text{STBMO} = 2.4 * (\text{STTPP} / 1000) 1.05$$

$$\text{STBMS} = 3.0 * (\text{STTPP} / 1000) 1.12$$

STBUG Residual Bugs (token-based estimate)

$$\text{STBUG} = 0.001 * \text{STEFF}^{2/3}$$

This is an estimate of the number of bugs in the file, based on the number of estimated tokens. Its value would normally be lower than the sum of the function based STPBG values. For a more detailed discussion of software bug estimates, see Hatton and Hopkins⁹.

STCDN Comment to Code Ratio

This metric is defined to be the number of visible characters in comments divided by the number of visible characters outside comments. Comment delimiters are ignored. Whitespace characters in strings are treated as visible characters.

A large metric value may indicate that there are too many comments and can make a module difficult to read. A small value may indicate

⁹ Hatton, L., Hopkins, T.R., (1989) *Experiences With Flint, a Software Metrication Tool for Fortran 77*, Symposium on Software Tools, Napier Polytechnic, Edinburgh, Scotland.

that there are not enough comments.

The code example below has 28 visible characters in comments and 33 visible characters in the code. The resulting metric value would be 0.85.

```
int test()
/* This is a test */
{
    int x;
    int y;
/* This is another test */
    return (x + y);
}
```

The calculation of this metric is affected by the way in which QA C counts comments. QA C will count characters in three ways:

- all comments, (a),
- all comments except for those from headers, (n),
- inline or internal comments (i). These are comments within functions and comments that annotate a line of code (comments that are on the same line as code at file scope).

You can determine how QA C counts comments in Comment Count on the Metrics tab of the Analyser Personality or by setting the `-co` option on the command line.

STDEV Estimated Development Time

This is an estimate of the number of programmer days required to develop the source file. Unlike COCOMO statistics, which are based solely on the number of lines of code, this estimate is derived from the file's difficulty factor. It is a more accurate measure of the development time, especially after the scaling factor has been adjusted for a particular software environment.

$STDEV = STEFF / dev_scaling$

where `dev_scaling` is a scaling factor defined in `qac.cfg`. The default is 6000.

STDIF Program Difficulty

This is a measure of the difficulty of a translation unit. An average C program has a difficulty of around 12. Anything significantly above this has a rich vocabulary and is potentially difficult to understand.

$$\text{STDIF} = \text{STVOL} / ((2 + \text{STVAR}) * \log_2 (2 + \text{STVAR}))$$

STECT Number of external variables

This metric measures the number of data objects (N.B. not including functions) declared with external linkage. It is an indication of the amount of global data being passed between modules. It is always desirable to reduce dependence on global data to a minimum.

```
extern int result;
int other_result;

main()
{
    result = 20 + 30;
    other_result = result * 2;
}
```

The above code sample has an STECT value of 2.

STEFF Program Effort

This metric is a measure of the programmer effort involved in the production of a translation unit. It is used to produce a development time estimate.

$$\text{STEFF} = \text{STVOL} * \text{STDIF}$$

STFNC Number of Function Definitions

This metric is a count of the number of function definitions in the file.

STFCO Estimated Function Coupling

See Brandl¹⁰. Since the actual value of Brandl's metric requires a full, well-structured calling tree, STFCO can only be an estimate. A high figure indicates a large change of complexity between levels of the calling tree. The metric is computed as follows from STFNC and the STSUB values of the component functions in the translation unit:

$$\text{STFCO} = \sum(\text{STSUB}) - \text{STFNC} + 1$$

The code example below has an STFCO value of 1 ($2 - 2 + 1$).

```

BOOL isActive(CHANNEL c);

BOOL okToRead(TEXTCHANNEL c)
{
    return !isActive(c);
}

BOOL okToPrint(PRINTCHANNEL c)
{
    return !isActive(c);
}

```

STHAL Halstead Prediction of STTOT

This metric and also STZIP are predictions derived from the vocabulary analysis metrics STOPN and STOPT of what the value of STTOT should be. If they differ from STTOT by more than a factor of 2, it is an indication of an unusual vocabulary. This usually means that either the source code contains sections of rather repetitive code or it has an unusually rich vocabulary. The two metrics are computed as follows:

$$\begin{aligned} \text{STZIP} &= (\text{STOPN} + \text{STOPT}) * (0.5772 + \ln(\text{STOPN} + \text{STOPT})) \\ \text{STHAL} &= \text{STOPT} * \log_2(\text{STOPT}) + \text{STOPN} * \log_2(\text{STOPN}) \end{aligned}$$

QA C checks the average of STZIP and STHAL against STTOT and issues a warning for unusual vocabularies.

¹⁰ Brandl, D.L. (1990), *Quality Measures in Design*, ACM Sigsoft Software Engineering Notes, vol 15, 1.

STM20 Number of Operand Occurrences

This metric is the number of operands in a software component and is one of the Halstead vocabulary analysis metrics. Halstead considered that a component is a series of tokens that can be defined as either operators or operands.

Unlike STOPN, this metric is the count of every instance of an operand in a file, regardless of whether or not it is distinct. STOPN only counts the operands that are distinct. The code example below has a STM20 value of 8.

```
void f( int a )      /* 1,2 -> f, a      */
{
    if ( a > 1 )      /* 3,4 -> a, 1      */
    {
        ++a;          /* 5 -> a           */
        while ( a > 1 ) /* 6,7 -> a, 1      */
        {
            --a;       /* 8 -> a           */
        }
    }
}
```

STM21 Number of Operator Occurrences

This metric is the number of operators in a software component and is one of the Halstead vocabulary analysis metrics. Halstead considered that a component is a series of tokens that can be defined as either operators or operands.

Unlike STOPT, this metric is the count of every instance of an operator in a file, regardless of whether or not it is distinct. STOPT only counts the operators that are distinct. The code example below has a STM21 value of 22.

```
void f( int a )      /* 1,2,3,4 -> void, (, int, ) */
{                    /* 5 -> {                  */
    if ( a > 1 )      /* 6,7,8,9 -> if, (, >, )    */
    {                /* 10 -> {                  */
        ++a;          /* 11,12 -> ++, ;          */
        while ( a > 1 ) /* 13,14,15,16 -> while, (, >, ) */
        {            /* 17 -> {                  */
            --a;       /* 18,19 -> --, ;          */
        }            /* 20 -> }                  */
    }                /* 21 -> }                  */
}                    /* 22 -> }                  */
```

STM22 Number of Statements

This metric is the number of statements in a software component. This is a count of semicolons in a file except for the following instances:

- within `for` expressions,
- within `struct` or `union` declarations/definitions,
- within comments,
- within literals,
- within preprocessor directives,
- within old-style C function parameter lists.

The code example below has a STM22 value of 5.

```
void f( int a )
{
    struct { int i;
            int j; }
        ij;          /* 1          */
    a = 1;            /* 2          */
    a = 1; a = 2;     /* 3,4        */
    if
        ( a >
          1 )
    {
        return;      /* 5          */
    }
}
```

STM28 Number of Non-Header Comments

This metric is a count of the occurrences of C or C++ style comments in a source file except for those that are within the header of a file. These comments are ones that appear after the first code token or preprocessor directive token. Comments within the file header are considered to be all comments in a file that precede the first code token.

STM28 is based on the method used to compute STCDN but differs from STCDN in that STCDN counts the visible characters within comments whereas STM28 counts the occurrences of comments. The

code example below has a STM28 value of 2.

```
/* Header comment 1      Count = 0 */
/* Header comment 2      Count = 0 */

/* Last header comment   Count = 0
   code follows */

#define CENT 100

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

STM33 Number of Internal Comments

This metric is a count of C style or C++ comments in a source file that are within functions or annotate a line of code at file scope. Comments within functions are all comments at block scope. Comments that annotate code are ones that start or end on the same line as code.

STM33 is based on the method used to compute STCDN but differs from STCDN in that STCDN counts the visible characters within comments whereas STM33 counts the occurrences of comments. The code example below has a STM33 value of 5.

```
/* Header comment 1      Count = 0 */
/* Header comment 2      Count = 0 */

/* Last header comment   Count = 0
   code follows */

#define CENT 100 /* Annotating comment STM33 = 1 */

int          /* Annotating comment STM33 = 2 */
    i;

/* Annotating comment STM33 = 3 */ int j; /*
   Annotating comment STM33 = 4 */
/* Non internal comment                                STM33 = 0 */
void f( int a )
{
    /* Block scope comment                                STM33 = 5 */
    a = CENT;
    return;
}
```

STMOB Code Mobility

$$\text{STMOB} = 100 * (\text{STDEV} - \text{STPRT}) / \text{STDEV}$$

This metric is a measure of the code portability as a percentage of the development time. In practice, most C code achieves over 95% portability; but many portability issues are caused, not by the language itself, but by the library support available on a given machine.

STOPN Halstead Distinct Operands

This is the number of distinct operands used in the file. Distinct operands are defined as unique identifiers and each occurrence of a literal.

Most literals, except 0 and 1, are usually distinct within a program. Since macros are usually used for fixed success and failure values (such as TRUE and FALSE), the differences in counting strategies are fairly minimal. The code example below has an STOPN value of 11.

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

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);
}
```

STOPT Halstead distinct operators

This covers any source code tokens not supplied by the user e.g. keywords, operators, punctuation and so on. STOPT is used in the calculation of a number of other metrics.

The code example below has an STOPT value of 11.

```
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 -> } */
```

STPRT Estimated Porting Time

This is an estimate of the number of programmer days required to port the source file. It is based on the number of warnings issued by QA C, assuming that there are upper and lower limits to how much code can be ported per day and work correctly. The upper and lower limits, and the porting scaling factor are all configurable in `qac.cfg`.

Three calculations are made to determine the STPRT metric:

1. A portability scale factor is calculated:

`port = weighting / 100 * port_scaling`

`weighting` is the sum of the portability weightings of all the warnings generated by QA C.

`port_scaling` is a portability scaling factor defined in `qac.cfg`.

If `port` is greater than 1.0, it will be reset to 1.0.

2. The porting rate is calculated in lines per day:

`lpd = min_port + (1 - port) * max_port`

<code>min_port</code>	minimum porting rate
<code>max_port</code>	maximum porting rate

3. The actual porting time is calculated:

$$\text{STPRT} = \text{STTPP} / \text{lpd}$$

STSCT Number of static variables

This metric is computed as the number of variables and functions declared static at file scope. The code example below has an STSCT value of 2.

```
static int other_result;    /* 1 */
int result;

static int test()          /* 2 */
{
    int x;
    int y;
    int z;

    z = 1;

    return (x + other_result);
}
```

STSHN Shannon Information Content

Also known as the “entropy” H, this metric is a widely recognised algorithm for estimating the program space required to encode the functions in a source file. STSHN is measured in bits and is calculated as follows:

$$\text{STSHN} = \text{STZIP} * \log_2 (\sqrt{(\text{STOPN} + \text{STOPT})} + \ln (\text{STOPN} + \text{STOPT}))$$

STTDE	Embedded Total Months
STTDO	Organic Total Months
STTDS	Semi-detached Total Months

The COCOMO metrics are produced for each source code file. You can display an accurate estimate of development costs for a whole project by choosing the COCOMO Cost Model option from the Reports menu. Refer to Chapter 6: Reports for a discussion of the COCOMO cost model and an explanation of the three programming modes, Organic, Semi-detached, and Embedded.

These metrics are a measure of the total number of months required to develop the source code in the respective environments.

$$\begin{aligned} \text{STTDE} &= 2.5 * \text{STBME}^{0.32} \\ \text{STTDO} &= 2.5 * \text{STBMO}^{0.38} \\ \text{STTDS} &= 2.5 * \text{STBMS}^{0.35} \end{aligned}$$

STTLN Total Preprocessed Code Lines

This metric is a count of the total amount of lines in the translation unit after preprocessing. The preprocessed file will reflect the processing of include files, preprocessor directives and the stripping of comment lines.

STTOT Total number of tokens Used

This metric is the total number of tokens, not distinct tokens, in the source file. The code example below has an STTOT value of 19.

```
int test()                /* 1,2,3,4 */
{                          /* 5 */
    int x;                /* 6,7,8
    int y;                /* 9,10,11 */
    /*Excluded Comment*/
    return (x + y);       /* 12,13,14,15,16,17,18
*/
}                          /* 19 */
```

STTPP Total Unprocessed Source Lines

This metric is a count of the total number of source lines in the file before preprocessing.

STVAR Number of identifiers

The total number of distinct identifiers. The code example below has an STVAR value of 5.

```
int other_result;           /* 1 */
extern int result;         /* 2 */

int test()                 /* 3 */
{
    int x;                 /* 4 */
    int y;                 /* 5 */

    return (x + y + other_result);
}
```

STVOL Program Volume

This is a measure of the number of bits required for a uniform binary encoding of the program text and is used to calculate various Halstead vocabulary metrics.

The following is the calculation for the program volume:

$$\text{STVOL} = \text{STTOT} * \log_2 (\text{STOPN} + \text{STOPT})$$

STZIP Zipf Prediction of STTOT

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

See STHAL.

Project-Wide Metrics

These metrics are calculated once per complete project. The complete

list is as follows:

STNRA	Number of Recursions Across Project
STNEA	Number of Entry Points Across Project
STNFA	Number of Functions Across Project
STCYA	Cyclomatic Complexity Across Project

STNRA Number of Recursions Across Project

This metric is defined to be the number of recursive paths in the call graph of the project's functions. A recursive path can be for one or more functions. The minimum, and often desirable, value of this metric is zero. Values greater than zero indicate the number of distinct loops in the call graph.

STNEA Number of Entry points Across Project

This metric is the number of functions that are not called in the project. It is the number of nodes in the call graph which are not themselves called. For example `main()` is not called hence the minimum (target) value of 1.

STNFA Number of Functions Across Project

This metric is the number of function definitions in the project. Note that this metric is the sum of QA C's STFNC metric values for each source file included in the project.

STCYA Cyclomatic Complexity Across Project

This metric is the sum of cyclomatic complexity values for each function definition in the project. Note that this metric is the sum of QA C's STCYC metric values for each source file included in the project.

APPENDIX E

Program Return Codes

The following chart lists the return codes from various executable programs and matching GUI messages where applicable. The Notes column outlines the binaries that each return code applies to.

Analysis Status Reported	Return Code	Explanation	Notes
Completed	0	Successful completion	(1)
Failed: parser hard error	1	Number of hard errors exceeds limit specified by the <code>-maxerr</code> option	(2)
-	1-99	Represents the highest level of non-suppressed message encountered <ul style="list-style-type: none"> • <code>#include</code> file not found • Recursive calls to files 	(3)
Failed: parser configuration	2	<ul style="list-style-type: none"> • No end to comment block • <code>#error</code> directive encountered 	(2)
Failed: configuration	10	<ul style="list-style-type: none"> • general configuration problem • missing <code>internat.rsc</code> file 	(1)
Failed: system error	18	Exception generated from parser executable	(1)
Failed: fatal error	19	Memory allocation error, or untrapped exception	(1)
Failed: licensing error	11	Product licensing failure	(4)

Analysis Status Reported	Return Code	Explanation	Notes
Failed: GUI personality	-	project-defined personality file not located	(1)
Failed: process error	-	Windows process error in analysis commencement	(1)
Failed: Sec Anal Configuration	-	Failure in secondary analysis configuration	

Notes:

- (1) Applicable to all executables (`qac.exe`, `pal.exe`, `prjdsp.exe`, `errwrt.exe`, and `errsum.exe`)
- (2) Applicable to `qac.exe` only
- (3) Applicable to `errdsp.exe` only
- (4) Applicable to `qac.exe` or `errdsp.exe` only

APPENDIX F

Metric Output File

During analysis, QA C writes information to the `.met` file about:

- include file usage,
- function calls,
- references to objects with external linkage,
- definitions and declarations of all functions, variables, macros, labels, tags, and typedefs,
- function control flow structure information,
- metric values,
- `#pragma` directives,
- Range of literals used.

Records are written sequentially to the file as QA C processes the translation unit. Their format is described below.

Relationship records

```
<R>I includingfile    includedfile    lineno
```

Generated whenever a `#include` statement is encountered. The *includingfile* may be either a source file or another included file.

```
<R>C callingfunction  calledfunction lineno
```

Generated for every instance of a function call with either internal or external linkage.

```
<R>X callingfunction  identifier    lineno  flag
```

Generated for every reference to an identifier with external linkage. This record will have either `R` in the *flag* field if the data is read or `W` if the data is written.

Note that every function call will generate an <R>C record. If the function is external, it will also generate an <R>X record.

Relationship type records

```
<RDEF> I Includes
<RDEF> C Calls
<RDEF> X Refers-to
```

These records exist simply to specify the description (i.e. “Includes”, “Refers-to” or “Calls”) of the corresponding <R> relationship type to the QA C menu system.

Note that a maximum of one <RDEF> record of each type will be present in a .met file and that the record will only be generated if one or more of the corresponding <R> relationship records are present.

External Reference Records

```
<EXTD> external_identifier filename lineno
```

Output for every definition of an object with external linkage.

```
<EXTN> external_identifier filename lineno
```

Output for every declaration of an object with external linkage.

```
<EXTV> identifier external_identifier lineno
```

Output for every definition of an object which is initialised using an object with external linkage.

```
<EXTT> identifier filename lineno
```

Output for every tentative definition of an object. A tentative definition of an object is generated by declaring an identifier at file scope without an initialiser and without a storage class specifier, or with the storage-class specifier `static`. The first tentative definition is followed by an <EXTN> record.

Example:

The file scope declarations:

```
int count = 0;
int *pcount = &count;
extern int major;
static int ifunc();
```

```
extern int efunc();
int (*pfi)() = ifunc;
int (*pfe)() = efunc;
int tdef;
```

will generate the following external reference records:

```
<EXTD> count file.c 1
<EXTD> pcount file.c 2
<EXTV> pcount count 2
<EXTN> major file.c 3
<EXTN> efunc file.c 5
<EXTD> pfi file.c 6
<EXTD> pfe file.c 7
<EXTV> pfe efunc 7
<EXTT> tdef file.c 8
<EXTN> tdef file.c 8
```

Define records

```
<DEFINE> identifier filename lineno inc def space scope
linkage type flag
```

A record generated for every identifier including functions, variables, macros, labels, tags, and typedefs.

Field	Content	Description
<i>identifier</i>		Identifier name ("-" for unnamed bit-field)
<i>filename</i>		File in which declaration appears
<i>lineno</i>		Line at which declaration appears
<i>inc</i>	I	Included file
	N	Not included, i.e. main source file
<i>def</i>	DF	Definition
	DC	Declaration
	DR	Re-declaration
	DI	Implicit definition
<i>space</i>	LB	Label
	TG	Structure, union or enumeration tag
	MB	Member of structure or union
	OT	Ordinary identifier – typedef
	OF	Ordinary identifier – function
	OM	Ordinary identifier – macro
	OV	Ordinary identifier – variable
<i>scope</i>	N	Function scope (for labels)
	F	File scope
	B	Block scope
	P	Function prototype scope

	-	Not applicable (macros)
<i>linkage</i>	I	Internal
	X	External
	N	none – local or typedef
	-	not applicable (macros)
<i>Type</i>	See table below	Symbolic type code, a shorthand notation used to describe a datatype.
<i>Flag</i>	F	Function-like macro
	O	Object-like macro
	S	static storage duration (local and global declarations. Not the same as <code>static</code> keyword.)
	R	Function parameter
	-	no further information

Symbolic Type Code	Datatype
<code>nc</code>	signed char
<code>uc</code>	unsigned char
<code>ns, ni, nl, nll</code>	short, int, long, long long
<code>us, ui, ul, ull</code>	unsigned short, unsigned int, unsigned long, unsigned long long
<code>fs, ff, fl</code>	float, double, long double
<code>pX</code>	pointer to X
<code>(Y,A1,..,An)</code>	function returning Y taking A1.. An
<code>[N,X]</code>	N-element array of X
<code>g</code>	pointer to void
<code>.</code>	as in 'function returning void'
<code>{sNAME}</code>	struct NAME
<code>{uNAME}</code>	union NAME
<code>{eNAME}</code>	enum NAME
<code>ne</code>	enumeration constant
<code>nb</code>	signed bit-field
<code>ub</code>	unsigned bit-field
<code>:</code>	as in 'void f(...)'
<code>=X</code>	const X
<code>^X</code>	volatile X

Example:**Source code:**

```
#define M(x) ((x) + 10)
int a = 0;
extern int b(char *c);
typedef double D;
```

```

static int e(D f);
extern int g;
static float h (int i)
{
    int j;
    struct k
    {
        int l;
        char m;
    } n;

    j = i + g;
    return(j);
}

```

.met file <DEFINE> records:

```

<DEFINE> M C:\Bugs\Code\TEST1.c 1 N DF OM - - - F
<DEFINE> a C:\Bugs\Code\TEST1.c 2 N DF OV F X ni S
<DEFINE> c C:\Bugs\Code\TEST1.c 3 N DC OV P N puc R
<DEFINE> b C:\Bugs\Code\TEST1.c 3 N DC OF F X (ni,puc) -
<DEFINE> d C:\Bugs\Code\TEST1.c 4 N DF OT F N ff -
<DEFINE> f C:\Bugs\Code\TEST1.c 5 N DC OV P N ff R
<DEFINE> e C:\Bugs\Code\TEST1.c 5 N DC OF F I (ni,ff) -
<DEFINE> g C:\Bugs\Code\TEST1.c 6 N DC OV F X ni S
<DEFINE> i C:\Bugs\Code\TEST1.c 7 N DC OV B N ni R
<DEFINE> h C:\Bugs\Code\TEST1.c 7 N DF OF F I (fs,ni) -
<DEFINE> j C:\Bugs\Code\TEST1.c 9 N DF OV B N ni -
<DEFINE> k C:\Bugs\Code\TEST1.c 10 N DF TG B N {sk} -
<DEFINE> l C:\Bugs\Code\TEST1.c 12 N DF MB B N ni -
<DEFINE> m C:\Bugs\Code\TEST1.c 13 N DF MB B N uc -
<DEFINE> n C:\Bugs\Code\TEST1.c 14 N DF OV B N {sk} -

```

Control Graph Records

Example:

```

<CTLG> functionname l
<CTLG> 1 2
<CTLG> 2 3
<CTLG> 2 4
<CTLG> 3 4
<CTLGN> 35 35 36 38

```

<CTLG> records are generated for every function to describe its structure. The sole purpose is so that the function can be displayed in the form of a Function Structure Diagram.

The first record contains the function name, the symbolic type description of the function and a count of the number of exits from the function.

Subsequent records represent connections between a pair of nodes in the control flow structure.

A <CTLGN> record follows the complete set of <CTLG> records, and is a list of the line numbers of all nodes in sequential order. See Chapter 7: Viewing Function Structure Source Code for application of the line number information.

Metrics Records

<S> MetricName MetricValues

Example:

```
<S>STCYC  8
<S>STMIF  0
<S>STSUB  3
```

A record, <S>STNAM, is generated for each file or function, containing its name, before outputting its metrics.

A record, <S>STFIL, is also generated for each file or function, containing the name of the analysed source file.

See Appendix D: The Calculation of Metrics for a discussion of all the various function and file metrics.

Pragma Records

<PRAGMA> filename lineno pragma_name

A record is generated for all #pragma directives.

Literal Records

This record type identifies all integer and floating constants used in source code including:

- constants used to dimension an array,
- constants used to initialise a scalar object,
- constants used in expressions.

It excludes:

- Bit-field width,
- Case label constant,
- The user specified values for enumeration constants,
- Constant zero 0 or 0.0,
- Initializers of struct, array, union type,
- Character constants (including escape sequences),
- All constants used in preprocessor expressions,
- All constants defined within a macro.

The output also separates positive and negative literal values, and records any suffixing being used as well as the underlying type of the literal.

The form of the record is:

```
<LIT> literal suffix sign filename lineno colno inc
      lit-type type scope context-flag
```

with each entry expanded as follows:

Field	Content	Description
<i>literal</i>		Literal reproduced verbatim, excl suffix
<i>suffix</i>		Suffix reproduced verbatim
<i>sign</i>	P	Positive
	N	Negative
	-	Not applicable
<i>filename</i>		Fully qualified filename
<i>lineno</i>		Line number in source code
<i>colno</i>		Column number in source code
<i>inc</i>	I	Included file
	N	Not an included file
<i>lit-type</i>	DN	Decimal number (e.g. 123)
	HN	Hex number (e.g. 0xFF00)

	ON	Octal number (e.g. 015)
	FN	Floating number (e.g. 12.34)
	BN	Binary number (e.g. 0b01011100)
	CN	Character literal (e.g. 'q')
	SN	String literal (e.g. "hello")
	CE	Simple escape sequence ('\n')
	HE	Hex escape sequence ('\0x0D')
	OE	Octal escape sequence ('\015')
	CW	Wide character constant (e.g. L'x')
	SW	Wide string literal (L"hello")
<i>type</i>		Symbolic type code
<i>scope</i>	F	File scope
	B	Block scope
	P	Prototype scope
<i>context- flag</i>	M	macro constant
	D	array dimension
	B	Bitfield size
	E	enum constant
	O	scalar object initialiser
	A	array initialiser
	S	Structure initialiser
	U	union initialiser
	P	Constant in preprocessor expression
	C	case label constant
	X	others, i.e. constants in ordinary statements

APPENDIX G

QA C Utilities

QA C is shipped with a number of utilities that can be used when creating additional checks or customised reports.

r_basename

Similar to the Unix `basename` utility, this utility removes all suffixes irrespective of further arguments. For example:

```
r_basename "C:\Program Files\PRQA\QAC\file.c"
```

will print:

```
file
```

r_close

The `r_close` utility performs an analysis of identifier names that differ by only one character. It reads a list of identifier names from a file, or from standard input, and generates a report on all pairs that could be confused.

Running `r_close` with the `-report` option will generate output as two centre-justified columns. Each line contains a pair of names that could be confused. A header line and a summary are also produced.

Without the `-report` option, the close name pairs are output in records with the tag `<CLOSE>`. The summary is output as:

```
<CLOSEPAIR> 30 180
<CLOSENESS> 0.001862
```

`<CLOSEPAIR>` shows the number of close pairs and the number of total names. `<CLOSENESS>` shows the closeness metric that is calculated by dividing the number of close pairs by the number of possible pairs. The number of possible pairs is calculated as $n * (n - 1) / 2$ for n names.

For example:

```
type analyze.c.met | r_grep -p"<DEFINE>" | r_fields +1
| r_uniq | r_close -report
```

will produce the following output:

The following identifiers might cause confusion because of their similarity:

```
output_style output_style
change changed
change changes
inserted insert
deleted delete
deleted deletes
line0 line1
...
```

```
123 pairs of close names were found in 265 names.
The "closeness" metric is 0.003516
```

r_fields

The `r_fields` utility allows you to select a subset of fields for each record in the `.met` file. Setting the option `+n` will select the field `n`. Setting the option `-n` will omit the field `n`. For example:

```
type alloca.cpp.met | r_grep -p"<DEFINE>"
| r_fields +2 +3 +1 +6
```

will extract the filename, line number, identifier and name space as follows:

```
alloca.cpp 30    SCCSid          OV
alloca.cpp 51    pointer         OT
alloca.cpp 54    NULL            OM
alloca.cpp 56    free            OF
alloca.cpp 57    xmalloc         OF
alloca.cpp 70    STACK_DIRECTION OM
...
```

You could produce the same results by removing the unwanted fields. For example:

```
type alloca.cpp.met | r_grep -p"<DEFINE>"
| r_fields -0 -4 -5 -7 -8 -9
```

will produce:

```
SCCSid      alloca.cpp 30    OV
pointer     alloca.cpp 51    OT
NULL        alloca.cpp 54    OM
free        alloca.cpp 56    OF
```

...

These fields could be arranged in order by adding:

```
| r_fields +1 +2 +0 +3
```

to the above command.

Note:

Since fields are separated by spaces, filenames that include embedded spaces will be spread over several fields.

r_grep

The `r_grep` utility is a portable pattern matcher similar to the Unix utility `grep`. It reads from a file, or standard input, and outputs lines of the `.met` file that contain one or more of the specified patterns. You can also use this utility to print out lines that do not contain any of the patterns.

To search for a pattern, set the `-p` option. The pattern should be plain text as `r_grep` does not understand regular expressions like `grep`. `r_grep` is more like `fgrep` in this respect. Multiple patterns can be specified by setting more than one `-p` option. For example:

```
r_grep -p"<S>STNAM" -p"<S>STPTH" alloca.cpp.met
```

will display the function name and path count metrics for the file `alloca.cpp.met` as follows:

```
<S>STNAM  find_stack_direction(.,)
<S>STPTH  3
<S>STNAM  alloca(puc,ui)
<S>STPTH  12
<S>STNAM  "D:\Program Files\PRQA\QAC\demo\alloca.cpp"
```

The last entry is the file name that appears in the file-based metrics so that other tools can distinguish between file-based and function-based groups of metrics.

To exclude those records which do not contain any of the specified patterns, you can set the `-v` option. For example, to prevent the last entry appearing in the above example, you could use:

```
r_grep -p"<S>STNAM" -p"<S>STPTH" alloca.cpp.met
| r_grep -v -p"alloca.cpp"
```

r_sort

This is a version of the Unix `sort` utility and reads standard input, sorts the lines and writes the results to the `.met` file. You can sort selected fields by specifying the number of fields. For example:

```
type data | r_sort +2 +1
```

will sort `data` on the third field followed by the second field. Fields are numbered from zero. A maximum of 100,000 records can be sorted.

r_uniq

This is a version of the Unix `uniq` utility and copies standard input to standard output, omitting the duplicated lines. The input must be sorted.

APPENDIX H

Code Suppression

This appendix deals with the specification and syntax for diagnostic suppressions invoked through code comments. Suppression of messages is possible against primary, secondary and CMA analysis. The implementation is designed to keep suppression directives separate from generation of diagnostics, so that the viewing tools determine which diagnostics to suppress.

The implementation also significantly extends our existing `#pragma PRQA_MESSAGES_OFF` functionality. The `#pragma` form of suppression is still supported, although deprecated functionality.

Atoms of Diagnostics

All diagnostics (messages) have the following basic information: message number and location. The concept of a location is the minimum information to describe the code context to which a message is referring. Currently this information is a data tuple `<file,line,column>`. This is all that is required to identify any location in source code. More complex and higher level semantic locations can all be mapped back to sets of these simpler locations.

Code-based Annotations

Code-based annotations are used to define locations and suppressions. Location tags are useful to tag specific locations that may be the subject of suppressions defined elsewhere.

All annotations are written in comments (either C or C++ style) and must be prefixed with the string `PRQA`.

Location Tag Syntax

The syntax for **location tags** is:

```
// PRQA L:tag_name [location_specifier]
```

where:

<code>tag_name</code>	is a special tag that can be used by other annotations to refer to this location.
<code>location_specifier</code>	specifies an optional location which is used to form the boundary for a location range starting here.

Example usage:

```
// PRQA L:tag_name
```

Create a location tag for the current line. `tag_name` may be referenced by other locations tags or by a suppression annotation.

```
// PRQA L:tag_name n
```

Create a location tag for the current line and for the following `n` lines.

```
// PRQA L:tag_name tag_name_end
```

Create a location tag which includes all locations between the current line and `tag_name_end` inclusive. No code-based suppression entry can refer to this range location, and it will only be supported in a future release when configuration-based suppressions can be defined.

Additional Constraints:

Real line numbers are used in all cases; lines containing continuations will not be treated as a single line.

When specifying a range location, `tag_name_end` is the first location found after the current location tag, and this location tag must appear in the current file otherwise an annotation error is generated. The end location for an annotation is always taken to be the line of the 'PRQA L' annotation, even if that annotation itself defines a range.

The current location is taken as column zero of the line containing the string `PRQA`.

Predefined Location Tags

There is a predefined location tag, which has been added to simplify a common suppression requirement:

EOF End of the current file

Note: You cannot redefine predefined location tags.

Suppression Syntax

The syntax for **suppressions** is:

```
// PRQA S[:tag_name] message_specifier  
    [ location_specifier ]
```

where:

tag_name	is an optional tag that can be used externally to refer to this suppression, for example an external deviation management system.
message_specifier	represents the set of messages this suppression should apply to.
location_specifier	is either a location tag representing the end of the suppression range, a number of physical lines, or a continuous location specifier.

Additional Constraints:

Suppression annotations without tags will be given a generic tag unique to that file.

A suppression annotation without a `location_specifier` refers to the current line only.

Where `location_specifier` is a location tag, then the suppression applies from the current line until the first location tag appearing in the current file.

Continuous Suppression Syntax

A special syntax will be implemented as a direct replacement to the existing `#pragma PRQA_MESSAGES_[ON|OFF]` syntax available in the tools today. Continuous suppressions are implemented using only a suppression annotation and therefore always start from the current line.

The syntax for **continuous suppressions** is:

```
// PRQA S message_specifier [++|--]
```

where:

<code>message specifier</code>	represents the set of messages this suppression should apply to.
<code>++</code>	specifies that the set of messages in <code>message_specifier</code> be added to the current set of suppressed messages from this line onwards.
<code>--</code>	specifies that the set of messages in <code>message_specifier</code> be removed from the current set of suppressed messages after the current line onwards.

Example usage (see the Use Case Examples below for detailed usage):

```
// PRQA S 100 ++
```

Message 100 will be added to the set of continuous suppressions from the current line onwards.

```
// PRQA S 200 --
```

Message 200 will be removed from the set of continuous suppressions from after the current line onwards.

Additional Constraints:

Continuous suppressions do not change the effects of other forms of suppression annotations.

The set of continuous suppressions at the beginning of an included file will be inherited from the including file.

The set of messages suppressed by continuous suppressions before and after an `#include` directive does not change for the current source file. The special behaviour of continuous suppressions is that messages “inherited” into a header file can be removed inside the header file through a matching continuous suppression removal, using syntax:

```
// PRQA S message_specifier --
```

Because of this special behaviour, there is an important restriction. Header file diagnostics generated for the first line/column of the header file cannot be suppressed, due to an architectural constraint in implementation.

Use-Case Examples

The aim of this section is to provide examples of suppression in various use cases, covering code annotations and the resulting effect.

Single instance suppression

Simple suppression of one message on the current line

```
int i; // PRQA S 100
```

Effects: Message 100 will be suppressed for any messages appearing on the line of the declaration of `i`.

Suppression of a blank line

```
// PRQA S 100  
int i;
```

Effects: Message 100 will be suppressed for any messages appearing on the line above the declaration of `i`.

Invalid suppression: missing message specification

```
// PRQA S:s1  
int i;
```

Effects: Message 4826 generated for invalid suppression syntax, missing message specifier. It does not matter whether the suppression contains a tag like `s1`.

Range suppression using location tags

Suppress between current line and tag (example 1)

```
int i;  
int j;  
// PRQA S 100 L1  
++i;  
// PRQA L:L1  
++j;
```

Effects: Message 100 against ++i suppressed.

Suppress between current line and tag (example 2)

```
int i;  
int j;  
// PRQA S 100 L1  
++i;  
// PRQA L:L1  
++j;  
// PRQA L:L1
```

Effects: Message 100 against ++i suppressed. No suppression against ++j.

Suppress between current line and tag (example 3)

```
int i; // PRQA S 100 L1  
int j; // PRQA L:L1  
int k;
```

Effects: Message 100 against i and j suppressed.

Suppression range start and end on same line (Error)

```
/* PRQA S 100 L1 */ int i; /* PRQA L:L1 */ // ERROR  
int j; // PRQA S 100 // OK
```

Effects: Suppression configuration message 4811 is generated against the first suppression entry, since the suppression range start and end are on the same line. The second suppression entry is a correct equivalent format.

Suppress across a header file (example 1)

```
// foo.h
int i;

// PRQA S 100 L1
#include "foo.h"
// PRQA L:L1
```

Effects: Message 100 against `i` suppressed.

Suppress across a header file (example 2)

```
// foo.h
int i;

#include "foo.h" // PRQA S 100
```

Effects: Message 100 against `i` suppressed.

Suppression range end not in same file (Error)

```
// foo.h
// PRQA L:L2

// foo.cc
// PRQA S 200 L2 // ERROR - L2 not found in this file
#include "foo.h"

// PRQA S 100 L1 // OK - L1 does appear in this file
// PRQA L:L1
```

Effects: Suppression configuration message 4810 is generated against the first suppression entry, since the suppression range end `L2` is not found in this file (`foo.cc`). The second suppression entry is a correct format.

Range suppression using line counting**For the next 'n' lines inclusive (example 1)**

```
// PRQA S 100 1
int i;
int j;
int k;
```

Effects: Message 100 against `i` suppressed.

For the next 'n' lines inclusive (example 2)

```
int i; // PRQA S 100 1
int j;
int k;
```

Effects: Message 100 against `i` and `j` suppressed.

Suppression beyond remaining lines in file

```
// foo.h
// PRQA S 100 100

// foo.cc
#include "foo.h"
int i;
```

Effects: Message 100 against `i` not suppressed. Suppressions do not pass from included files back to their including files.

From current physical line to end-of-file

```
int i;
int j; // PRQA S 100 EOF
int k;
```

Effects: Message 100 against `j` and `k` suppressed.

Attempted redefinition of predefined location tag

```
int i;
// PRQA L:EOF // cannot redefine tag 'EOF' (Msg 4828)
int j;
```

Effects: Error message 4828 generated.

Continuous suppressions

Simple on/off continuous suppression

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
int k;
// PRQA S 100 --
int l;
```

Effects: Message 100 suppressed for `j` and `k`.

Simple on/off continuous suppression with include entry

```
// foo.h
int j;

// foo.cc
int i;
// PRQA S 100 ++
#include "foo.h"
int k;
// PRQA S 100 --
int l;
```

Effects: Message 100 suppressed for `j` and `k`. This shows the suppression around the `#include` is also active inside the header file.

Continuous suppression with messages added

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
// PRQA S 200 ++
int k;
// PRQA S 100,200 --
int l;
```

Effects: Messages 100 and 200 suppressed for `k`. Message 100 only suppressed for `j`.

Continuous suppression with messages removed

```
// foo.cc
int i;
// PRQA S 100,200 ++
int j;
// PRQA S 200 --
int k;
// PRQA S 100 --
int l;
```

Effects: Messages 100 and 200 are suppressed for *j*. Message 100 suppressed for *k*.

Non-overlapping combination of continuous and explicit suppression

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
// PRQA S 100 --
int k; // PRQA S 100
int l;
```

Effects: Message 100 suppressed for both *j* and *k*.

Overlapping combination of continuous and explicit suppression (example 1)

```
// foo.cc
int i;
// PRQA S 100 ++
int j; // PRQA S:S1 100
// PRQA S 100 --
int k;
```

Effects: Message 100 suppressed for *j*. Disabling the suppression *s1* will not cause the message to be displayed, as it is also suppressed by the continuous suppression.

Overlapping combination of continuous and explicit suppression (example 2)

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
// PRQA S:S1 100 L1
int k;
// PRQA S 100 --
int l;
// PRQA L:L1
int m;
```

Effects: Message 100 suppressed for `j`, `k` and `l`. Disabling suppression `S1` changes the suppression to that of `j` and `k` only, `l` will no longer be suppressed.

Continuous suppression configuration failure

```
// foo.cc
int i;
// PRQA S:S1 100 ++
int j;
```

Effects: Suppression configuration message 4812 generated, due to invalid `s1` tag.

Suppressions in header files

Range suppression at end of header file

```
// foo.h
// PRQA S 100 10
int j;
// EOF foo.h

// foo.cc
#include "foo.h"
int i;
```

Effects: Message 100 suppressed for `j` only.

Invalid range suppression from a header file into a source file

```
// bar.h
// PRQA S 100 L1
int i;

// foo.cc
#include "bar.h"
int j;
// PRQA L:L1
int k;
```

Effects: The attempted suppression in `bar.h` does not persist into the remainder of the including file `foo.cc`, and a suppression configuration message will be issued. This can be fixed by moving the suppression to within the source file `foo.cc`.

Non-terminated file suppression

```
// foo.h
// PRQA S 200 ++
int j;

// foo.cc
int i;
// PRQA S 100 ++
#include "foo.h"
int k;
// PRQA S 100 --
int l;
```

Effects: Messages 100 and 200 are suppressed for `j`. Message 100 suppressed for `k`. This shows that the suppression defined inside the include, although not terminated, does not apply after the end of file.

Force include suppression entries

Unlike all other headers, declarations and entries in the Force Include file are treated as if they appeared at line 0 of the main source file.

This enables code-based suppressions to exist outside of the source code, but having the same effect as if located in each source file.

This special treatment does not extend to files and their entries `#included` from the Force Include file.

Range suppression in Force Include file (example 1)

```
// forceincl.h
// PRQA S 100 3

// foo.h
int i;

// foo.cc
#include "foo.h"
int j;
int k;
```

Effects: Message 100 is suppressed for `i` and `j`.

Range suppression in Force Include file (example 2)

```
// bar.h
// PRQA S 100 10

// forceincl.h
#include "bar.h"

// foo.cc
int j;
int k;
```

Effects: None of the declarations are affected by suppressions in the above example.

Non-terminated continuous suppression combined with Force Include

```
// forceinclude.h
int i;
//PRQA S 200 ++
int j;

// foo.cc
int k;
// PRQA S 100 ++
int l;
// PRQA S 100 --
int m;
```

Effects: Message 200 suppressed for `i`, `j`, `k`, `l` and `m`. Message 100 is also suppressed for `l`. It may appear strange that `i` also has the message suppressed, however due to `forceincludes` being treated as if they appear on line 0 column 0 of the main source file, then technically

all of the contents of the force include are treated as if they have the same line.

Location suppression combined with Force Include

```
// forceinclude.h
int i;
//PRQA S 200 L1
int j;

// foo.cc
int k;
// PRQA L:L1
int l;
```

Effects: Message 200 suppressed for *i*, *j*, and *k*. The same comment as above about suppressing *i* applies in this case.

Suppression input failures

Invalid/missing annotation type

```
// PRQA L:L1           // Location annotation
// PRQA S 100          // Suppression annotation

// PRQA X              // ERROR: Unknown annotation type
// PRQA                // ERROR: No annotation specified
```

Effects: Message 4820 is generated when a blank annotation or an invalid annotation kind is specified.

Missing annotation tag

```
// PRQA L:L1           // Location annotation
// PRQA L L1           // ERROR: Missing ':'

// PRQA S 100          // Ok, no tag specified
// PRQA S:S1 100       // Ok, tag specified
```

Effects: For suppression annotations, the tag is optional. For location annotations, they are mandatory. Message 4821 is generated for an improperly formatted or missing location tag.

Invalid annotation tag

```
// PRQA S:S1 100          // Annotation tag 'S1'  
// PRQA L:L1              // Annotation tag 'L1'  
  
// PRQA L:123             // ERROR: improper format  
// PRQA S:123             // ERROR: improper format
```

Effects: Annotation tags must begin with a letter. Message 4822 is generated for a wrongly named tag.

General syntax error

There are two types of annotation syntax. The syntax for location annotations is:

```
// PRQA L:<tag-name> [<end-tag-opt>]
```

And for suppression annotations is:

```
// PRQA S [:<tag-name-opt>] <msg-spec> [<end-tag-opt>]
```

In some cases it is not possible to determine the exact cause of a failure to read an annotation, and so this general error may occur.

```
// PRQA S 100 -           // ERROR
```

Effects: Message 4823 is generated for wrong suppression syntax.

Invalid character in tag name

Annotation tags may only contain letters of the alphabet, numbers and underscores. When specifying continuous suppressions, the special tags '++' and '--' are allowed.

```
// PRQA S 100 A1          // OK  
// PRQA S 100 A_1         // OK  
// PRQA S 100 ++          // OK  
  
// PRQA L:A1              // OK  
// PRQA L:A_1             // OK
```

It is not possible to use characters other than these when specifying an annotation tag.

```
// PRQA S 100 A&1          // ERROR in tag name  
// PRQA S 100 A@1          // ERROR in tag name
```

Effects: Depending on token sequence, message 4824 or 4825 is generated for invalid characters in tag name.

APPENDIX I

Naming Convention Checking

Introduction

The Name Checker is a secondary analysis process that can enforce a naming convention. It will issue diagnostic messages against identifiers that do not match a specified regular expression.

The Name Checker is typically configured to run from the GUI, but can also be run from the command line. It analyses a single file at a time. As a prior step, Primary Analysis must have been completed successfully.

Configuration Basics

The Name Checker needs the following two inputs:

- Source file and corresponding .met and .err files from Primary Analysis.
- Configuration file specifying a set of naming rules – this is normally specified in the Message Personality, Secondary Analysis set-up.

For any additional messages mentioned in the configuration file, it will also be necessary to provide these through a user message file. Message 4800 is reserved as a default configuration entry, and provided in the base message file for this purpose.

Configuration File

This file, also referred to as the name rule file, contains the rules to specify for the Name Checker. As a minimum, each rule should specify the pattern (regular expression) that would apply to each type of identifier (e.g. macros, functions, typedefs, variables etc). It is possible to restrict which sort of identifiers the pattern matching will be applied to by adding extra items to the rule.

Rules are applied sequentially from the input configuration; no checking is done for contradictory rules, and more than one rule can apply to a particular identifier. For example, a company coding standard may have the following 2 rules:

- 1) Identifiers must be 31 characters or less in length.
- 2) Function names must begin with an upper case letter.

It is possible to have a function that violates either or both of these rules, so there may be cases where an identifier has more than one message issued against it.

Rule Format (JSON Syntax)

The configuration file is based on the JSON syntax – see www.json.org for full details. The only difference is that the Configuration File uses single quotes to de-limit strings, rather than double quotes.

The JSON object is defined as a rule by beginning the line with 'rule='. Blank lines and those starting with a hash '#' are ignored. Each rule must be on a separate line and not split across two lines.

All the Rule Values are strings, except for the following:

- Invert is a Boolean (specified with a the literal text true or false)
- Message Number is a positive number

The Rule Names and Values (where applicable) match those in the <DEFINE> record in the met file – see Appendix F: Metric Output File.

Note that an empty rule (`rule={}`) is allowed. This will issue message number 4800 against all identifiers. Similarly the rule `rule={'space': 'OV'}` will issue message number 4800 against all variables. This can be useful for checking that rules will apply to the intended identifiers before moving on to add the Regular Expression.

Rule Names

This section describes the Rule Names and what values they can take. Some configuration options are only available in either QA C or QA

C++, which are noted below accordingly.

Filename

Name: filename

Value: string, regular expression

This object is used to specify particular files that the rule applies to. It is a Regular Expression. Only files whose name matches the regular expression have the rule applied to it. Note that the full path of the file is compared to the Regular Expression, so it is important to include the end-of-line anchor (\$) in the regular expression to avoid matching fragments of the full path.

Included File

Name: inc

Value: string from list

This specifies that the rule will only be applied to either the main source file or those files included via the pre-processor directive #include.

Value	Description
I	Included file
N	Not included – i.e. main source file

When omitted the rule will apply to both source and included files.

Definition

Name: def

Value: string from list

This restricts the rule to different forms of declaration/definition.

Value	Description
DF	Definition
DC	Declaration
DR	Re-declaration
DI	Implicit Definition
DS	Class / Function Specialisation (C++ only)

When omitted the rule applies to all definitions, declarations and (in the case of QA C++) specialisations. Combinations of these values can be obtained by separating entries with commas or spaces.

Space

Name: space

Value: string from list

This restricts the rule to different types of identifier.

Value	Description
LB	label (QA C only)
TG	class (QA C++ only), structure, union or enumeration tag
MB	member of structure or union (QA C only)
OT	typedef
OF	function
OE	enumerator (QA C++ only)
OM	macro
OV	variable

When omitted the rule is applied to all identifiers. Combinations of these values can be obtained by separating entries with commas or spaces.

Scope

Name: scope

Value: string from list

This restricts the rule in terms of scope of the identifier. Note that scope and linkage are often confused.

Value	Description
N	function scope – labels (QA C only)
F	file
B	block
P	function prototype
C	class (QA C++ only)
T	template (QA C++ only)
S	namespace (QA C++ only)

When omitted the rule applies to identifiers with any scope. Combinations of these values can be obtained by separating entries with commas or spaces.

Linkage

Name: linkage

Value: string from list

This restricts the rule in terms of the linkage of the identifier.

Value	Description
I	internal
X	external
N	none – local or typedef

When omitted the rule applies to identifiers with any linkage.

Type

Name: type

Value: string, regular expression

This restricts the rule in terms of the type of an identifier. This is a Regular Expression utilising a symbolic shorthand notation as described in the user guides. A few examples are given here:

Value	Description
nc	signed char
uc	unsigned char
ni	signed int
g	pointer to void (QA C)
p	pointer to void (QA C++)
pX	pointer to X, eg pni is a pointer to int
{uc, ni}	function returning an unsigned char and taking an int.
{sNAME}	structure NAME

When omitted the rule applies to all types. Combinations of these values can be obtained by separating entries with commas or spaces.

Note that the type of an identifier is the actual type, rather than any typedef used. Hence in the following code:

```
typedef unsigned int UINT32;  
UINT32 foo;  
unsigned int bar;
```

both `foo` and `bar` will have the type 'ui' for unsigned int.

Flags

Name: flag

Value: string from list

This restricts the rule in terms of additional information regarding the identifier. This is additional to the previous information and helps

narrow the identifier down. When omitted the rule applies to all types/declarations/access shown.

Value	Description
F	function like macro
O	object-like macro
S	static storage duration
R	function parameter
Additional flags for QA C++ only:	
I	inline declaration
V	virtual declaration
PV	pure virtual declaration
D	POD-class – C-style structure in C++
A	aggregate class
1,2,3	explicitly public, protected, private access
5, 7	implicitly public, private access

For QA C++, the flags field in the define record may hold combinations of these values.

Pattern

Name: pattern

Value: string, regular expression

This specifies the regular expression to match the identifier. When omitted no identifiers will be matched – i.e. they fail the name check. This can be useful to check that other parts of the rule are directed at the desired identifiers.

Invert

Name: invert

Value: Boolean

If specified as true this will invert the logic of the Pattern matching. Typically a Regular Expression can be constructed to perform this negation, however it may be easier (and easier to maintain) to specify what not to match. If omitted then the result of the matches are not inverted.

Message Number

Name: message

Value: number

This specifies the message number to issue if the identifier fails the pattern match. If omitted then the built in message number 4800 will be used.

It is important that messages generated do not clash with other messages, whether from Primary Analysis (1-4999) or any other Secondary Analysis (5000+). If not using the default message (4800), messages generated must be present in the user message file, otherwise they will be displayed as level 99 fatal errors.

Namespace

Name: namespace

Value: string, regular expression

This specifies the namespace within which the rule will be applied. The namespace for an identifier is the fully qualified namespace. Identifiers within nested namespaces will have all the namespace names in their fully qualified name.

Note this only applies to QA C++.

Perl Compatible Regular Expressions

This section is a brief introduction to Perl Compatible Regular Expressions; it is not an exhaustive tutorial. There are many books and online resources for Perl Compatible Regular Expressions (hereafter referred to as just Regular Expressions) and the Reader is encouraged to look into these.

Online Resource: <http://perldoc.perl.org/perlre.html>

Matching Characters

Most of the Regular Expressions in a Naming Convention will specify a character or range of characters to match and how many times to perform the match.

A literal character will match itself only. The period '.' matches any character. A range of characters is specified by enclosing them in

square brackets '['']. For example [A-Z] will match any capital letter. If a selection of letter is needed (rather than a range) then these should be enclosed in square brackets. For example [AMz] will match A, M or z only.

These can be concatenated so as to match sequences of letters. For example [A-Z][a-z] will match any two sequences of letters where the first is upper case and the second lower case.

Matching a Number of Times

There are several ways to specify the number of times to match. Where no number of matches is specified the pattern must match exactly once. The following can be used to specify the number of times to match:

- An asterisk '*' matches zero or more
- A plus sign '+' matches one or more
- A question mark '?' matches zero or one
- Braces '{', '}' are used to match a specific number of times or a range of times

For example '[A-Z][a-z]*' will match any word (including single letter ones) that starts with a capital followed by lower case letters. So 'Speed' and 'V' both match, 'velocity' does not match.

'[AEIOU][a-z]+' matches any 2 letter or longer word starting with a capital vowel followed by lower case letters. So 'Add' and 'Egg' match but 'A' and 'open' do not match.

There can be one or two numbers and a comma in the braces:

- One number means match that exact number of time e.g. '^[a-z]{6}\$' will match a string of six lower case letters.
- Two numbers can specify a range to match e.g. '^[A-Z]{6,8}\$' means match a string of six, seven or eight capital letters.

To match anything of a particular length use the period '.' to match any character e.g. '^.{1,31}\$' will match any strings between 1 and 31 characters.

Anchoring

It is important that Regular Expressions are matched against the entire identifier. The following example should clarify this:

Suppose a rule states that member variables should start with `m_`. The pattern `'m_.*'` seems suitable as it will match both `'m_speed'` and `'m_size'`.

However it will also match `'sim_speed'`. Indeed the pattern `'m_'` also matches this, as the pattern will match anywhere within the identifier name.

To make sure that the expression matches the entire identifier the start must be anchored with a caret `'^'` and the end with a dollar `'$'`. In this particular case the end is not so important as we are matching any character. However, if we wanted all member variables to end with an `'e'`, the revised pattern `'^m_.*e'` would still match `m_speed` as it starts with `'m_'` and has zero or more characters followed by an `'e'`. The correct pattern for this would be `'^m_.*e$'`, so that there can be any number of any characters following the `'m_'`, but there must be an `'e'` at the end.

Alternate Matching

Sometimes more than one pattern may be appropriate for a particular identifier. For example, the naming of local variables in functions may follow some convention, but they may have exceptions for loop variables like `'i'`. Rather than an exception mechanism, the Regular Expression can be constructed to match the identifiers that can be ignored.

The pipe character `'|'` is used to separate two (or more) Regular Expressions. Matching of these is done in turn and once a match is found then the match is deemed true and checking will stop. Using the above examples, the Regular Expressions are:

`'^[ij]${^b_}'` – matches variables named `i` or `j` or beginning `b_`

Escaping Special Characters

Occasionally there may be a requirement to match characters that have a special meaning in a Regular Expression. These special characters are ‘^.\$()*+?{\''. The parenthesis may be needed in a type pattern, as this designates a function or the open square bracket to indicate the type is an array. If these characters are placed in the Regular Expression in an attempt to match them they are interpreted as having a special meaning. Usually this means that the Regular Expression will not compile, or the desired matching will not occur. To interpret these as the literal characters they need to be escaped with a backslash ‘\’. However, this also needs to be escaped as it is used to escape special characters in the configuration file. Hence to specify a left parentheses in a Regular expression use ‘\\(’ and to specify a backslash use ‘\\\\’

Configuration File Example

A simple example file is shown below along with a description for each rule below.

#1 internal functions words separated by _ and begin with capitals

```
rule={ 'space': 'OF', 'linkage': 'I', 'pattern': '^([A-Z][a-z]*) (_[A-Z][a-z]*)*$', 'message': 4800 }
```

#2 block scope variables (except i and j) must start b_

```
rule={ 'space': 'OV', 'scope': 'B', 'linkage': 'N', 'pattern': '^([ij])$|^([b_].*)$', 'message': 4800 }
```

#3 macros must be all capitals, underscores allowed

```
rule={ 'space': 'OM', 'pattern': '^([A-Z_]+)$', 'message': 4800 }
```

#4 all functions and variables must be 31 or less chars

```
rule={ 'space': 'OF,OV', 'pattern': '^.{1,31}$', 'message': 4800 }
```

Getting Feedback on Errors

If there are any problems then these are logged in a text file called ‘NameCheckErrors.txt’. It will be written in the output directory. If a project has more than one output directory the file will appear in each

output directory (and will be identical in each one) for the corresponding source file the Name Check was run on.

Configuration File

If the configuration file cannot be found the log file will display the name of the file as specified in the Message Personality. Check that the correct file is specified and that it has the correct access permissions set. If there are spaces in the path, make sure that the entire filename is enclosed in quotes.

JSON Syntax Errors

All JSON error messages start with the line number of the configuration file that the error occurred on. Any syntax errors in the JSON will be reported with the words 'Error: option parsing;' followed by an explanation of the problem and its' location.

The value of 'invert' is a Boolean value so must be either the literal text true or false. The value of 'message' is a number. Quotes must not be used for either of these values, as this would make them string values.

Unknown Name

This occurs when a Rule Name other than those in section 5.2.2 is used. Note that these must all be lower case. For example {'Scope':'I'} will produce the following message:

```
Line 1 Unknown object member name 'Scope' with  
      string value I  
      { 'Scope': 'I' }
```

Change the capital 'S' to lower case to correct the error.

Unknown value

This happens when the value for an object is not one of the permissible options. Note that objects that take a Regular Expression (filename, pattern, type) can never have this error. For example {'linkage':'E'} will produce the following message:

Line 2 Unknown value 'E'

In this case the intention is probably external linkage, whose value is 'X', rather than 'E'.

Regular Expressions

The Regular Expressions for filename, pattern and type are compiled as the configuration file is processed. If they contain errors then these are reported. For example:

```
Line 29 Pattern Regular expression '[:ower:]*'
      compile fail: bad class at offset 3
      {'pattern': '[:ower:]*'}
```

The offset shows where in the regular expression the problem is. In some cases problems (like mismatched brackets) are reported at the end of the string.

INDEX

#

#define, 22, 131
 #error, 101, 206
 #pragma, 30, 140, 153, 213

.

.err, .met, and .i files. see File extensions
 .extension. see File extensions
 .met file. see Metric output file

A

Akiyama's Criterion (STAKI), 173
 alignment. See Data types
 Analyser Personality, 24, 124
 Analysing files, 36
 Analysis Log, 49
 Annotated source code, 40, 50
 format, 142, 148
 generating, 104
 settings, 33, 41, 57
 Application options, 33, 37
 Average Size of Function Statements (STAVx), 173

B

Batch files, 96, 97
 Brace style, 60, 156
 Bug Estimate, path-based (STPBG), 184
 Bug Estimate, token-based (STBUG), 193

C

C++, 134
 Calculating metrics, 172
 character sets, 133
 Close Name Analysis report, 72
 CMA. see Cross-module analysis
 COCOMO

Cost Model, 193
 reports, 73
 Code Mobility (STMOB), 200
 Code structure, 77
 Coding standards, 84
 Command line, 96
 Command line options, 97
 -align, 127, 152
 -bitsigned, 129
 -cmaf, 47
 -comment, 130
 -define, 131
 -echo, 132
 -encoding, 133
 -expandmultihomedmessages, 132
 -extensions, 133
 -file, 135
 -foldplainchar, 137
 -forceinclude, 135
 -format, 136
 -hdrs suppress, 138
 -help, 137
 -hiddenmessage, 138
 -hiddenwarnings, 139
 -html, 139
 -include, 140
 -indentlevel, 140
 -intrinsictype, 141
 -k+r, 141
 -line, 142
 -list, 143
 -maxcount, 144
 -maxerrors, 100, 144
 -maxlinelength, 145
 -messagesonly, 144, 147
 -metrics, 145
 -namelength, 147
 -namerulefile, 147
 -nomsg, 146
 -nosort, 147
 -onelineonly, 148
 -outputpath, 105, 135, 143, 149
 -pfilename, 149
 -pplist, 149
 -ppmetrics, 150
 -quiet, 138, 151
 -references, 151
 -remark, 151
 -settings, 153

- size, 128, 152
- skippragma, 153
- slashwhite, 153
- standard, 156
- strictsignedness, 155
- style, 156
- summary, 145, 158
- suppresslvl, 157
- tablesummary, 160
- tabstop, 158
- text, 158
- threshold, 159
- total, 159
- unsignedchar, 160
- usrfile, 161
- usrpath, 161
- version, 162
- via, 162
- warncall, 162
- xcase, 163
- xnamelength, 163
- Comment to code ratio (STCDN), 130, 193, 198
- Comments, 194
- Compiler Personality, 20, 125
- Compilers
 - extensions, 22
 - implementation defined types, 21
- Configuration, 19, 121
 - Project Configuration File, 10, 19, 31, 47
- Context messages
 - information-based, 53, 57
 - location-based, 53, 57
- Cross-module analysis, 45, 46, 106, 172
 - command line, 102
 - configuring, 46, 48
- Custom reports, 118
- Customising
 - creating new reports, 118
 - messages, 110
 - user message file, 109
 - warning calls, 162
- Cyclomatic Complexity (STCYC), 64, 82, 90, 92, 175, 205
- Cyclomatic Complexity Across Project (STCYA), 205

D

- Data types
 - alignment, 127
 - bit-field interpretation, 129
 - char interpretation, 160
 - intrinsic types, 21, 141
 - size, 152
 - unsigned char, 137
- Demographics, 90
 - creating a demograph file, 89
 - exporting data, 93
 - finding functions, 92
 - printing, 93

E

- Embedded Programmer Months (STBME), 74, 193
- Embedded Total Months (STTDE), 75, 203
- Environment variables, 96, 116
 - QACBIN, 96, 116, 161
 - QACHELPFILES, 96, 106, 116, 161
 - QACOUTPATH, 97, 100, 102, 105, 116, 118, 135, 149
- errdsp.exe, errwrt.exe. see Program files
- Essential Cyclomatic Complexity (STM07), 180
- Estimated Development, programmer days (STDEV), 194
- Estimated function coupling (STFCO), 196
- Estimated Porting Time (STPRT), 201
- exdentend, brace style, 60
- Exporting metrics data, 89
- External Name Cross-Reference report, 73

F

- File extensions
 - .err, 20, 32, 34, 37, 38, 41, 46, 51, 57, 97, 102, 104, 105, 117
 - .i, 34, 37, 39, 149
 - .met, 20, 32, 34, 37, 38, 41, 46, 84, 97, 119, 145, 208
 - .prj, 10, 12
 - .txt and .html, 33, 37, 40

- source files, 11
- filelist.lst. *see* Files
- Files
 - filelist.lst, 15, 47, 48, 103, 119, 143
 - housekeeping, 34
 - qac.cfg, 96, 194, 201
 - qac.exe, 101
 - qac.msg, 96, 104, 107, 161
 - qacmet.txt, 88, 94
 - settings.via, 15, 99, 119
- Folder parameters, 10, 19
 - output file path, 11, 12
 - force-create, 13
 - personalities, 11, 12
 - root folder, 11, 12
- Function structure diagram, 77, 80, 164
 - break, 169
 - continue, 170
 - for, 167
 - if, 165
 - if-else, 166
 - if-else inside a loop, 168
 - knot, 169, 170
 - return, 170
 - straight code, 165
 - switch, 166
 - unreachable code, 171
 - while, 167
- Functions
 - calls, 77, 208
 - control graph records, 212
 - external, 209
 - metrics, 86
 - Relationship graphs, 77
 - warning calls, 162

H

- Halstead Distinct Operands (STOPN), 196, 200
- Halstead Distinct Operators (STOPT), 196, 200
- Halstead Prediction of STTOT (STHAL), 196
- Hard errors, 100, 206
- Header files, 77, 97, 98, 99, 100, 116, 140
 - messages, 54, 55
 - preprocessed output, 39
 - project, 24
 - suppressing, 41, 105, 138

- system headers, 20

I

- Identifier Declarations report, 72
- ignore, 23, 131
- implicit cast, 61
- include files. *see* Header files
- indentation, 60
- internal identifiers, 147
- intrinsic types, 21

K

- K&R, 141
- Kiviat diagram, 90, 93, 94
 - exporting data, 95
- Knot, 169, 170
- Knot Count (STKNT), 178
- Knot Density (STKDN), 177

L

- Language extensions, 131, 133
- Licensing
 - auto-release, 35
 - return codes, 206

M

- Macros, 24, 25, 97, 100
 - ignore, 23
 - project, 131
 - system, 21
- Maximum nesting of control structures (STMIF), 90, 183
- Message Browser, 34, 51, 96, 97, 102, 106
 - annotated source, 52
 - context messages, 53
 - message listing, 56
 - Summary view, 55
- Message file, 96, 107
 - #define, 107, 108
 - #levelname, 108
 - configuring in message personality, 25
 - duplicate messages, 109, 110
 - groups, 107, 108, 121

- levels, 26, 58, 70, 108, 121, 122,
 - 156, 158, 160
 - changing, 110
 - renaming, 111
- message records, 108
- multi-homed messages, 56, 57, 132
- reference text, 109
- rewording, 110
- user message file, 58, 84, 107, 109, 161
- Message Personality, 25, 34, 58, 109, 121
- Metric output file, 145
 - control graph records, 212
 - define records, 210, 214
 - external reference records, 209
 - literal records, 214
 - metrics records, 84, 213
 - pragma records, 213
 - relationship records, 208
 - relationship type records, 209
- Metric Types
 - STAKI, Akiyama's Criterion, 173
 - STAVx, Average Size of Function Statements, 173
 - STBAK, Backward Jumps, 174
 - STBME, Embedded Programmer Months, 74
 - STBMO, Organic Programmer Months, 74, 193
 - STBMS, Semi-detached Programmer Months, 74, 193
 - STBUG, Bug Estimate, token-based, 193
 - STCAL, Number of Distinct Function Calls, 174
 - STCDN, Comment to code ratio, 84, 130, 193, 198
 - STCYA, Cyclomatic Complexity Across Project, 205
 - STCYC, Cyclomatic Complexity, 64, 82, 90, 92, 175, 205
 - STDEV, Development time, 194
 - STDIF, Program Difficulty, 195
 - STECT, External Variables, 195
 - STEFF, Program Effort, 195
 - STELF, Dangling else-if's, 176
 - STFCO, Function Coupling, 196
 - STFN1, Number of Function Operator Occurrences, 177
 - STFN2, Number of Function Operand Occurrences, 177
 - STFNC, Function Definitions, 195, 205
 - STGTO, Number of goto's, 177
 - STHAL, Halstead prediction of STTOT, 196
 - STKDN, Knot density, 177
 - STKNT, Knot count, 178
 - STLCT, Local variables declared, 178
 - STLIN, Maintainable code lines, 179
 - STLOP, Logical Operators, 179
 - STM07, Essential Cyclomatic Complexity, 180
 - STM19, Number of Exit Points, 181
 - STM20, Number of Operand Occurrences, 197
 - STM21, Number of Operator Occurrences, 197
 - STM22, Number of Statements, 198
 - STM28, Number of Non-Header Comments, 198
 - STM29, Number of Functions Calling this Function, 181
 - STM33, Number of Internal Comments, 199
 - STMCC, Myer's Interval, 182
 - STMIF, Maximum nesting of control structures, 90, 183
 - STMOB, Code mobility, 200
 - STNEA, Number of Entry Points Across Project, 205
 - STNFA, Number of Functions Across Project, 205
 - STNRA, Number of Recursions Across Project, 205
 - STOPN, Halstead distinct operands, 196, 200
 - STOPT, Halstead distinct operators, 196, 200
 - STPAR, Number of Function Parameters, 183
 - STPBG, Bug Estimate, path-based, 184
 - STPDN, Path Density, 184
 - STPRT, Porting time, 201
 - STPTH, Static path count, 64, 90, 184
 - STRET, Number of Function Return Points, 186
 - STSCT, Static variables, 202
 - STSHN, Shannon information content, 202

STSTx, Number of Statements in Function, 187
 STSUB, Number of function calls, 188
 STTDE, Embedded Total Months, 75, 203
 STTDO, Organic Total Months, 75, 203
 STTDS, Semi-detached Total Months, 75, 203
 STTLN, Preprocessed source lines, 203
 STTOT, Total number of tokens, 203
 STTPP, Unpreprocessed source lines, 204
 STUNR, Number of Unreachable Statements, 189
 STUNV, Unused and Non-Reused Variables, 190
 STVAR, Number of identifiers, 204
 STVOL, Program Volume, 204
 STXLN, Executable lines, 190
 STZIP, Zipf prediction of STTOT, 196, 204
 Metrics, 83
 Browser, 86
 calculation of, 172
 creating a demograph file, 89
 displaying, 86
 exporting data, 89, 90, 93
 file-based, 192
 filtering, 88
 function-based, 172
 Kiviat diagram, 90, 93
 plotting, 87
 project-wide, 204
 selecting files, 83
 thresholds, 83
 multi-homed messages. *see* Message file
 Myer's Interval (STMCC), 182

N

Naming convention checking, 43, 147, 236
 Number of Backward Jumps (STBAK), 174
 Number of Code Lines (STLIN), 179
 Number of Dangling else-ifs (STELF), 176

Number of Distinct Function Calls (STCAL), 174
 Number of Entry Points Across Project (STNEA), 205
 Number of Executable Lines (STXLN), 190
 Number of Exit Points (STM19), 181
 Number of extern Variables Declared (STECT), 195
 Number of function calls (STSUB), 188
 Number of function definitions (STFNC), 195, 205
 Number of Function Operand Occurrences (STFN2), 177
 Number of Function Operator Occurrences (STFN1), 177
 Number of Function Parameters (STPAR), 183
 Number of Function Return Points (STRET), 186
 Number of Functions Across Project (STNFA), 205
 Number of Functions Calling this Function (STM29), 181
 Number of goto's (STGTO), 177
 Number of Identifiers (STVAR), 204
 Number of Internal Comments (STM33), 199
 Number of Local Variables (STLCT), 178
 Number of Logical Operators (STLOP), 179
 Number of Non-Header Comments (STM28), 198
 Number of Operand Occurrences (STM20), 197
 Number of Operator Occurrences (STM21), 197
 Number of Recursions Across Project (STNRA), 205
 Number of Statements (STM22), 198
 Number of Statements in Function (STSTx), 187
 Number of Static Variables (STSCT), 202
 Number of Unreachable Statements (STUNR), 189
 Number of Unused and Non-Reused Variables (STUNV), 190

O

Organic Programmer Months (STBMO),
74, 193
Organic Total Months (STTDO), 75,
203

P

Path Density (STPDN), 184
Personalities, 19, 121
 Analyser, 24, 124
 Compiler, 20, 125
 Message, 25, 34, 58, 109, 121
Personality files, 98
Pragma directives, 153, 213
 PRQA_MACRO_MESSAGES, 31
 PRQA_MESSAGES_ON and OFF,
 30
 PRQA_NO_RETURN, 31
 PRQA_NO_SIDE_EFFECTS, 31
preprocessed source code, 39, 149
Printing
 demographics, 93
 Function structure, 82
prjdsp.exe. see Program files
Program Difficulty (STDIF), 195
Program Effort (STEFF), 195
Program files
 errdsp.exe, 96, 97, 98, 104
 errsum.exe, 96, 103
 errwrt.exe, 101, 111, 117
 prjdsp.exe, 96, 97, 105, 119, 120
 qac.exe, 96, 97, 98, 100, 101, 104
 viewer.exe, 102, 120
Program Volume (STVOL), 204
Programming standards. see Coding
 Standards
Project Configuration File, 10, 19, 31,
47
Project metrics reports, 76
Projects
 Auto-create, 10, 11
 clean-up, 12
 creating, 10
 environment variables, 14
 relative paths, 14
 root path, 14
PRQA_MESSAGES_ON and OFF, 139

Q

qac.exe. see Program files
qac.msg. see Files

R

r_basename, 216
r_close, 216
r_fields, 217
r_grep, 218
r_sort, 219
r_uniq, 219
rank, 155
Relationship graphs, 77
Relationships, 77
 calls, 77, 208
 includes, 77, 208
 printing, 79
 records in .met file, 209
 Refers-to, 77
 saving, 80
Reports, 69
 Close Name Analysis, 72
 COCOMO Cost model, 73
 External Name Cross-Reference, 73
 Identifier Declarations, 72
 Project Metrics, 76
 Project Warning Summary, 70, 103
 Warning Listing, 34, 40, 41, 71, 96,
 105, 139, 143, 149
Return codes, 100
 licensing, 101

S

scripts, 42
Secondary Analysis, 38, 116
 configuring, 42
 running, 101
 writing new messages, 117
Secondary Analysis, 42
Semi-detached Programmer Months
 (STBMS), 74, 193
Semi-detached Total Months (STTDS),
75, 203
Shannon Information Content (STSHN),
202
Static Path Count (STPTH), 64, 90, 184

Suppressing warning messages, 32,
58, 122, 140, 146
Suppressions, 106, 220
 annotations, 220
 continuous, 223
 EOF tag, 222
 location tags, 220
Syntax errors, 52, 56, 123

T

Tab spacing, 25, 100
Total Number of Tokens Used
 (STTOT), 203
Total Preprocessed Code Lines
 (STTLN), 203
Total Unpreprocessed Source Lines
 (STTTP), 204

U

Unreachable Code, 171
Utilities
 r_basename, 216
 r_close, 216
 r_fields, 217
 r_grep, 218
 r_sort, 219
 r_uniq, 219

V

Viewing
 Demographics, 90
 Function structure, 80
 Kiviat diagram, 94
 Metrics, 83
 Relationships, 77

W

Warning calls, 162
Warning Listing report, 34, 40, 41, 71,
96, 105, 139, 143, 149
Warning messages
 escape sequences, 109
 format in annotated source code,
142, 148
 summary, 70
Warning summary report, 70

Z

Zipf Prediction of STTOT (STZIP), 196,
204