PRQA
Programming Research

QA·C 8.1 SOURCE CODE ANALYZER

USER GUIDE FOR WINDOWS

May 2013

This document describes functionality and GUI of the QA·C Source Code Analyzer application for Windows.

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

TRADEMARKS

CONTACTING PROGRAMMING RESEARCH LTD

For technical support, contact your nearest Programming Research Ltd authorized 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
www.programmingresearch.com

PRQA Programming Research

Table of Contents

CHAPTER 1

Introduction to QA·C

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

QA·C analyzes source code on a file-by-file and complete project basis to identify dangerous usage of the C language. A library of over 1300 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 also identifies language usage which is not compliant with the ISO C90 standard (ISO/IEC 9899:1990) or relies on unspecified, undefined or implementation-defined behavior. The tool warns when features specific to the ISO C99 standard are used, but does support semantics of many of these features.

Warning messages for the source code of a complete project are displayed through a Message Browser that categorizes and groups message occurrences across all source files.

Other features of the tool include:

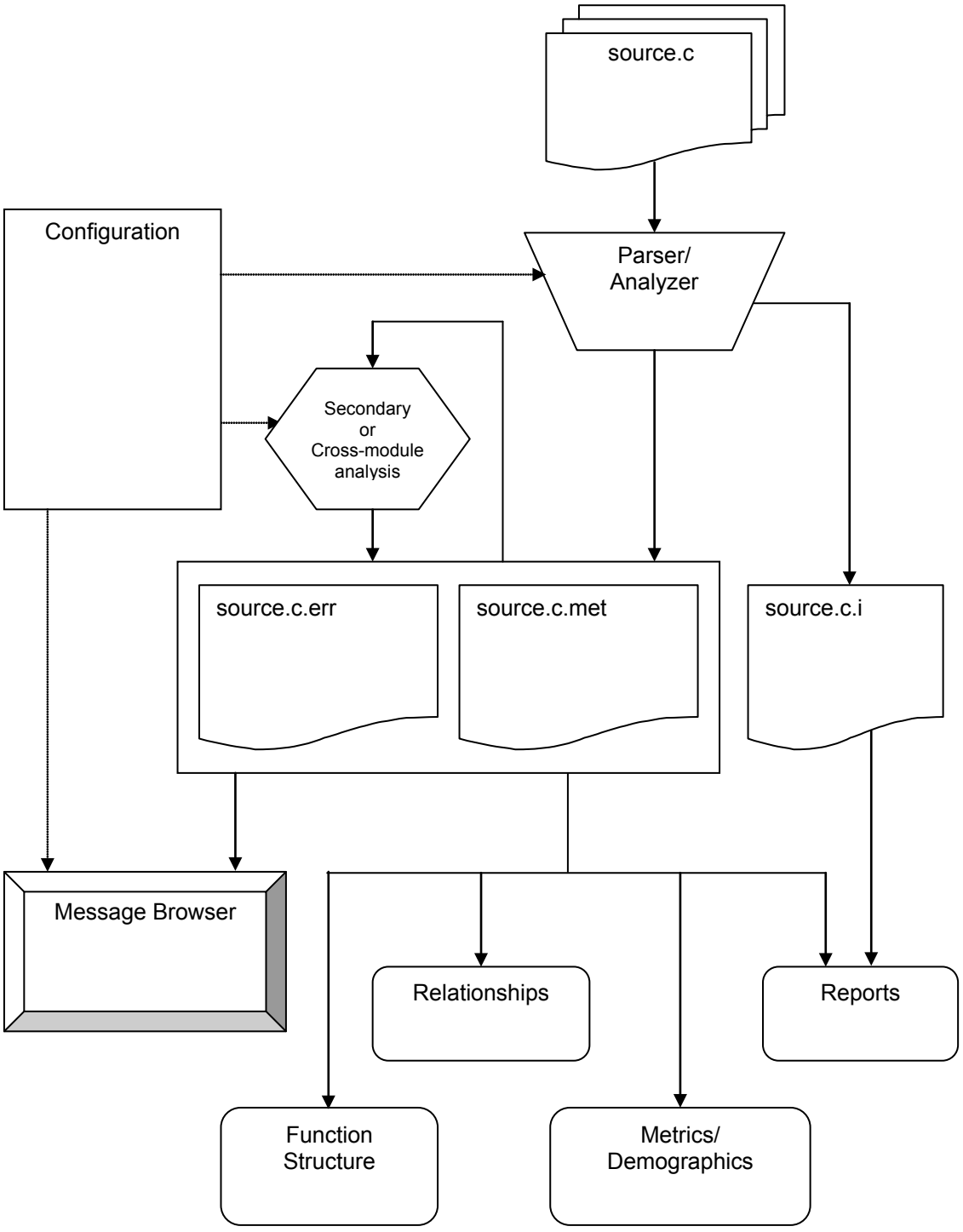
- code metrics which provide a quantifiable measure of code attributes: 33 function-based, 30 file-based, and 4 at project-level;
- function structure diagrams to provide an insight into control flow;
- relationship diagrams to demonstrate function calling, global referencing, and include file trees;
- demographic analysis to provide an overall assessment of code quality against industry benchmarks;
- cross-module analysis (CMA) capability, analyzing function recursion and various global identifier issues;
- a baselining module to simplify modification of legacy code.

QA·C Action Overview

QA·C performs 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 checks which are industry-specific, company-specific, or standards-specific. Secondary analysis is also used by the baselining process. Cross-module analysis (CMA) covers those checks that need to operate across translation units.

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

The schematic diagram 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. Typically, 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, these folders should not share the same output path, since the successive analyzes 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 customization. See Chapter 3: Project Configuration File for more details.

A new project can be created automatically or manually.

Opening a QA·C 8.0 Project

GUI Projects for QA·C 8.0 are compatible with the QA·C 8.1 GUI, with the exception of dataflow configuration.

QA·C 8.0 Projects store Dataflow Analysis settings in a Project Configuration File (.pcf). QA·C 8.1 projects store Dataflow Analysis settings in a Dataflow Personality file (.p_d). Dataflow settings cannot exist in both file locations concurrently.

The GUI will detect situations where Dataflow settings are detected in more than one location, in order to prevent the user from creating an illegal configuration. An error is displayed in the following situations:

1) When a QA·C 8.0 project with Dataflow settings configured in a Project Configuration File (.pcf) is loaded into the QAC 8.1 GUI (see below for how to upgrade an 8.0 project).
2) When a GUI project has a Dataflow Personality File configured and the User attempts to select a PCF that contains Dataflow settings.

Upgrading an 8.0 Project

Upon loading a QA·C 8.0 project with dataflow configured in a PCF file into the QA·C 8.1 GUI, a conversion for the project is offered. This conversion will read through an attached Project Configuration File and search for dataflow settings. Any files included using the -via or through environment variables will also be searched, and any files found to contain dataflow settings will be modified with the dataflow settings removed. Any detected dataflow settings will be added into a new Dataflow Personality File. The new Dataflow Personality File will have Dataflow enabled by default, and will be stored in the same location as the original Project Configuration File.

Using an old Project Configuration File

If a Project Configuration File contains dataflow settings, the User will not be allowed to add that file to the Project. First remove the Dataflow settings from the PCF, and then attach the updated PCF file to the project.

Creating a Project Automatically

When creating a new project automatically, QA·C starts from a specified directory and uses its sub-directories and files. The resulting structure reflects that 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. It will also be 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 to be associated with the root folder of your project.
4. Enter an *Output File Path.* This is the directory in which QA·C generates output files during analysis. It is good practice to use a dedicated directory associated with your project.
5. *Replicate source tree structure in output paths* is used to create a sub-directory structure for output file path(s). You have two options for this. You can choose *Parallel to Source Structure* to create a directory structure in parallel with the source code, or *Sub-path to each source location* to embed the specified output path subdirectories beneath each source directory. Either option will avoid problems caused by the same-named files from different source directories overwriting their analysis output files.
6. Select the *File Extensions* of the files that you want to add to your project. QA·C will add all the sub-folders beneath your Starting Directory that contain files with the specified file extensions.

   Usually you will want to select only source code files (.c). The location of included files (.h) will be specified later.
7. Select the personalities for the folder. You may want to use the default values as a starting point. Personalities can be edited later by selecting them from the *Configuration* menu. See Chapter 3: Configuring QA·C.
8. Click *OK* to create your project. The project structure should contain the specified source files and sub-folders.
9. Save the project. Typically, 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

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

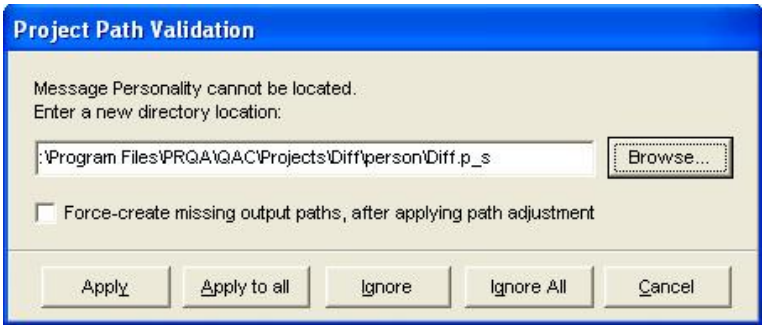
1. Choose *New Project* from the *File* menu. The *New Project Parameters* dialog box appears.
2. Enter the *Root Folder Name*, which will be the name of your project.
3. Enter the *Default Source Path* for the project's initial folder. This path can be changed for each sub-folder.
4. In *Output File Path*, enter the directory to which QA·C writes output files during analysis.
5. Select personalities for the folder.
6. Click *OK* to create the project. Your project is now configured with a single folder.
7. Add further sub-folders and files as required.

8. Save the project. Typically, projects are saved with the .prj file extension.

Reopening Projects

With the *File* > *Reopen* menu item, you can select from up to ten recently-opened projects. Any projects which are no longer accessible will appear greyed-out. For such cases, there is a menu item, *Clean-up*, to remove all unavailable projects in the 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 be applied automatically.



1. First, make a correction to the entry, either by using the *Browse* button or by entering a corrected location into the edit box.
2. Click *Apply* to make the 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 ticked, output paths are force-created after making any necessary path adjustment. This can be useful for projects extracted from a Version Control System, where the output directories were not checked in, and therefore will 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.

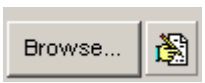
Project Folders

Each folder of a project contains parameters, consisting of the folder name, the default source path, the output path, and three personalities.

You can change these through the *Edit* > *Folder Parameters* menu item. Changes to a folder's parameters will only apply to that folder. To apply your parameter changes to sub-folders as well, use *Edit* > *Propagate Changes to Sub-Folders.*

Alternatively, *Configuration* > *[Message* | *Analyzer* | *Compiler] Personalities…* gives a list of all personalities on your system and the selections made for each project folder. You can change or edit any personality from this screen.

You can use the speed button to edit a personality. The speed button is the icon to the right of the Browse button:



Notes:
If you create and save a new personality, you must select it in order to apply it to the current folder parameters.

Ctrl-F is an alternative way of displaying the Folder Parameters dialog.

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 file or a group of files is selected, these form the selection;

b) otherwise the current selected folder, plus all its sub-folders, form the selection. This means all constituent files in these folders are included.

From within the metrics, structure and relationship browsers, it is possible to alter this starting selection of files, using *File* > *Select Files...*

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 the 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 configured as part of the project and located within the directory branch defined by the project file location (such that it is subject to relative path reduction).
- In all analysis and display processes, any path reductions in projects are converted to their fully-qualified paths. This means 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 optimize 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. It 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 it is possible to apply relative path entries manually to the existing (saved) projects, it is recommended to apply relative paths and environment variables only through a project-saving

operation. This will automatically include the adjustment to associated personalities.

GUI-based Environment Variable Creation

You can create new environment variables (EVs) through the graphical user interface.

Application-level environment variables are created using the *Environment* tab of the *Configuration* > *Options* menu item. Application-level EVs override any inherited global EVs, and will be alerted through a dialog message.

Project-level EVs are created using the *Project* > *Properties* menu item. Project-level EVs override both global EVs and application-level EVs.

All EVs created through the GUI can be used in any secondary analysis or cross-module analysis, or in creating Custom Reports.

Note:

Application-level EVs are saved into the [EnvVars] section of the qac.ini file.

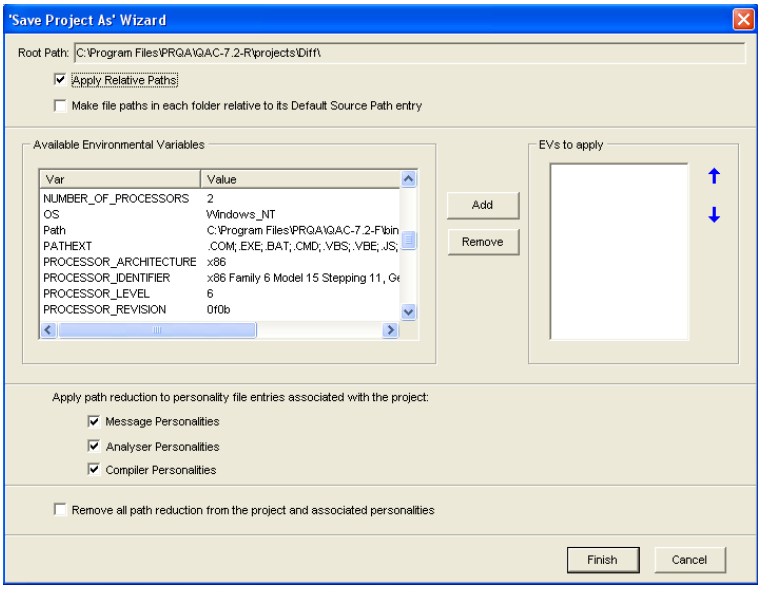
Project-level EVs are saved into the project file, near the top, below the “Project Configuration File” entry.

Applying Relative Paths 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.

When you select *File* > *Save As*… with an existing saved project, you will be prompted for a new name, and then the “Save Project As” wizard will appear.

A dialog window presents various options for the application of relative paths and environment variables.



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

1. 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.

2. 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.
3. 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.

4. 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.
5. 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.

Notes:

A project already containing relative paths will only be re-saved as a full-path project. This is necessary because of the difficulties of managing relative paths in associated personality files. It can later be saved with relative paths in its current or new location.

New entries added to a project – new source files, or path-based entries in personalities – will always be added with full paths. To apply relative paths to the entire project will then require saving once with all entries as full paths, and then saving again, this time applying all Relative Path and Environment Variable settings.

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

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

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, a project configuration file, and a dataflow configuration file. A personality is a group of configuration options that specifies how QA·C analyzes 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. There can be a single dataflow configuration file for each project.

Personalities can be defined differently for each folder in the project. QA·C analyzes the 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.
- Analyzer Personality, which defines analysis options that are associated with your project.
- Message Personality, which defines the enabled messages and how output is displayed.

QA·C is supplied with some default personalities, but you may prefer to adapt these for your company, software project, or individual use.

To view the set of personalities in your installation along with those defined in your project, choose *Configuration* > *[Message* | *Analyzer* | *Compiler] Personalities...* For each of these, you will be presented with the complete list of available personalities, including those supplied in add-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 are used by QA·C to match the behavior of your compiler. See Appendix A: Personalities for a list of all the Compiler Personality settings.

Note:
When amending personality information, make sure that you are altering the file currently being used in your project folder settings, rather than one 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. A further set of include paths can be specified in *Project Headers* of the Analyzer Personality. Both sets of include paths will be searched. It is usually appropriate to specify paths for files associated with the project in the Analyzer Personality and paths associated with system or compiler files in the Compiler Personality.

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: Analyzing Source Code.

Using a text editor you can enter the path relative to the current project location in the personality file, although it is recommended to make full path selections and then apply any chosen path reduction during the project save operation. See Applying Relative Paths and Environment Variables for more details.

Note:
The ordering of files in the lists is important. QA·C will search the directories specified in the order they are listed.

When you create a new Compiler Personality, it will include, by default, the path to the special abbreviated analysis versions of the standard ISO C header files. They are located in the include\ansi directory.

Setting System Macros

In *System Macro Defines* on the *System Macros* tab, set up those macros which are required by your compiler or development environment. Macros can be defined in either the Compiler Personality or the Analyzer Personality.

There are three ways to define a macro:

IDENT	equivalent to	#define IDENT 1
IDENT=value	equivalent to	#define IDENT value
IDENT=	equivalent to	#define IDENT

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:

size_t	An unsigned integral type that reflects the type returned by the sizeof operator.
ptrdiff_t	A signed integral type that reflects the type resulting from the subtraction of two pointers.
wchar_t	An integral type whose range of values can accommodate all the extended (wide) characters supported by the compiler.

Intrinsic Types on the *Data Types* tab control 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) 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.

A set of standard library header files is shipped with QA·C. If you make use of these stub headers, you will not need to be concerned about intrinsic type settings. These headers 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 which correspond to the intrinsic type settings in the Compiler Personality.

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. 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 it is not usually able to interpret the extensions semantically. Usually, the tool must be configured so that non-standard keywords are ignored.

There are several 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 statement 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 handles the keyword in one of several ways.

For example:

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

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 recognized 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.

Additional Include Files

Macros can be defined by incorporating them into a personality. If you want to define typedefs, they will need to be specified within your source code. However, rather than introducing them directly into your source code or header files, usually it is 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 of the Compiler Personality.

The run-time absence of a specified force-include file will be flagged as a Level 9 error.

Identifiers

Internal Identifier Significant Length

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 that are identical in more than 31 characters but it is not advisable make use of this feature. Programming standards may choose to restrict the declaration of identifiers so that they are distinct within a smaller number of characters.

QA·C will generate message number 779 if it encounters identifiers that are not distinct within the number of characters you have specified.

External Identifier Significant Length

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 if it encounters identifiers that are not distinct within the number of characters you have specified.

Ignore External Identifier Case

When ticked, this option implements the ISO standard which specifies that alphabetic case should not be treated as significant in external identifiers. This option only applies to the lengths that you have specified in **Internal Identifier Significant Length** and **External Identifier Significant Length**.

Configuring Your Analyzer Personality

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

Note:
Use the Folder Parameters (Ctrl-F to display the dialog box) to ensure that you are altering the personality file currently being used in your project folder settings.

Project Header Files

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

The ordering of files in the lists is important. QA·C will search the directories listed in the order specified.

Using a text editor you can enter the path relative to the current project location in the personality file, although it is recommended to make full path selections and then apply any chosen path reduction during the project save operation. See Applying Relative Paths 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: Analyzing Source Code.

Project Macros

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

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 a different thing than the indentation. It defines the number of character positions associated with a single tab character.

Warning Calls

A **warning call** is a diagnostic produced in the analysis when a call is encountered to a specified function.

Press **Add…** to insert a warning call entry. There are two entry formats:

1. function_name=msg_number will display msg_number every time function_name is called.
2. function_name will display the default warning:

```
++ WARNING ++: <=4=(2010) 'function_name' must not be called.
```

Pragma Blocks

#pragma statements are pre-processor directives that are specific to a particular compiler. When QA·C encounters a #pragma it will generate warning message 3116 if it has not been declared. You can declare a #pragma in two ways:

- You can declare a single #pragma identifier. If you do this QA·C will ignore all occurrences of that #pragma.

- You can declare a #pragma block by entering two identifiers separated by a comma. If you do this QA·C will ignore both #pragma directives and all source code in between.

For example:

```
#pragma startasm
...
...
#pragma endasm
```

This pragma block would be declared by entering startasm,endasm.

QA·C will also recognise the wildcard character * in #pragma declarations. For example, entering ABC* will match all #pragma directives that begin with ABC.

Note:

QA·C also implements a set of special message suppression #pragma directives to cover a range of message generation situations.

K&R Compatibility

These options represent a number of constructs which are valid in ISO standard C but which would not necessarily be recognized by a K&R compiler. If you wish 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.

Ticking a checkbox will cause QA·C to 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, you can leave all the checkboxes blank.

Clicking the K&R button will enable a selection of settings that will enforce compatibility with the Sun K&R cc compiler under SunOS 4.1 or earlier.

Configuring Your Message Personality

Because of the large number of diagnostic checks performed by QA·C, diagnostic output may be very large. Some of the messages generated may not be applicable to your project and environment. It is therefore important to be able to show diagnostic output 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. However, 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:

Use the Folder Parameters (Ctrl-F to display the dialog box) to ensure that you are altering the personality file currently being used in your project folder settings.

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.

Notes:

Messages at Level 9 describe configuration, syntax and constraint errors. Level 9 messages cannot be suppressed.

Suppressed message levels will appear with a red **X**. No messages from these levels will appear.

It is not necessary to re-analyze files in order to view changes to the configuration of enabled levels.

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 organized 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. The standard message file can be enhanced with your own user message file. Following is an explanation for each level and group in the standard message file.

Level 0: Information

These are information level warnings with the lowest level of importance.

Annotations	Syntax problems in annotations (used in code-based suppressions).
CMA information	Setup problems with cross-module analysis.
Parsing recovery	Informational messages generated in the course of recovering from a parsing error.
Dataflow recovery	Informational messages generated in the course of recovering from an internal dataflow analysis failure.
Sub-Messages	Additional context sub-messages giving clarification to a primary message.

Level 1: Obsolete Messages

Messages marked as obsolete are kept for backwards compatibility for 2 product releases, and will be subsequently removed.

Level 2: Minor

Level 2 messages identify issues which may infringe coding standard requirements, but which are not necessarily serious errors. Level 2 message groups are associated with different aspects of the C language.

Level 3: Major

Major messages are those 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. Level 3 message groups are associated with different aspects of the C language.

Level 4: Local Standards

Level 4 messages identify 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: Dataflow Analysis

Messages in this level perform analysis of run-time behavior, identifying problems ranging from serious issues such as undefined behavior to conditions which are frequently associated with coding and logic errors.

Conversion to signed	Conversions to a signed type which is not large enough to represent the expected values.
Conversion to unsigned	Conversions to an unsigned type of values which are either negative or too large to be represented.
Shift operations	Shift operations which have an undefined result or which result in the loss of non-zero bits.
Overflow and wraparound	Operations which result in signed arithmetic overflow (undefined behavior) or unsigned arithmetic wraparound.
Arrays	Operations on objects of array or pointer type which result in undefined behavior.
Pointers	Comparison or subtraction of pointers which address different objects.
NULL pointers	Arithmetic or dereferencing operations on NULL pointers.
Unset data	Accessing the value of unset data.
Redundancy	Redundant initializations, expressions or assignments, often associated with faulty logic.
Invariant operations	Expressions which always evaluate as true or false, frequently associated with faulty logic.
Control flow	Unreachable code, infinite loops, incorrect return statements.

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 Behavior

CMA Undefined Behavior	Constructs whose behavior is explicitly described as undefined in the language standard and which have been identified by using cross-module analysis.
Explicitly undefined	Constructs (not classified as dataflow dependant) whose behavior is explicitly described as “undefined” in the language standard.
Implicitly undefined	Constructs whose behavior is not described in the language standard and whose behavior is therefore undefined by implication (only).

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 #pragma message suppression, which is deprecated. There is a simple and intuitive syntax for entering suppressions. 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. This string should be given as shown, in upper case.

There is a full explanation of suppression syntax in Appendix H: Code-based Suppressions. 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 recognizes several #pragma identifiers in the source code which can be used to control the display of messages, but this usage is deprecated in favour of using comment-based suppressions (see Appendix H).

For example:

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

will suppress the specified messages for the remainder of the current translation unit (TU). 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.

Level 9 messages cannot be suppressed, as messages from this level can affect the quality and scope of analysis. They 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 *File* > *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 these analysis and reporting functions.

If a Project Configuration File is defined in the project file, its location is specified with the ProjectConfigFile entry, which is placed before the StartProjectMarker.

It is possible to use this file to make project-wide configuration entries, which will override any specific configuration in personality files. This can be done by manually creating the appropriate entries in the project configuration file.

When QA·C is extended to match a project 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.

Dataflow Analysis Settings

Deep Flow Dataflow analysis (DF^2) is a sophisticated additional phase of analysis, and fully described in Deep Flow Dataflow Analysis in Chapter 4: Analyzing Source Code.

Configuration of the dataflow module is available from the menu item *Configuration* | *Dataflow Analysis Settings*. There is also a quick selection of available dataflow settings from the toolbar menu.

This Dataflow Analysis Settings window shows a **Basic** and **Advanced** tab.

The **Basic** tab presents a choice from a selection of pre-defined dataflow settings to achieve different levels of dataflow analysis, from performing no dataflow to performing full regulatory level of dataflow analysis. The choices presented in the vertical slider bar are:

Off	Dataflow is turned off. This is the fastest analysis mode with no dataflow analysis. No dataflow messages will be generated.
Preset 1	This is a fast and shallow dataflow analysis mode. Select this level for initial dataflow analysis. Settings are saved into <Personality Directory>\preset_1.p_d
Preset 2	This is a medium depth dataflow analysis mode. Select this level for more in-depth analysis. Settings are saved into <Personality Directory>\preset_2.p_d

Preset 3	This is a full depth dataflow analysis mode. Select this level for rigorous in-depth analysis. Settings are saved into <Personality Directory>\preset_3.p_d
Preset 4	This mode is suitable for full regulatory compliance with, for example, ISO 26262. Settings are saved into <Personality Directory>\preset_4.p_d
User Defined	Use the **Advanced** tab to manually adjust the dataflow settings, which are saved into a Dataflow Personality. Settings are saved in a user-defined p_d file. This option is only recommended for advanced users.

The **Advanced** tab contains the means to create a custom set of dataflow analysis settings or to view the settings within the preset files.

- An **Individual Query Timeout** can be set in milliseconds. Queries are an internal part of the system, and several are typically run for each type of dataflow analysis performed. There can be many hundreds of queries performed for more complex functions. Limiting the time spent on queries will cut short longer ones and may affect accuracy of results.
- An **Aggregate Function Timeout** can be set in seconds to limit the overall time spent in dataflow analysis for each function. This may curtail results, and is designed to avoid lengthy analysis attempts on severely complex functions.

  Note:

  The underlying parameter passed to the analyzer component is specified in milliseconds, but is presented in seconds in the GUI for convenience.
- The **Inter-function Analysis Depth** setting determines the extent of function expanded into caller that occurs for inter-function dataflow analysis. To perform inter-function analysis, called functions that exist within the translation unit are expanded similar to compiler inlining into the call-site, and function parameters bound to internal arguments. With different non-zero settings of this slider-bar, a different size of resulting pseudo-function can be accommodated.

  The settings for this entry are:

  Off: No calls are expanded in analyzed functions.
  1. A function can grow by up to 20 simplified statements when expanding called functions.
  2. A function can grow by up to 200 simplified statements when expanding called functions.
  3. A function can grow by up to 2,000 simplified statements when expanding called functions.
  4. A function can grow by up to 20,000 simplified statements when expanding called functions.
  5. A function can grow by up to 200,000 simplified statements when expanding called functions.

  200,000 simple statements are considered to be a maximum, so when this number is exceeded, a 2756 message will appear.

  A “simple statement” refers to an internal deconstruction of source-level statements which includes expansion of temporaries, deconstruction of compound statements according to sequence points, and other requirements of dataflow examination.

These settings are saved into a dataflow configuration file (.p_d) and attached to the project. When the project is saved, this file is listed in the project file (.prj).

The complete list of dataflow configuration files located in the application personality directory, including the set of presets, is listed in the dataflow combo-box on the main GUI window for convenient application to any project.

When using a custom dataflow configuration file, the slider in the Basic tab will be positioned at the Custom setting, and the path and filename of the dataflow configuration file presented in the accompanying filename entry and in the dataflow combo-box on the main window. When creating a custom dataflow configuration file, dataflow will be automatically enabled (the –enabledatafow+ flag will be added to the p_d file). To later turn off dataflow, the choice of dataflow configuration should be changed to Off.

Personalities and Analysis

The Compiler, Analyzer, Message and Dataflow personalities contain configuration options that affect the output that is written to the .err and .met files. If you change a configuration option in any of these personalities, you will need to re-analyze your source file to generate updated .err and .met files.

When you analyze a file, QA·C will generate messages in the .err if they are not disabled within the message personality. Suppression of messages in the Message Personality will affect the display of the warning messages in the annotated source code.

If you change the message personality and enable a message that was previously disabled, you will need to reanalyze the code for the message to be generated.

Configuration of Application Options

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

- **Editor Preferences**, which determine the selection and configuration of external viewers.
- **Project File Options**, which control the file housekeeping activities in the application.
- **Default Personalities**, which specify the personalities to use for newly created projects.
- **Custom Reports**, which organise and configure additional user-defined reports.
- **Cross-Module Analysis**, which configures the additional analysis modules that operate on a project rather than on a single file.
- **Environment**, to set-up application-level environment variables and operate dependency-based file analysis.
- **Extensions**, to view and reconfigure automatically-detected add-on reports and baselining.

Editor Preferences: 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 so that HTML reports are viewed in your default browser. You can adjust these settings, if you wish, from the *Editor Preferences* tab of the *Configuration* > *Options* menu item.

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.

If you are using Internet Explorer, 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 request.

Setting the Housekeeping Options

QA·C's housekeeping options determine how long your output files are kept. The housekeeping options are defined by the *File Deletion Settings* on the *Project File Options* tab.

In *Analysis Output Files*, set the time period after which output files (.err, .met, and .i) are deleted. The *Clean Up Now* button will delete all analysis output files that are older than the time period specified in *Delete beyond*.

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

You can also delete output files manually through the menu item *Remove Analysis Output*, reached by right-clicking on a source file selection or from the *Analyze* > *Remove Analysis Output* menu item.

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

In *Project Autosave Settings,* set the time period after which QA·C will automatically save changes to your project.

In *Licence Auto-release*, you can force application closedown after a prescribed period 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.

The Message Browser also contains a 30-minute idle application closedown, which is triggered by the absence of application actions.

Default Personalities

Choose the personalities to be used as the default for all new projects. When you are adding new folders to an existing project, the personalities from the parent folder are used as the default.

Environment

Application Environment Variables

To assist in operation of environment variables used by the application, you can manage application-level environment variables in this tab. These environment variables persist through the life of the application, and can be used as part of the command line for custom reports, and form part of paths when saving a project or its personalities.

Application-level environment variables are overridden by any same-named project-level environment variables (defined through File | Project Properties and saved in the project file). All such environment variables are used to formulate full paths for a project on opening, and for any custom report.

Add a new application-level environment variable in the form EV=value, where EV is the environment variable name (as referred on command line using %EV%), and value is its chosen value. You can also Edit or Delete any EV in the list.

Note:

When creating an application EV, a dialog warning is issued when attempting to redefine an existing global one.

Analysis Environment Options

Apply dependency-based rules will cancel analysis for files that were already analyzed previously, and whose analysis results are up-to-date.

The algorithm for determining whether a source file requires re-analysis is as follows:

The newest file among the source file, all of its incuded files, and its collection of applicable personality files, project configuration file and baseline suppression file
is newer than
the oldest of its analysis output files.

Note:

Applying a different personality to a folder, when that personality is dated older than the relevant output files, will not of itself trigger a need to re-analyze.

Multi-core Bitmap Hex controls the enablement for analysis of each core in a multi-core system. This setting is represented as a bit-field with one bit for each core available in the system. By default, all bits are enabled. For example, in a two core system the value will be ‘3’ and a four-core system will show ‘F’. The only change permitted is to reduce the set of enabled cores by dropping bits from this hex value.

Source file analysis proceeds by scheduling each file through all the available and enabled cores in a system, and generally takes maximum advantage of splitting process time across all cores.

Product Extensions

Various separate utilties are designed to operate dynamically from within the QA C GUI.

QA Reports are installed along with the product, and can be configured from associated registry entries. They are presented on the **Reports** menu according to their registry configuration.

You can make a selection from the list of these reports, and choose **Edit Command Line** to view and alter its command line. You cannot alter the name of the report, nor the executable program itself. You can only change the composition of the command line parameters according to custom report configuration.

Changes are saved into the applicable registry entries, and will affect all QAC and QAC++ product versions that use these common add-on reports.

Baselining is also installed along with the product, and can be configured from its own registry entries, covering Generation of a baseline and application of this baseline to source files when they are re-analyzed. See Baseline Analysis Summary for more details.

CHAPTER 4

Analyzing Source Code

You can analyze a project, a folder, or a 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 cross-module analysis.

You can choose to *Pause* or *Cancel* the analysis. When you *Pause*, analysis queue will be paused (analysis currently in progress will be completed anyway). When you *Resume*, analysis will continue from that point. If you *Cancel*, analysis of the current file is aborted. If you now *Resume*, analysis will start on the next file.

If you wish to cancel a queue of analysis files, use the menu item or the associated button, *Clear Queue*, for this. You can *Pause* or *Cancel* the current file analysis before or after this action.

When a new set of source files is presented for analysis, previously successfully completed analysis entries are automatically removed from the Analysis Status window list. Files that terminate without a successful “Completed” status, usually 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 the return codes and text entries.

When a file has been analyzed 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 corresponding output files are newer than source **and** newer than the relevant 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 performed automatically.

Deep Flow Dataflow Analysis (DF^2)

Deep flow Dataflow analysis (DF^2) is a sophisticated dataflow analysis engine based on a Satisfiability Modulo Theories (SMT) solver. This phase of analysis is optionally run for each function after its primary parsing and semantic examination, and uses internal control and data flow structures as part of its operation.

To implement DF^2 analysis capability, QA·C builds an advanced dataflow model along with accompanying logic to produce suitable query inputs at various stages of function analysis. These queries are submitted to the SMT solver to yield high-grade dataflow diagnostic results and make further dataflow determinations on the code structure.

QA·C harnesses the reverse call tree processing to allow translation unit functions that were already processed to be “inlined” into their callers, providing for cross function analysis. This coupled with pointer aliasing provides for an extremely powerful and accurate level of analysis.

DF^2 analysis is intrinsically an intensive and sometimes time-consuming process and a QA·C analysis will usually take significantly longer if dataflow is enabled. This analysis phase is governed by a new set of parser configuration options to enable or disable functionality and control performance according to preferences of the user and source project needs. The main items of DF^2 configuration are the settings governing Query Timeout, Aggregate Function Timeout and Inter level, which are configured in Dataflow Analysis Settings.

There can be many internal SMT queries, especially for more complex functions, and most of these will run

efficiently to provide internal results to DF^2.

Note:
The number of queries is affected by the number and kind of analysis checks enabled. The greater the amount of analyzed code is, the greater the number of queries will be made.

The option to limit queries through the Query Timeout setting can help to avoid delay for queries that would take a long time (up to a few seconds). There can be a degradation in resulting diagnostic accuracy due to enforcing this timeout, however in practice the longest queries are often ones that do not provide much value to DF^2.

Note:
With no Query Timeout configured, DF^2 operates with a "safety valve" query timeout of 10 seconds. In practice, this length of SMT query is unlikely to ever be encountered.

Query timeout can achieve substantial time savings in overall analysis time of a source file. In addition, there is an optional Aggregate Function Timeout setting that will curtail the overall analysis time of a function. This will halt any further DF^2 processing, and cause no further analysis results to be produced for that function. Diagnostic message 2755 will be produced for each function that hits a configured aggregate function timeout.

Note:
When Aggregate Function Timeout is applied and the timeout value reached, any currently running SMT query is allowed to complete.

A full explanation on DF^2 operation and details of messaging output is provided in the Dataflow Reference Manual, located in the help\pdf installation folder.

Recommendations for First Time Operation of DF^2

Following are recommendations for setting up and configuring DF^2 analysis for the first time on a code project:

- First run each source file without DF^2 enabled and address any level 9 messages encountered.
- Run each source file with DF^2 enabled, and the "Preset 1" (preset_1.p_d) settings which consist of:

```
po df::query_timeout=50
po df::function_timeout=10000
po df::inter=1
```

  These values will provide an initial dataflow analysis without significantly affecting the analysis performance.

- If this initial run is successful and not excessive in time duration, more intensive DF^2 analysis is possible by progressing to "Preset 2" (preset_2.p_d) settings which consist of:

```
po df::query_timeout=100
po df::function_timeout=15000
po df::inter=2
```

- Further and more intensive DF^2 analysis is available with Presets 3 & 4.

Note:

The timings mentioned above are dependent on the machine speed and load, so they may not be suitable for all developer machines.

Recommendations for Fine-Tuning DF^2 Configuration Settings

Following are recommendations for setting up and configuring DF^2 analysis on code projects:

a) First run each source file without DF^2 enabled.

b) Run each source file with DF^2 enabled, no timeouts configured, df::inter=3, and with all level 0 and level 5 messages turned on. Obtain a Message Browser output of this result and keep this Message Browser instance open in successive analysis cycles for comparison.

c) If any source files take excessive time compared to step a), apply a Query Timeout, with initial suggested values between 100ms and 500ms.

d) Open a Message Browser for the full project and compare diagnostic results and counts in levels 0 and 5 with cycle b) results. If the message counts are similar, and in particular no additional occurances of message 2755 appear in level 0 dataflow recovery section, then it may be possible to reduce Query Timeout further to achieve overall analysis time savings.

e) Repeat steps c) and d) until a satisfactory overall analysis time with suitable DF^2 message output is reached.

f) Optionally consider Aggregate Function Timeout to cater for specific source files that may contain especially complex functions, and which may produce stubbornly high analysis times through DF^2.

g) DF^2 will be further speeded up if various categories of dataflow analysis are excluded from consideration. This is achieved by excluding various dataflow messages in the applicable Message Personality.

Commencing Analysis

You can analyze single files, sets of selected files, folder branches, or the complete project, which is equivalent to starting folder analysis from the root folder.

These three possibilities appear within the *Analyze* menu list, covering each of these choices. An additional selection determines whether cross-module analysis is run at the end of the analysis of all files in the project. See Cross-Module Analysis later in this chapter.

The **Analyze** 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, QA·C will perform the menu action *Analyze Selected Files.* If any folder other than the root folder is selected, it will perform *Analyze Folder*. If the root folder is selected, it will perform *Analyze 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 per source. The output files have the same name as their source file but with the addition of further file extensions. For example, the .met file for source.c will be named source.c.met, and its .err file will be source.c.err.

QA·C always outputs the following files:

.err	Diagnostic records of messages
.met	Parsing/semantic data

QA·C may output the following files, depending on your configuration:

.i	Pre-processed 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 using *Analyze* > *Remove Analysis Output.* You can also apply housekeeping checks on output files for a file or folder manually, by selecting the file or folder and using *Analyze* > *Check Analysis Output.* For both of these actions, folder selections will also ripple to sub-folders.

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

.err Files

Each .err file contains a set of diagnostic records for messages switched on in the Message Personality when the analysis is done. In the Message Browser, 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

Each .met file contains analysis data from the parsing process, as described in Appendix F: Metric Output File. The data from these files can be used in reports and with the various browsers.

You can inspect these contents for a selected analyzed file by choosing *Metric Output File* from either the *View* menu or the pop-up menu.

Note:
.met files may be deleted automatically, 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 pre-processed source code. This file will only be generated if you have enabled *Preprocessed Output* in the *Analyzer Personality.* Pre-processed source code can be viewed by choosing *Preprocessed Source* from the *View* menu or by clicking the *Preprocessed Source* button.

Preprocessed source code can be useful when investigating warnings generated or in the expansion of macros.

Annotated source code may not provide sufficient information to diagnose the root of the problem, if warnings have been generated in the expansion of macros. A pre-processed file can be useful here: generate a pre-processed file, add it temporarily to your project, and analyze the pre-processed file itself. You will be able to see all the code from header files, and all macros will be fully expanded.

Comments are always stripped from pre-processed code. However, QA·C has a feature *Include filename and line numbers in preprocessed listing,* whereby new comments are introduced in the pre-processed file showing the source or header file name and line numbers. This allows you to see the origin of the pre-processed source 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);
}
```

Pre-processed 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 ) ;
}
```

Pre-processed 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.c
 */
/*      2 */ int step1 ( int x )
/*      3 */ {
/*      4 */ int r = 0 ;
/*      5 */ if ( glob > 10 )
/*      6 */ {
/*      7 */ r = x * x ;
/*      8 */ }
/*      9 */ return ( r ) ;
/*     10 */ }
```

Managing Output Files

When you analyze many source files, the volume of data written out can become very large. You should take care to control the amount of data generated or displayed.

Suppressing Analysis Output from Header Files

The data in .err and .met files is associated with both source and header files, and that association with certain header files may be of marginal importance. If a 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 cross-module 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 suppress the output, apply *Suppress Output* to an include path in the Analyzer Personality or Compiler Personality. This should only be done when you are satisfied that the warning messages from the header files are not important.

Note:

When output for an include path or file is suppressed, .met file will not contain the corresponding entries. Consequently, secondary or cross-module analysis will not be performed on suppressed headers. This is typically intentional for third party libraries. However, this is unlikely to be correct for project specific include files.

If the header file is directly associated with the source, you may want to generate the messages from the header file, but suppress their display. This is recommended for your project headers, if you are satisfied that they are stable and free from major problems.

The display of header file warnings is controlled by *Message Personality* > *Display* > *Display Header Warnings.* By default, QA·C does display header file warnings, 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 primary QA·C file analysis. A script or program can make use of the parsing information in the .met file to generate additional warnings.

Configuring Secondary Analysis

To run a secondary analysis program/script:

1. Open your *Message* Personality for your current project via *Edit* > *Folder Parameters…* > *Message Personality*, or for any message personality file via *Configuration* > *Message Personalities…*
2. On the *Advanced Settings* tab, click *Setup* in *Secondary Analysis Command String.*
3. In the *Secondary Analysis Settings* dialog box, click *Enable Secondary Analysis Processing.*
4. 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 shortcuts can be used to pass relevant information to the secondary analysis executable. These shortcuts are described below. The resulting command line string will appear 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 to the secondary analysis executable. The optional "+" flag will have prefix –via before this file on the command line.
%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 "outputpath" 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, or any other descriptive code for a known exit. For details, see Appendix E.

Secondary Analysis Naming Convention Checking

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

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

1. From the *Configuration > Message Personalities …> 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, a special strategy is needed.

The GUI must invoke a script file that runs each secondary analysis task in sequence. This script must pass all relevant parameters and execute the analysis supplying the correct parameters for each task. The script should also check the return code from each task and take appropriate action if there is a problem. Finally, if all tasks complete successfully, the script should exit with a zero.

The following is an example batch process for this. There can be many variations, depending on the project – this is intended as a suitable starting point, which can be customized to suit the individual projects. It is also possible to write a C or C++ executable to start the secondary analysis tasks.

This 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 -op "<outputpath>"
"<filename>""
```

Note the double quotes enclosing the entire string argument to /c. The double quotes tell 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 have contents similar to:

```
@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%
```

Note:
To ensure that errwrt.exe will work through the batch file above, it may be necessary to adjust the PATH environment variable, i.e. set path=%qacbin%;%path%

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. Output from CMA is normally written to a special project .met or .err file. For example:

- Certain metrics 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

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

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, and correspond to:

Program	Your CMA program
%Q	A fixed label representing the application that generated the analysis data, in this case QAC.
%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 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. The GUI creates a file in the temp directory, which 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. –cmaf <root_folder> which specifies the output files, into which CMA err and met data is written (see below).
%D	This parameter can specify the root folder default source path.
%O	This parameter can specify the root folder output path.
%J[+]	This parameter can specify the project name. Since this entry may be used as a path or filename, any illegal file I/O characters in the name are substituted with underscore.

CMA Output Files

The output from cross-module analysis is placed into two files, one for message records (.err) and one for 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 are written to a file formed from this base filename with a .err extension, and metric and other textual information into an equivalent file formed with a .met extension (such as root_folder_name.met).

Running Cross-Module Analysis

To run CMA, choose *Cross-Module Analysis* > *Run* from the Analyze menu.

You can also set CMA to run automatically after analysis of all source files in a project, by choosing *Cross-Module Analysis* >*Include CMA in Project Analysis* from the Analyze menu. With this menu option checked, when you run Analyze Project from the Analyze menu, CMA will be included.

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.

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 that are 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 various declaration checks which are in line with coding standard advice. These cover conflicting typedefs, namespace function, object and type declarations or definitions.

When running CMA, the project .met and .err files are created or recreated.

These files are used by various browsers when displaying metrics or analysis messages.

Analysis Log

At different stages of the analysis, the source files can have one of a number of analysis codes. These codes will be returned before or during analysis or after analysis has completed.

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.

A list of analyzer return codes is included in Appendix E: Program Return Codes.

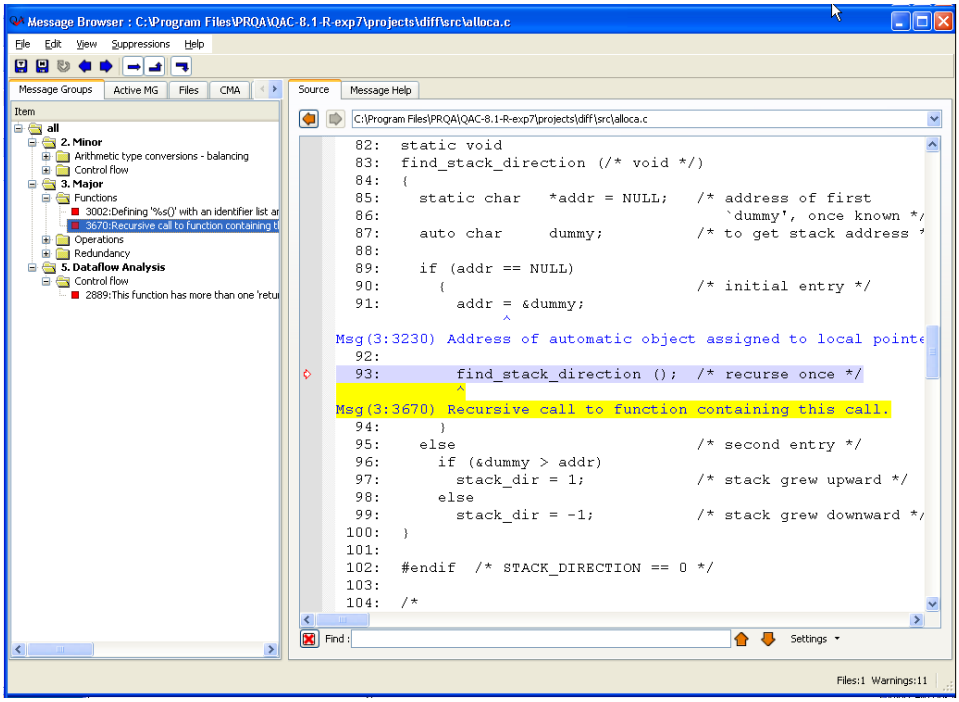
CHAPTER 5

Viewing Analysis Results: Message Browser

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

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 (CMA) .err file in all activations from the GUI whether single files, groups of files, folders, or the entire project.



The Message Browser can be launched by several methods:

- by selecting one or more files and choosing *Messages* from the *Browse* menu;
- by selecting a folder and choosing *Messages* from the *Browse* menu (this will display messages for the selected folder and for all sub-folders).

Messages will only appear if they are enabled in the Message Personality during file analyzing. 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 re-launch it with the revised message subset.

If you re-analyze 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 analysis messages.

The Message Browser has three principal 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 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 (Warning 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. After re-analyzing the project and re-launching the Message Browser the new message subset will be displayed.

If you have re-analyzed the files included in the Message Browser without changing the Message personality, you will need to *Refresh Warnings* from the *File* menu to load the new set of warning messages.

Messages originating from cross-module 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 to which they point, 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 it can provide more specific information than the main message but 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 the annotated source code produced by the Message Browser.

Clicking on 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.

Navigation is remembered, which lets you click back through the history. New clicks are inserted into the history, enabling you 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.

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.

Displaying Line Numbers

To display the 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 five possible tabs to be seen in the *Warning Summary View*:

Message Groups

The Message Groups tab lists all detected messages according to the message level and group. The number of diagnostics for each message group and level is displayed next to the entry.

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 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 Analyzer 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 includes 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.

Active MG

The Active Message Groups tab lists levels and groups for which the active message count is greater

than 0. The number of diagnostics for each message group and level is displayed next to the entry. A message that has been suppressed with Baselining or a Comment-based directive will not be considered active.

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.

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.

The index column displays the output sequence number for a diagnostic.

Note:

Dataflow analysis takes place after the entire translation unit has been parsed, as a result these analysis messages will have higher indexes than other QA C messages.

The warning list view also contains associated context messages, which are expanded a single level at a time.

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

Changing the Format of Annotated Source

During analysis, QA·C generates a .err file containing a set of diagnostic records for messages switched on in the Message Personality when the analysis is done. The Message Personality allows you to suppress the display of individual messages, or of complete message levels and to specify the format in which the warnings are presented.

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, as it would appear in the Message Browser, is reproduced with some extra commentary added, 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 were to be 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 (Analyzer 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 Analyzer 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 initialized 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 Estimated Static Program Paths metric has exceeded the specified threshold. (For this function, the value of STPTH 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 identifer 'entry' has been replaced with 'misc'
   so there is no confusion over K&R C.
```

```
*/
int     Miscchar[MEMALLOC] = {0};
/*
 Initialize the array with all 0's first, through
 default initialization.
*/
int     spaces=0;
int     lines=0;
FILE *  fp;
/*
 Typing error fixed on 'FILE *fp'.
 Variables are now initialized and aligned properly.
 'unsigned int' is removed as it causes implicit
 conversion.'count' has been removed as it is redundent.
*/
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 initialized
       (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 helped to reduce the values of both STCYC
and STPTH metrics.
      */
```

Displaying Diagnostic Messages

Typically, many diagnostics are produced when a project is analyzed. To make it easier to focus on particular facets of the analysis, there are several ways in which the display of diagnostics can be controlled, including:

- Message Personality
- Hiding Messages in Message Browser
- Baselining

The most important of these is your Message Personality, but the display of messages can also be influenced by choosing to hide certain messages from view in the Message Browser.

Where baselining has been enabled, diagnostics can be suppressed from view altogether, by transferring them to a "suppressions file".

Overview of Baselining

Baselining is a technique to reduce diagnostics. It is particularly useful when legacy code requires maintenance.

The baselining process considers the differences in two versions of a source project. Pre-existing diagnostics from the earlier version are removed from view in the later version, making it easier to focus on the newly-introduced diagnostics.

Baseline generation is designed to operate primarily through the QA·C graphical user interface (GUI), using *Analyze* > *Generate Analysis Baseline*. However, baseline generation can be run in an equivalent command line operation by experienced users.

You will need a backup of your baseline source for use when creating the difference information, unless you are using a version control system.

Generating a baseline will create a file which is used to hold all baseline suppressions for the project. The default name for this "suppressions file" is

```
<project>.sup
```

You can make modifications to the configuration through the menu item *File* > *Project Properties*.

Before generating a baseline for the first time for any project, you need to open your project and analyze it completely, including secondary analysis and cross-module analysis. Now you can use *Analyze* > *Generate Analysis Baseline* to create the baseline in the <project>.sup file.

Initially, all the diagnostics from the existing analysis are used to form the baseline. Diagnostics are described as “suppressed” or “baselined” if they form part of the current baseline, or as “active” if they do not. You can select and remove diagnostics from the baseline through *File* > *Baseline Admin Mode* in the Message Browser.
When you create a different version of the project (that is, when any source or header file changes), the diagnostics previously created for the earlier version will have to be recalibrated in line with the changed source.

This means that the tool needs to know the differences between the version of the source that was used to create the baseline and the version of the source that you are working on. The tool maintains the baseline in accordance with the difference information, thus keeping the suppressions in line with the working source.

When a baseline is regenerated, that is when the name given for a source matches the file name which already exists in <project>.sup, the new entries to <project>.sup will be appended to those already present. The existing data will not be overwritten. However, if the filename is the same as for an existing file but the identifier is different, the old data will be removed before the new data is written.

Message Groups and Active Message Groups

When the Message Browser starts, it loads the diagnostics from the .err files. The top left pane of the Message Browser contains an extra tab labelled *Active MG* next to the *Message Groups* tab.

The *Active MG* tab displays the unsuppressed message groups. If you right-click on a message group in the active list, you are offered the option to *Hide* it, that is, to remove the group from the active list. This does **not** baseline the group, just hides it from view within the Message Browser, but you can choose to baseline it later if you wish. To make a Hidden group visible in the *Active MG* list again, right-click on the group where it appears under the *Messages Group* tab, and accept the offered option to *Show* message.

There is much more detail about various aspects of using baselining available in Appendix J.

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, for each analyzed file, the number of messages generated at each message level.

- **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 used in your project.

- **COCOMO Cost Estimation Model**

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

- **QA Reports**

  If QAR add-on component is installed, four additional menu items are presented to generate HTML and XHTML reports: Code Review Report, Compliance Report, Quality Report and Suppression Report.

- **Structure101 Generation**

  Depending on whether an add-on component is installed, an additional menu item is presented to generate model data for Structure101, a rich architectural analysis add-on component to QA·C and QA·C++.

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. Messages suppressed with comment based suppressions and #pragma PRQA_MESSAGES_OFF are not included in the count of messages unless the –hiddenwarnings option is set to on. The output lists all analyzed source files and

associated header files, and details the number of detected warnings for each message level from 0 to 9. The report provides a total for each warning level, 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 numbers on the Active MG tab, when there is no baseline present. 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 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.

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 used 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 Boehm[1]. 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 types:

- 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. banking or accounting systems).
- 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. Windows or database applications).

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 the delivered code is calculated as the metric STTPP.

The values of a and b depend 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:

$$D = a * E^b$$

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

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

The values of a and b depend 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 Euro.

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 Cross-Module 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 visualization of interfaces and the internal structure of source code. QA·C displays code structure as:

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

Both views are displayed for the selected folder or files, and .can be changed within these browsers through menu selection of the *Select Analyzed Files* dialog box.

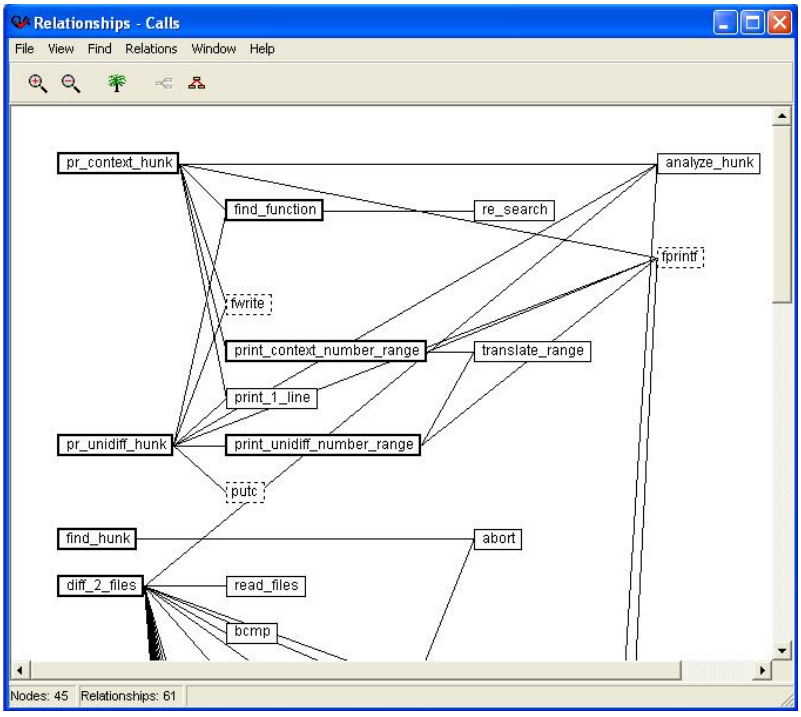
To generate these diagrams choose *Relationships* or *Function Structure* from the *Browse* menu or click the respective toolbar button.

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 the source/header files where the file has not been quitened (-q).
- Call relationships that show all function calls, both internal and external (but excluding functions called through pointers), within your source files. Recursive (both single and multiple) functions are marked with dotted red lines. When selecting two functions that have a recursion between them, the first selected will determine the line that is marked in bold when selecting the other.
- 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.

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: Analyzing 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 *Single Fan In/Out* or *Full Fan In/Out* from the View menu. *Single 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

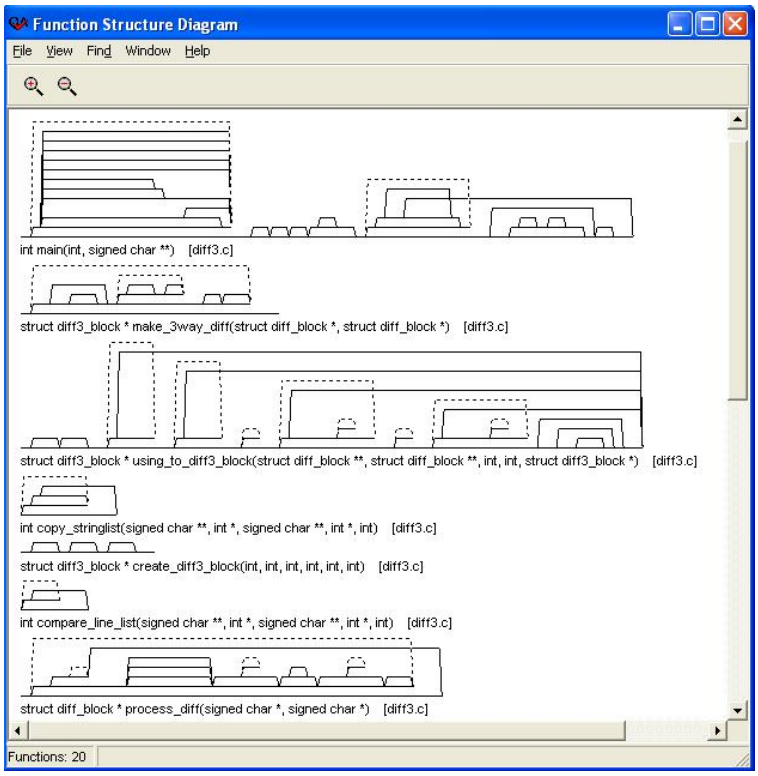
Choose *Save Image* from the *File* menu. The image can be saved as an either enhanced metafile (.emf), a windows metafile (.wmf), or a bitmap (.bmp).

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 visualization 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
- return 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.
- To view the source code location at which the function is defined, select the function and click **Source Code**.

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.

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. Choose *Source Code* from either the *View* menu or the pop-up menu. This displays the source code at the corresponding point in the function if the editor can jump to the specific line of code.

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, but each provides a different view of aspects such as complexity, readability, residual bugs, or testability.

QA·C calculates 67 different metrics, of three types:

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

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 cannot be usefully presented through any of these visualizations, 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 generated metric value falling outside these limits will result in an analysis warning. Thresholds can be set as part of configuring a coding standard.

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

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

where:

metric	The metric name as it appears in Appendix D.
op	A relational operator (<, <=, ==, >, >=).
value	The integer value for which QA·C will generate a warning message if the threshold is broken. The value can be an integer or a floating point value.
message	The warning message that is generated, as an optional entry. This message number should refer to a warning message that you have added through your user message file. See 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.
```

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 abandon further lookup in the threshold list for that value.

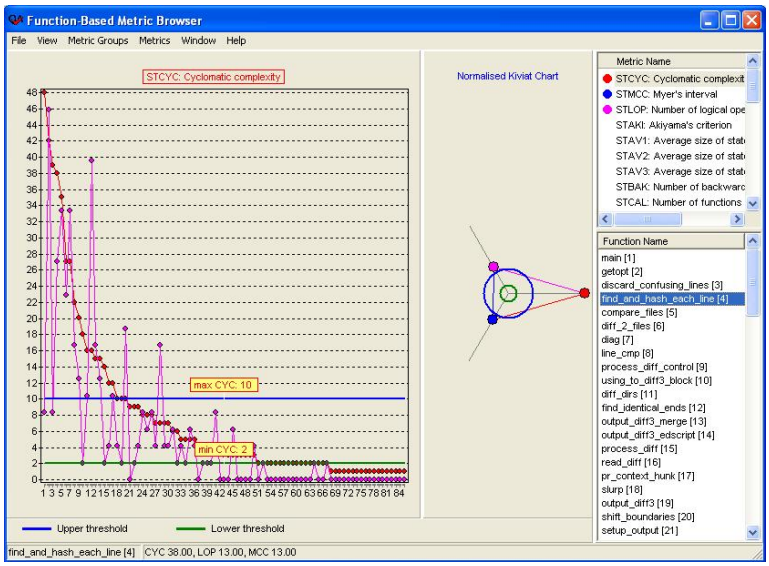
As a result, the order in which thresholds appear in the list is important. If you want QA·C to issue different warnings for increasing degrees of severity, you should ensure that your thresholds appear in decreasing order of severity in the list. For example:

```
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 will be 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.

The Metrics Browser

The Metrics Browser is used to display both 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, or alternatively from the *Metrics* menu, you can select up to ten metrics. Selected metrics are plotted on the graph by double-clicking them.

The color 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 the names of all the functions or files currently displayed in the

browser. The item is only displayed if its file has been selected for display. 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 each 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 Values* from the *View* menu. If you sort by name, the list will be sorted in the alphabetical order. If you sort by metric, the list will be ordered 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 sort in either ascending or descending order, by right-clicking on either graph or on the selected metric in the Metric Name Listing Panel.

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

Note:
Certain function metrics, calculated in cross-module analysis, are calculated for declarations as well as definitions, and therefore cause additional function entities to display in the Metrics Browser.

Metrics Graph

The metrics graph plots the metrics that you have chosen to display for each function or file. A different colored line corresponding to the color 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 focused metric.

Also displayed on the graph are two horizontal lines that reflect lower and upper limits for the focused 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 distinct from any thresholds used to generate warning messages. The 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.

Reloading Metrics Data

If you have re-analyzed 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 there 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 information for this group 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 have been selected. 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, you should save the file as a .prm file in the demographics directory. The Demographics Browser will only display demograph files that are located in this directory.

Printing the Graph

To print 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

To save 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 Diagram 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

To display the 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.

Displaying 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

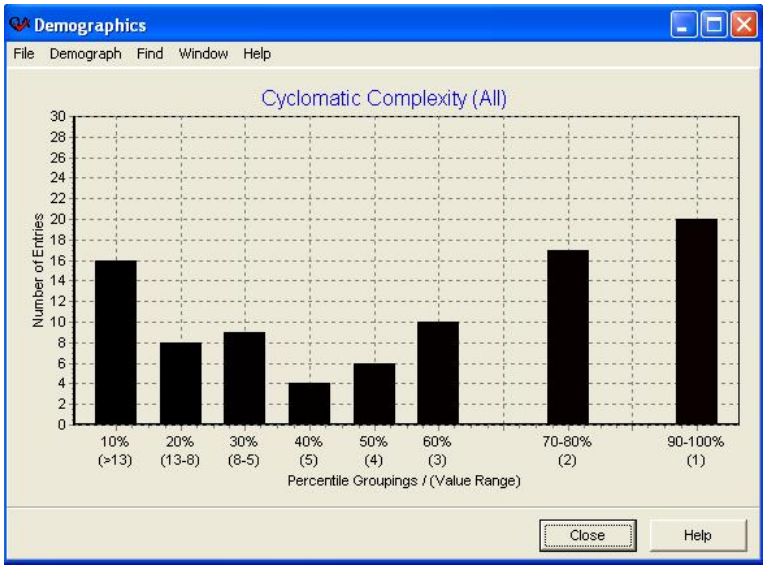
The Demographics Browser is activated by selecting *Demographics* on the *Browse* menu or by clicking the *Demographics* toolbar button. Three function metrics are supplied with various demograph data to display in the Demographics Browser:

- Depth of Nesting (STMIF)
- Cyclomatic Complexity (STCYC)
- Estimated Static Program Paths (STPTH)

A Demograph displays a bar chart of metric values plotted against some industry averages. PRQA has analyzed 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. Your metrics are plotted against these standards to measure the quality of your source code. The value range for each percentile is displayed in brackets below each 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 categorizations that address different development modes:

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

You can choose one of these default demographs from the *Demograph* menu. Cyclomatic Complexity (All) is displayed by default. If you have created a demograph file in 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 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 the functions that contribute to a particular bar by right-clicking on the bar to bring up a Metrics Listing window.

If you select *Listings* from the *View* menu, you will be presented with a list of all functions in the selected files.

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 item, or to view the source code.

Selecting Additional Files

To select 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-analyzed your source files, choose *Reload Data* from the File menu to load the new analysis data.

Saving the Demographics Graph

To save 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

To print the graph, choose *Print* from the *File* menu. The graph will be printed on your default printer.

Displaying Source Code

To display the source code, select an entry from the Demographics Listing and click *Source Code*.

Displaying Function Structure

To display the function structure, select a function from the Demographics Listing and click *Structure*. This option is only available for function-based metrics.

Displaying a Kiviat Diagram

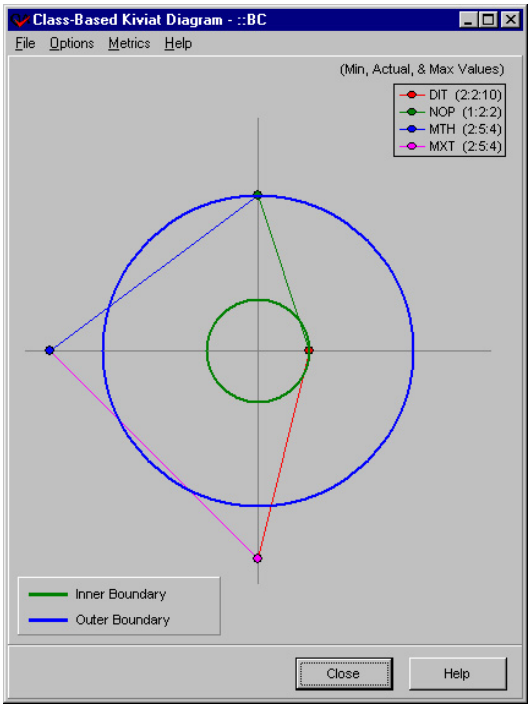
To display the Kiviat diagram, select an entry from the Demographics Listing and click *Kiviat Diagram*. The diagram will be drawn for all available metrics of the given type.

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 Kiviat Diagram

From the demographics browser, you can generate a standalone Kiviat diagram. A Kiviat diagram is a way of displaying a number of different metrics for a single file or function.



Two concentric circles represent high and low boundaries 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 colored nodes and its value is represented by the distance of from the node from the centre of the circles. The metric values are normalized 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 ease of display. 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 a 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 normalized 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 Kiviat Diagram

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

Saving the Kiviat Diagram

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 visualization features, such as relationships and metrics browsers.

The principal programs are:

- qac.exe which analyzes your files;
- viewer.exe which launches the Message Browser.

In addition, there is an errsum.exe utility program, which generates a summary of diagnostics across a project or set of source files.

These programs are normally launched from within the GUI, but they can be run directly from a command line batch or make file. It is often convenient to configure your own development environment so that source code can be analyzed 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.
QACHELPFILES	The location of the HTML help pages, explaining the rationale behind each generated message. This should be set to the help directory, and message HTML 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.

Note:
If you intend to run QA·C concurrently (e.g. GUI and qac.exe), it is unsafe to rely on QACOUTPATH. It is recommended to specify the output path with an -op entry, instead.

Configuration Options

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

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

instructs QA·C to analyze 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 **viewer** and other utilities. 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:

```
-size int=16
-SIZE int=16
-s int=16
```

will all set the size of the int datatype to 16 bits.

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.

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 in an option file. This is usually a text file that contains a number of configuration options, each on a separate line. Option files are specified in 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.

Note:

You can include comments in the options files by setting the –rem option or by prefixing the comments with an asterisk (*).

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. Options that are not relevant to a particular program 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 analyzed from the GUI, options from the three personality files are concatenated and saved in a temporary file in a directory specified in the [DirLocations] section of the qac.ini file. Analysis is then run using the command:

```
qac –via <temp-dir-location->\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 to settings.via.

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

Analysis would be run with the command:

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

This is the recommended approach if you want to analyze 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.

Analyzing Source

Configuring Primary Analysis

Files are analyzed on the command line by entering:

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

where:

Option	any combination of configuration options.
source.c	the source file to be analyzed.

Output files will be generated in the directory specified by QACOUTPATH, unless you specify a different directory with the -op option.

Note:
If you intend to run QA·C concurrently (e.g. GUI and qac.exe), it is unsafe to rely on QACOUTPATH. It is recommended to use an -op entry, instead.

Common Analyzer Options

-i *path*	Specifies the paths of header files.
-q *path*	Suppresses messages from header files in path.
-d *ident*[=[*value*]]	Defines macros.

-op *path* Specifies the path to which output files are generated.

-tabstop *columns* Specifies the number of spaces represented by a tab character.

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

Analyzer Return Codes

When qac.exe has been run successfully, it exits 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. A hard error is triggered if 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 analyzer return codes is included in Appendix E: Program 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, typically, will use the relevant .met file to perform customized checks.

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

As an 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:

QAC the product label for internal usage.

option any combination of configuration options.

naming.nrf the set of naming rule configuration entries (See Appendix I, Naming Convention Checking for more details).

source.c the source file to be analyzed.

Note:
Secondary analysis configuration through the GUI is handled in a special way, which is not available when running analysis on the 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 launch and control the program using a batch script.

Running CMA Checks

A cross-module analysis program can be run after all file-based analysis is complete. Typically, it will use the set of all .met files to perform further customized checks.

For example, the command to run the default supplied CMA program is:

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

where:

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

Viewing Diagnostic Output

Message Browser (viewer)

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 held in the .err files for all source files in its input list. viewer.exe will look for the .err files in the path set by either QACOUTPATH or the -op 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 for a list of files enter:

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

where:

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

Note:
Internally, GUI also uses –list option when invoking the Message Browser. The temporary file used as the argument is formed as a unique filename. At the end of the process, it is renamed to filelist.lst as a convenience. See Appendix B: Configuration Options.

Project Warning Summary (errsum)

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:

QAC	the product label for internal usage.
option	any combination of configuration options.
filelist.lst	the full set of project source files (see Appendix B: Configuration Options for formatting details).
output.txt	represents the output file to which a summary of diagnostics will be generated.

Optional additional parameters:

-CrossModuleAnalysisFile <file>	Additional cross-module analysis (normally included in *filelist.lst* above.
-po "ERRSUM::width=<width>"	Specify a target report width between 60 and 160 columns wide.

The errsum.exe program output is formatted in plain text, with columns representing the de-duplicated set of diagnostics captured from all phases of project analysis (primary, secondary, and CMA), and rows representing individual source and header files.

Project Warning Summary (errsum.exe) produces the most comprehensive and accurate count of diagnostics across a project. It de-duplicates common header warnings, merges primary, secondary and CMA diagnostics for each file (source and header) in the project, and takes proper account of comment-based suppressions.

CHAPTER 10

Advanced Topics

This chapter discusses topics that are helpful in the efficient administration of QA·C. Most users will want to integrate QA·C into an existing development environment. Usually, this will involve developing some guidelines for the location of directories, and customizing the product 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 cross-module analysis checks, or configure custom reports, or configure a layout standard.

Customizing the Message System

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

The two files used by the message system are:

- the standard 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.

Message File (qac.msg)

The message file, qac.msg, defines:

- 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 from 0 to 9. In the standard message file, there are 10 #levelname statements that are defined as follows:

```
#levelname 0 Information
#levelname 1 Obsolete Messages
#levelname 2 Minor
#levelname 3 Major
#levelname 4 Local Standards
#levelname 5 Dataflow Analysis
#levelname 6 Portability
#levelname 7 Undefined Behavior
#levelname 8 Language Constraints
#levelname 9 Errors
```

You can 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 an unlimited 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 are defined in a set of #levelname and #define statements such as:

```
#levelname  3  Major
#define  MAJ_Decl          3     Declarations and definitions
#define  Maj_Ops           3     Operations
#levelname                 8     Language Constraints
#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:

\n	new line
\t	tab-stop
\	line continuation
\\	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 customize 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 will allow for customizing messages without changing the standard message file, which may be revised in the 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 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 that way.

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 all parts of the original message record into your user message file. That means 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 from 0 to 9. Message levels must be added in the following format:

```
#levelname levelnumber levelname
```

where:

levelnumber a number used to organize 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 in

the GUI by *Message Personality* > *Display* > *Message Format*.

Message Display

The -format option accepts the following display specifiers:

%f	file name (as passed to qac.exe)
%F	file name (absolute, including path)
%B	base filename
%l	line number
%c	column number
%C	column number less 1 (left column is 0)
%g	message level
%n	message number (raw integer format)
%N	message number (zero-padded to 4 digits)
%t	message text
%v	verbose/reference text
%u	context message depth
%^	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):

```
%?c>0%(%?u==0%(%q%R(%C, )^\n%:%?L%(%:%q%R(%C, )^\n%)%)%)
```

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 specialize 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 the 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 behavior of %^, or to implement similar functionality from scratch.

As an example of %R(num,char) usage:

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

will cause the following display of caret 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 from that of the previous error.
%f(%l) ++	The file name and line number (in brackets) is 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.
```

Writing Secondary Analysis Checks

Some examples of what could be done with Secondary Analysis are given below.

Note:
pal.exe provides some standard Secondary Analysis, i.e. Naming Convention checking, for more details see Chapter 4: Analyzing Source Code.

Global Data Analysis

Certain aspects of global data usage could be enforced, e.g. that access is protected with a critical section, or simply reporting on access to global data, for further manual review. This could be implemented by inspecting the <DEFINE> and <R>X records in the .met, see Appendix F: Metric Output File.

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 you are 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 Files section in this chapter.

When your secondary analysis process 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 with the standard QA·C warnings. To run the errwrt.exe utility, enter:

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

where

prod	QAC (errwrt is a utility which is also used in other PRQA products.)
file	The complete path to 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 built-in 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. This must be used on a multicore system since each analysis process may write to different output paths.

-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

<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 mode.

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

Writing Custom Reports

The Reports menu contains a section for adding Custom Reports, where you can add new user-defined reports that you have developed. The setup and execution configurations 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 shortcuts:

%Q Specifies the product identifier (QAC) that is required for many of the product components, e.g. Warning Summary (errsum.exe) or Message Browser (viewer.exe).

%P[+] Formats and passes the personality settings of the root folder, in a temporary file labelled settings.via.

With the + postfix, it prepends –via to the entry.

%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.

%D Specifies the root folder default source path.

%O Specifies the root folder output path.

%J Specifies the project name. Since this entry may be used as a path or filename, any illegal file I/O characters in the name are substituted with underscore.

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, errsum.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 a custom report 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 analyzes 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.

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.

You can reword existing level names with a user message file. See User Message Files in Chapter 10: Advanced Topics.

The default personality displays all messages detected by the 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 Warning Level settings of the Message Personality, this only affects the **display** of messages. As a result, you will not need to re-analyze your source code.

If you change the display setting of individual messages and then re-analyze your source code, diagnostic occurrences of all disabled messages will **not** be output to the .err file.

The standard message level configuration is described in Standard Message File of Chapter 3: Configuring QA·C. 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)

These messages indicate information level warnings with the lowest level of importance.

Obsolete Messages (1)

Messages marked as obsolete are kept for backwards compatibility for 2 product releases, and will be subsequently removed.

Minor (2)

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

Major (3)

Source constructs involving dangerous conversions, expressions, and assignments based on data types and alignments.

Local Standards (4)

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.

Dataflow Analysis (5)

These messages provide analysis of run-time behavior, identifying problems ranging from serious issues such as undefined behavior to conditions which are frequently associated with coding and logic errors.

Portability (6)

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

Undefined Behavior (7)

These messages identify constructs that are either explicitly or implicitly undefined in the ISO C standard.

Language Constraints (8)

These messages identify constructs that are syntactically correct, but dangerous because their behavior 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 before resolving all level 9 messages.

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

Personality Settings	Option
Warning Messages	
Message Groups	
Information	-su 0
Obsolete Messages	-su 1
Minor	-su 2
Major	-su 3
Local Standards	-su 4
Dataflow Analysis	-su 5
Portability	-su 6
Undefined Behavior	-su 7
Language Constraints	-su 8
Errors	-su 9

Advanced Settings	
User Message Files	
User message file (path)	-up
User message file (extension)	-usr
Secondary Analysis Command String	N/A
Message Expansion List	-emhm
Save Message Settings By	
Suppressed Messages	-n
Selected Messages	-o
Display	
Message Display	
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

Analyzer Personality

Personality Settings	**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	-ppl
Include filename and line number in preprocessed listing	-ppf
Implicit Conversion Message Options	
Generate unsigned to larger signed messages for wider types	-ss
Generate type conversion messages for equivalent 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 Settings	**Option**
System Headers	
System Header Includes	-i
Suppress Output	-q
System Macros	
System Macro Defines	-d
Data Types	
Data Type Interpretation	
Right Shift Behavior	-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. In the descriptions below, any alternative abbreviated versions are 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. The syntax within the GUI is the same as on the command line, except that the option name is not included.

-a, -align

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

The alignment of data types may be constrained by your hardware and compiler. For example, your environment may allocate a char at any memory address, but require an int to start 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, invalid to do the reverse. QA·C generates 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 data 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	<=short	1
short	16	16	<=int	2
int	16	32	<=long, <=64 bits	4
long	32	32	<=long long, <=128 bits	4
long long	32	64	<=128 bits	8
float	32	32	<=128 bits	4
double	64	64	<=128 bits	8
ldouble	64	64	<=128 bits	8
codeptr	16	32	e.g. int (*)()	4
dataptr	16	32	e.g. char *	4

-ar, -arithrsh

Syntax:	-arithrsh {+|-}
Default:	-arithrsh-
Program:	qac.exe
GUI:	Compiler Personality > Data Types > Data Type Interpretation > Right Shift Behavior
Function:	C has only one right shift operator (>>). It can be interpreted as either an arithmetic shift or a logical shift. The –ar option defines whether the right shift operator >> performs an arithmetic (-arithrsh+) shift or a logical (-arithrsh-) shift. The default is that the logical shift is used. The ISO C standard states that the result of an arithmetic right shift on a negative signed number is implementation-defined. That is, each individual compiler has to specify the effect of shifting negative values right.

-bits, -bitsigned

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

The ISO C standard states that a bit-field has a type that is 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:	-comment {a, n, i}
Default:	-comment n
Program:	qac.exe
GUI:	Analyzer 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:	-cmaf *filename*
Default:	no entry
Program:	pal.exe, viewer.exe, errsum.exe
GUI:	N/A
Function:	For Cross-Module Analysis tools, e.g. pal.exe, this option specifies the output files into which project-based err and met information is delivered. For display tools, it specifies the location where this same information is located. When operated through the GUI, the -cmaf entry is constructed from the root folder output path and the name of the root folder.

-d

Syntax:	-d *ident*[=[*value*]]
Default:	no entry
Program:	qac.exe
GUI:	Analyzer 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**
-d *ident*	#define ident 1
-d *ident=value*	#define ident value
-d *ident=*	#define ident

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:

-d *ident*=_ignore ignores the specified identifier and the next token.

-d *ident*=_ignore_semi	ignores the identifier and everything up to and including the next ;.
-d *ident*=_ignore_paren	ignores the specified identifier and the following block of code. The block may be of any length and may be surrounded by (…), […], or {…}.
-d *ident*=_ignore_N	ignores the specified identifier and the next N tokens.
-d *ident*=_ignore_at	when the specified identifier is preceded by an '@' character, both tokens are ignored.

The ignore operation is processed before any other macro substitution takes place.

Notes: using double quotes within a macro

The configuration system strips out quotes, so a quoted string cannot be defined directly.

The recommended way to generate a quoted string macro is to use STRINGIFY.

For example:

adding the following to your personality

```
-d "_PRQA_STRINGIFY(x)=#x"
```

and using

```
-d A="_PRQA_STRINGIFY(1 2 3)"
```

generates a quoted string of "1 2 3", equivalent to the definition #define A "1 2 3".

Another method is to create a header file that contains the definitions in standard syntax, and add a link to that header file as a force include file from the compiler personality.

For example:

In myheader.h:

```
#define __VERSION__ "this version"
```

In the personality (and it is best to give full pathnames here):

```
-fi myheader.h
...
```

Notes: _munch

The macro _munch is a deprecated alternative to _ignore.

-e, -echo

Syntax:	-echo *text*
Default:	no entry
Program:	qac.exe
GUI:	N/A – console output is ignored
Function:	Specifies that *text* is echoed to the console. This option can be used to show the configuration files that are being referenced.

-ed, -enabledataflow

Syntax:	-ed {+|-}
Default:	-ed-
Program:	qac.exe
GUI:	Configuration > Dataflow Analysis Settings
Function:	Turns on the added dataflow (DF^2) analysis functionality. See Deep Flow Dataflow Analysis in Chapter 4 for more details.

-emhm, -expandmultihomedmessages

Syntax:	-expandmultihomedmessages *message_number_list*
Default:	no entry
Program:	viewer.exe
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. The purpose of this option is to highlight, directly in annotated source, the problems (or the contributing incompatibilities) associated with that source file. There is no parser default, and the default set of candidate messages, as configured in the GUI when the option is turned on, is: -emhm 1500, 1501, 1506, 1507, 1508, 1509, 1510, 1512, 1513, 1515, 1516, 1517, 1520, 1525, 1526, 1527, 1528, 1529.

-en, -encoding

Syntax:	-encoding {*ASC* , *EUC*, *NEWJ*, *OLDJ*, *NECJ*, *SJ, UTF*}
Default:	-encoding *ASC*
Program:	all
GUI:	Analyzer Personality>Analysis Processing>Encoding
Function:	Specifies the character set. The default is ASC, representing ASCII encoding. The UTF option enables support for files encoded in UTF8/UTF16/UTF32 with or without a BOM.

-ex, -extensions

Syntax:	-extensions {*ANSIPC*, *PC*, *ASM*, *C++*, *DOLLAR, LONGLONG, ALL*}
Default:	no entry
Program:	qac.exe

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 the command line, you can enable all the extensions by setting the option to -ex+ or -ex all.

QA·C recognizes the following extension classes:

Extension	Behavior
ansipc	• Backslash (\) is permitted within filepaths. • The keywords __near, __far, __huge, __cdecl, __pascal, and __fortran are recognized and ignored. A warning message is issued.
PC	• The keywords near, far, huge, cdecl, pascal, and fortran are recognized and ignored. A warning message is issued.
asm	• Recognizes the assembler code directives #asm, #endasm and asm("string");. If assembly language within #asm 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++	• The reference operator & is recognized in declarations but has no semantic effect. • C++ style comments are allowed. Text between // and a new line is treated as a comment.
dollar	• $ is legal in identifiers.
LONGLONG	• Type “long long” is recognized 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 “long long” result in generation of message 1027 and the type is treated as a synonym for type “long”. • The rules which determine the type of a numeric integer are determined conform to ISO C99. Unsuffixed decimal values which are too large to fit in a signed long are of type “long long” rather than “unsigned long”, if they fit.

-fi, -forceinclude

Syntax:	-forceinclude *filename*
Default:	no entry
Program:	qac.exe
GUI:	Compiler Personality > Extensions > Additional Include Files
Function:	Causes *filename* to be implicitly included at the beginning of each file. This method of “force-including” a file can be sometimes a useful way of defining macros and other project settings.

-file

Syntax:	-file *filename*
Default:	no entry
Program:	errsum.exe
GUI:	N/A
Function:	Generates output to *filename*. If the full path of the file is not given, this file will be created in the path specified by QACOUTPATH or the -op option.

-forget, -forgetall

Syntax:	-forgetall *option*
Default:	no entry
Program:	all
GUI:	N/A
Function:	Resets previous configuration settings for *option*. Some options are cumulative in their action in that successive usage of the option adds extra settings rather than replacing the previous. See -i, -d, -n, -o, -su, -q, and –wc.

For example:

```
qac –via personality.p_a -forget i –iC:\mypath source.c
```

This command line will cause the analyzer to disregard any include paths previously defined and to use only the path C:\mypath. It will cause qac to disregard any include paths in personality.p_a.

-format

Syntax:	-format *string*
Default:	see below
Program:	viewer.exe
GUI:	Message Personality > Display > Message Format
Function:	Determines how warning messages will be displayed. When the –format option is used, –ref and –text options are ignored, and usage of the –line option is restricted. See Formatting Message Output of Chapter 10: Advanced Topics for a detailed explanation.

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
```

-fpc, -foldplainchar

Syntax:	-foldplainchar {+|-}
Default:	-foldplainchar-
Program:	qac.exe
GUI:	Compiler Personality > Data Types > Data Type Interpretation > Fold type "pointer to plain char" to "pointer to [un]signed char"
Function:	The -foldplainchar option operates in conjunction with the –unsigned option. If the -foldplainchar option is enabled, the type "pointer to char" will be treated as semantically equivalent to either type "pointer to signed char" or type "pointer to unsigned char", depending on how the signedness of plain char is configured with the -unsigned option.

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 initialize an array of type signed char with a literal character string (e.g. "ABC"). The only way to achieve this is to apply an explicit cast (signed char *) to the literal.

This distinction is often considered to be pedantic and it 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:	-help
Default:	no entry
Program:	all
GUI:	N/A
Function:	Displays help description for the program.

-hdr, -hdrsuppress

Syntax:	-hdrsuppress {+|-}
Default:	-hdrsuppress-
Program:	viewer.exe
GUI:	Message Personality > Display > Display Header Warnings
Function:	With –hdrsuppress+, all warnings detected in header files are suppressed, except for level 9 messages (which cannot be suppressed). The default setting will result in displaying all messages (not turned off through –n, –o, –q or other diagnostic suppression means) arising from parsing of header files.

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.

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:	-hiddenmessage {+|-}
Default:	-hiddenmessage-
Program:	errwrt.exe
GUI:	N/A
Function:	With –hiddenmessage+, a flag is set on the created error record to mark the diagnostic as suppressed.

This is equivalent to the behavior for messages generated under the control of #pragma PRQA_MESSAGES_OFF.

Notes:
Hidden messages can be displayed by utilities with the -hw+ option.

Comment-based suppressions or Baselining do not use this technique to mark warnings as suppressed: this is only used by the #pragma PRQA_MESSAGES_OFF alternative.

-hw, -hiddenwarnings

Syntax:	-hiddenwarnings {+|-}
Default:	-hiddenwarnings-
Program:	viewer.exe, errsum.exe
GUI:	Message Personality > Display > Show Suppressed Warnings
Function:	With –hiddenwarnings+, 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

Syntax:	-i *path*
Default:	no entry
Program:	qac.exe
GUI:	Analyzer Personality > Project Headers

	Compiler Personality > System Headers
Function:	Specifies a search path for header files.

-il, -indentlevel

Syntax:	-indentlevel *value*
Default:	-indentlevel 0
Program:	qac.exe
GUI:	Analyzer 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 inconsistent indentation and will issue message 2201.

-it, -intrinsictype

Syntax:	-intrinsictype {*size_t* | *ptrdiff_t* | *wchar_t*}=*datatype*
Default:	-intrinsictype ″size_t=unsigned int″ -intrinsictype ″ptrdiff_t=long″ -intrinsictype ″wchar_t=unsigned char″
Program:	qac.exe
GUI:	Compiler Personality > Data Types > Intrinsic Types
Function:	Determines the underlying type associated with the sizeof operator (size_t), the result of pointer subtraction (ptrdiff_t), and the wide character data type (wchar_t). 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.

-k+r

Syntax:	-k+r [*sun*, *a*, *b*, *c*, *e*, *g*, *i*, *l*, *n*, *p*, *s*, *t*, *u*, *v*, *x*]
Default:	no entry
Program:	qac.exe
GUI:	Analyzer Personality > K&R Compatibility
Function:	Identifies ISO language features that should be avoided if using a K&R compiler.

If you want your code to be compliant with 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 QA·C 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	Behavior
sun	enforces compatibility with the Sun K&R cc compiler under SunOS 4.1 or earlier. This is equivalent to: -k+r acegipstux.
a	initializing locally-declared structures and unions.
b	bit-field
c	string concatenation
e	#error
g	glue operator ##
i	signed keyword
l	#elif
n	enumerated types
p	function prototypes
s	stringify operator #
t	const and volatile
u	unary +
v	void keyword
x	initializing extern variables.

-l, -line

Syntax:	-line {+|-}
Default:	-line-
Program:	viewer.exe
GUI:	Message Personality > Display > Display Line Numbers
Function:	With –line+, the filename, line and column number are displayed with the warning message. If the –format option is specified, the –line option is ignored for message formatting. The Message Browser uses –line+ to enable the display of line numbers beside the source code.

-list

Syntax:	-list filelist.lst
Default:	no entry
Program:	viewer.exe, pal.exe, errsum.exe
GUI:	N/A
Function:	Passes a list of files to various utilities and external processes, including the Message Browser and the Project Warning Summary 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 Project Warning Summary report, and when used with the %L expansion key in Custom Reports and Cross-Module 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 so that the user can identify it readily.

-maxerr, -maxerrors

Syntax:	-maxerrors *value*
Default:	-maxerrors 0
Program:	qac.exe
GUI:	Analyzer 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:	-maxlinelength *value*
Default:	-maxlinelength 0
Program:	qac.exe
GUI:	Analyzer Personality > Style > Maximum Line Length
Function:	Specifies the maximum physical line length in the source code, before the line-splicing phase of translation. Permitted values are zero (for no checking of maximum physical line length), or any number 50 or greater. Using the GUI spinedit control, any value entered in the range from 1 to 49 will be automatically converted to 50, the minimum check value.

-met, -metrics

Syntax:	-metrics {+|-}
Default:	-metrics+
Program:	qac.exe
GUI:	Analyzer Personality > Metrics > Display Metrics
Function:	With –metrics+, analysis data is written to the metric output file.

	With –metrics-, a metric output file is generated but will be empty.

-n, -no, -nomsg

Syntax:	-nomsg *message_number_list*
Default:	no entry
Program:	viewer.exe, errsum.exe, qac.exe
GUI:	Message Personality > Advanced Settings > Save Message Settings By > Suppressed Messages
Function:	Suppresses the display of specified messages. qac.exe will not generate the specified warning messages in the err file.

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:	-namelength *value*
Default:	-namelength 31
Program:	qac.exe
GUI:	Compiler Personality > Identifiers > Internal Identifier Significant Length
Function:	Determines the number of characters in valid internal identifiers. QA·C will generate message number 779 for each internal identifier it encounters that is not distinct 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.

-nrf, -namerulefile

Syntax:	-namerulefile *rulefile*
Default:	no entry
Program:	pal.exe
GUI:	Message Personality > Advanced Settings > Secondary Analysis Setup
Function:	Identifies the file that contains rules for Naming Convention checking in the Secondary Analysis. See Naming Convention Checking of Chapter 4 and Appendix I.

-o, -only

Syntax:	-only *message_number_list*
Default:	no entry
Program:	viewer.exe, errsum.exe, qac.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: -o 2356 -o 1234,1236,1240 -o 2111,2200-2300 qac.exe will only generate the specified warning messages in the err file.

-op, -outputpath

Syntax:	-outputpath *path*
Default:	%QACOUTPATH% or no entry
Program:	all
GUI:	Folder Parameters>Output File Path
Function:	During analysis, the output path determines the location to which all output files will be generated. When displaying the Message Browser or generating the Project Warning Summary report, the output path determines the location of the .err files.

Note:
In command line usage, the output path is usually set by the QACOUTPATH environment variable, unless it is overridden by this option. If you intend to run QA·C concurrently (e.g. GUI and qac.exe), it is unsafe to rely on QACOUTPATH. It is recommended to specify this option explicitly.

-po, -prodoption df::analyze_header_functions

Syntax:	-po df::analyze_header_functions {+|-}
Default:	-po df::analyze_header_functions-
Program:	qac.exe
GUI:	Configuration > Dataflow Analysis Settings > Advanced > Analyze function definitions in header files
Function:	Add -po df::analyze_header_functions+ to perform dataflow analysis for function definitions seen in the common included header files. It may be preferable to turn off this option in order to reduce analysis overhead if this style of coding is in evidence.

-po, -prodoption df::function_timeout

Syntax:	-po df::function_timeout=*{value}*
Default:	no entry
Program:	qac.exe
GUI:	Configuration > Dataflow Analysis Settings > Advanced > Aggregate Function Timeout
Function:	Set an aggregate function timeout in milliseconds to limit the overall time spent in dataflow analysis for each function. This may curtail related results, and is designed to avoid lengthy analysis attempts on severely complex functions.

-po, -prodoption df::inter

Syntax:	-po df::inter={0|1|2|3|4|5}
Default:	-po df::inter=0
Program:	qac.exe
GUI:	Configuration > Dataflow Analysis Settings > Advanced > Inter-Function Analysis Depth
Function:	Sets the maximum amount that a function will be allowed to grow by as a result of inserting called function code into the caller as part of inter-function analysis. The values applied have the following specific meaning. When set to zero, no calls are expanded in analyzed functions. Otherwise, with a setting between 1 and 5, a function can grow by up to the following number of simplified statements when expanding called functions: df::inter=1 : 20 df::inter=2 : 200 df::inter=3 : 2,000 df::inter=4 : 20,000 df::inter=5 : 200,000 A simplified statement refers to an internal deconstruction of source-level statements which includes expansion of temporaries, deconstruction of compound statements according to sequence points, and other requirements of dataflow examination. Therefore the number of explicit statements in a function will nearly always be less than the number of simple statements. With df::inter=5, if the expanded set of simplified statements exceeds the limit of 200,000 message 2756 is generated.

-po, -prodoption df::map_old_messages

Syntax:	-po df::map_old_messages {+|-}
Default:	-po df::map_old_messages-
Program:	qac.exe
GUI:	Configuration > Dataflow Analysis Settings > Advanced > Map Old Dataflow Messages
Function:	With -po df::map_old_messages+, the analysis output includes an additional set of message numbers produced by an older heuristic version of dataflow analysis in

	QA·C 7.2 or earlier. Not all messages in dataflow have a direct mapping to these older messages, and this option is designed for those who wish to retain existing message configurations.

-po, -prodoption df::query_timeout

Syntax:	-po df::query_timeout=*{value}*
Default:	-po df::query_timeout=10000
Program:	qac.exe
GUI:	Configuration > Dataflow Analysis Settings > Advanced > Individual Query Timeout
Function:	Set an individual query timeout in milliseconds. Queries are an internal part of the dataflow analysis system, and several queries are typically run for each type of dataflow analysis performed. There can be many hundreds of queries performed for more complex functions. Limiting the time spent on queries will cut short longer ones and may affect accuracy of results.

-po, -prodoption df:: struct_last_array_member_size_1

Syntax:	-po df:: struct_last_array_member_size_1
Default:	no entry
Program:	qac.exe
GUI:	N/A
Function:	If the last member of a struct is an array of 1 element, the member is treated in a similar way to a C99 *flexible array member*, and the upper bound is not checked by Dataflow by default. If this option is specified, Dataflow will consider the size of this array to be 1, instead.

-ppf, -ppfilename

Syntax:	-ppfilename {+|-}
Default:	-ppfilename-
Program:	qac.exe
GUI:	Analyzer Personality > Analysis Processing > Include filename and line numbers in preprocessed listing
Function:	With –ppfilename+, 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 –pplist.

-ppl, -pplist

Syntax:	-pplist {+|-}
Default:	-pplist-

Program:	qac.exe
GUI:	Analyzer Personality > Analysis Processing > Preprocessed Output
Function:	With –pplist+, a preprocessed file is generated in the output directory with a .i file extension.

-ppm, -ppmetrics

Syntax:	-ppmetrics {+|-}
Default:	-ppmetrics-
Program:	qac.exe
GUI:	Analyzer Personality > Metrics > Metrics After Preprocessing
Function:	This option is designed to be used with file metrics, and governs the inclusion of header file information in the calculation of metrics: -ppmetrics+ results in all code/tokens in included header files being used when calculating metrics. -ppmetrics- results in code/tokens in included header files being ignored when calculating metrics.

Note:
The setting of this option is ignored by all function metrics except STLIN. Function metrics, STLIN excepted, always incorporate all code/tokens in the 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 {*filename* | *path*}
Default:	no entry
Program:	qac.exe, pal.exe
GUI:	Compiler Personality > System Headers > Suppress Output Analyzer Personality > Project Headers > Suppress Output
Function:	Suppresses the generation of output to the .err and .met 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.

Note:
This option should not be used for project header files or directories, as it may affect the results of CMA, Project Metrics and QA Reports.

-ref, -references

Syntax:	-references {+|-}
Default:	-references-
Program:	viewer.exe
GUI:	Message Personality > Display > Show References
Function:	With –references+, 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. If the –format option is specified, the –references option will have no effect. All formatting of messages is determined by the –format string, including the %v specifier for displaying verbose/reference text.

-rem, -remark

Syntax:	-remark *comment*
Default:	no entry
Program:	all
GUI:	N/A
Function:	With –remark, comments can be written into configuration files. Alternatively, comments can be included in the configuration files by prefixing the comments with an asterisk.

-s, -size

Syntax:	-size [*type*]=[*bits*]
Default:	See chart under option –align.
Program:	qac.exe
GUI:	Compiler Personality>Data Types>Type Size and Alignment>Size
Function:	Defines the type sizes (measured in bits) implemented in your compiler environment.

In the GUI, select the type for which you want to change the size and enter the new size in the *Size* box, or use the trackbar. Click *Apply* to confirm the setting.

The available size and alignment values are constrained by the ISO C rules on integral size ordering. The size is often related to the alignment of a data type, see the -align option for the full set of constraints on size and alignment.

-set, -settings

Syntax:	-settings {+|-}
Default:	-settings-
Program:	qac.exe, errsum.exe
GUI:	N/A
Function:	When -settings+ is supplied, the current values for all configuration options are listed to the 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 [*pragma_start*],[*pragma_end*]
Default:	no entry
Program:	qac.exe
GUI:	Analyzer Personality > Pragma
Function:	Defines the behavior 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 ...	ABC*		Ignores all #pragmas that

#pragma ABC2 ... #pragma ABC3 ...			begin with ABC.

Note:
QA·C also recognizes specific the #pragma directives of its own, e.g. 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:	Analyzer Personality > Analysis Processing > Implicit Conversion Message Options > Generate type conversion messages for equivalent types
Function:	The –strictrank option affects the behavior of implicit type conversion messages which report conversions from one integer type to a type of the same signedness but lower rank. These messages are referred to in the standard message file as “narrowing” conversions. When the strictrank option is off (-strictrank-), these messages are not generated when converting to a type which is of lower rank but is actually an “equivalent” type.

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 _Bool 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:	-strictsignedness {+|-}
Default:	-strictsignedness+
Program:	qac.exe
GUI:	Analyzer Personality > Analysis Processing > Implicit Conversion Message Options
Function:	Generate unsigned to larger signed messages for wider types. This option is obsolete: it has no effect on the ‘Essential Type’ Aritmentic Type messages. It is effectively always on to enforce the Essential Type system. The range of “Unsigned to Larger Signed” messages has been replaced with a single message 4443. The messages that were affected by this option are in Level 1, Obsolete Messages: Arithmetic Type – Implicit Conversions.

-st, -standard

Syntax:	-standard {+|-}
Default:	-standard-
Program:	viewer.exe
GUI:	Message Personality > Display > Display Highest Warning Level
Function:	Displays the highest warning level of a message. When set to –standard+, this option causes annotated source output to display the highest level warning message in the annotated source code. When set to –standard- and a detected warning occurs at more than one warning level, the lower level message is displayed.

A typical compliance module will include some messages at level 4 that are also present at level 8 in the message file. Typically, –standard+ is specified so that these messages are displayed as level 8 warnings.

-sty, -style

Syntax:	-style {k+r, exdented, indented}
Default:	-style exdented
Program:	qac.exe
GUI:	Analyzer 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 it will not enforce any particular depth of indentation, unless that is specified by using the –indentlevel option.

-su, -suppresslvl

Syntax:	-suppresslvl *message_level*
Default:	no entry
Program:	viewer.exe, errsum.exe
GUI:	Message Personality > Warning Messages > Message Groups
Function:	Suppresses all warning messages from a specified message level. To suppress a number of levels, select a range with a dash, for example, -su 4-7, or specify a selection as a comma-separated list, for example, -su 1,2,5.

Note:
The –suppresslvl option only operates on primary messages, and has no effect on sub-messages produced in tandem with enabled primary messages.

-t, -tab, -tabstop

Syntax:	-tabstop *columns*
Default:	-tabstop 8
Program:	qac.exe, viewer.exe
GUI:	Analyzer Personality > Style > Source Code Tab Spacing
Function:	Specifies the number of spaces represented by a tab character. Tab spacing is not related to the depth of indentation. QA·C will warn about inconsistent indentation, but it 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.

-thresh, -threshold

Syntax:	-threshold *metric*{<|<=|=|>|>=}*value*[:*msg_no*]
Default:	no entry
Program:	qac.exe
GUI:	Analyzer Personality > Metrics > Metric Thresholds
Function:	When specified, message *msg_no* is output whenever the threshold condition (from the set {<|<=|=|>|>=}) for specified *metric* is exceeded. The metric is specified by the name that appears in the metrics records of the .met file. See Metrics Records of Appendix F: Metric Output File. There are no default values, and if *:msg_no* is omitted, the default message 4800 is used.

-u, -unsignedchar

Syntax:	-unsignedchar {+|-}
Default:	-unsignedchar-
Program:	qac.exe
GUI:	Compiler Personality > Data Types > Char Interpretation

Function:	With –unsignedchar+, the char datatype is interpreted as unsigned char. With –unsignedchar-, it is interpreted as signed char.

The ISO C 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:	-usrpath [*path*]
Default:	%QACBIN%
Program:	viewer.exe, errsum.exe
GUI:	Message Personality > Advanced Settings > User Message File
Function:	Specifies the path of the user message file. If this option is not specified, viewing tools use the path defined by QACBIN. In the GUI, this option needs only be set if you specify a user message file in a directory other than the bin directory. QA·C first searches the path specified by the –up attribute and the –up setting needs to be right at the directory where the message HTML files are located. QA·C also uses a variable QACHELPFILES which is formed from the StartDir entry in qac.ini, with \help appended, and searches for message HTML within this.

-usr, -usrfile

Syntax:	-usrfile *extension*
Default:	no entry
Program:	viewer.exe, errsum.exe
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 use 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 qac.msg file.

-ver, -version

Syntax:	-version {+|-}
Default:	-version-
Program:	all
GUI:	N/A

Function:	With –version+, the version number for the executable is displayed.

-via

Syntax:	-via *file*
Default:	no entry
Program:	all
GUI:	N/A
Function:	Specifies a file containing configuration options.

-wc, -warncall

Syntax:	-warncall [*function*[=*msg_no*]]
Default:	no entry
Program:	qac.exe
GUI:	Analyzer Personality > Warning Calls
Function:	Displays warning message *msg_no* if QA·C encounters a call to *function*.

If no message number is declared, the following message is displayed:

```
Msg(3:2010) The function 'func_name' must not be called.
```

-xc, -xcase

Syntax:	-xcase {+|-}
Default:	-xcase-
Program:	qac.exe
GUI:	Compiler Personality > Identifiers > Ignore External Identifier Case
Function:	The C90 Standard allows an implementation to ignore alphabetical case in external identifiers. With –xcase+ this restriction is strictly imposed. See –xnamelength for implications of external identifier significance.

-xn, -xnamelength

Syntax:	-xnamelength *value*
Default:	-xnamelength 6
Program:	qac.exe
GUI:	Compiler Personality > Identitifers > External Identifier Significant Length
Function:	Determines the number of significant characters in an external identifier.

The C90 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 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.

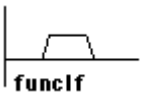
Straight code


```
static int code = 0;

void funcStraight (void)
{
    code = 1;
}
```

There is only one path through this particular function, which is represented as lying along the x-axis.

If



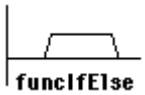

```
static int code = 0;

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



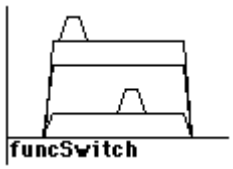

```
static int code = 0;

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.

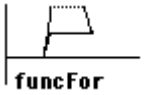
While loop


```
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;
    }
}
```

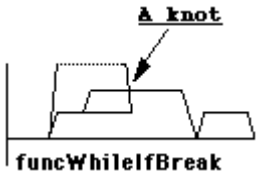
Nested Structures


```
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

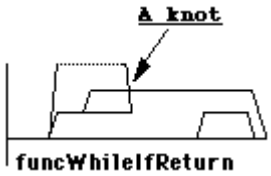



```
static int code = 0;

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

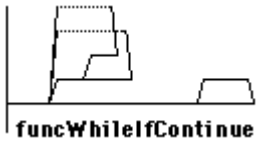



```
static int code = 0;

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.

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 red 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 analyzed. Project metrics are calculated once per complete project, and are generated from cross-module analysis.

All metrics, except STLIN and STTPP, ignore code that is disabled on evaluation of pre-processor directives. That is, both the code in the source file and the #include'd files inside conditional pre-processor directives are ignored for all metrics except STLIN and STTPP. See Appendix B (-ppm) for an example.

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. 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 Statement in Function (variant 1)
STAV2	Average Size of Statement in Function (variant 2)
STAV3	Average Size of Statement in Function (variant 3)
STBAK	Number of Backward Jumps
STCAL	Number of Functions Called from Function
STCYC	Cyclomatic Complexity
STELF	Number of Dangling Else-Ifs
STFN1	Number of Operator Occurrences in Function
STFN2	Number of Operand Occurrences in Function
STGTO	Number of Goto statements
STKDN	Knot Density
STKNT	Knot Count
STLCT	Number of Local Variables Declared
STLIN	Number of Code Lines
STLOP	Number of Logical Operators
STM07	Essential Cyclomatic Complexity
STM19	Number of Exit Points
STM29	Number of Functions Calling this Function
STMCC	Myer's Interval
STMIF	Deepest Level of Nesting
STPAR	Number of Function Parameters
STPBG	Residual Bugs (STPTH-based est.)
STPDN	Path Density
STPTH	Estimated Static Program Paths
STRET	Number of Return Points in Function
STST1	Number of Statements in Function (variant 1)
STST2	Number of Statements in Function (variant 2)
STST3	Number of Statements in Function (variant 3)
STSUB	Number of Function Calls
STUNR	Number of Unreachable Statements
STUNV	Unused or Non-Reused Variables
STXLN	Number of Executable Lines

STAKI Akiyama'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 Akiyama[2] and Shooman[3] for more details. The metric is calculated as:

STAKI = STCYC + STSUB

STAV*x* Average Size of Statement in Function

These metrics (STAV1, STAV2, and STAV3) measure the average number of operands and operators per statement in the body of the function. They are calculated as follows:

STAV*x* = (N1 + N2) / number of statements in the function

where:
N1 is Halstead's number of operator occurrences.
N2 is Halstead's number of operand occurrences.

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:

STAV1 = (STFN1 + STFN2) / STST1
STAV2 = (STFN1 + STFN2) / STST2
STAV3 = (STFN1 + STFN2) / 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.

[2] Akiyama, F. (1971) *An Example of Software System Debugging*, Proc. IFIP Congress 1971, Lbujlana, Yugoslavia, American Federation of information Processing Societies, Montvale, New Jersey.
[3] Shooman, M.L. (1983) *Software Engineering*, McGraw-Hill, Singapore.

STCAL Number of Functions Called from Function

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.

McCabe[4] 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 it is conventionally written without additional indentation and so the nesting is not visually apparent.

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;
    }
}
```

[4] McCabe, T. J. (1976) *A Complexity Measure*, IEEE Transactions on Software Engineering, SE-2, pp. 308-320.

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. Instead, STCYC calculation is based on statements alone.

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 for which QA·C issues a warning 2004. STELF 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 Operator Occurrences in Function

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 summarized 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 Operand Occurrences in Function

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 summarized 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 Goto statements

Some occurrences of goto simplify error handling. However, they should be avoided whenever possible. According to the Plum Hall Guidelines, goto should not be used.

STKDN Knot Density

This is the number of knots per executable line of code. The metric is calculated as:

STKDN = STKNT / STXLN

The value is computed as zero when STXLN is zero.

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 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 of storage class auto, register, or static declared in a function. These are variables that 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 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. 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 McCabe[5] 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 subgraph possesses multiple entry or exit points then it cannot 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 */
```

[5] McCabe, T. J. (1976) *A Complexity Measure*, IEEE Transactions on Software Engineering, SE-2, pp. 308-320.

```
    int nlines = 0;
    while ( --n >= 0 )
    {
        pos = back_line( pos );
        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). Calls to non-returning functions 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 high value of L indicates that there are many 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;
    }
}
```

Note:
In the calculation of STMCC, the ternary operator (?:) is ignored.

When exporting metric values or displaying in the Metrics Browser, rather than attempting to display a value pair, the value of L is chosen for STMCC.

STMIF Deepest Level of Nesting

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. The nesting level of code is not always visually apparent from the indentation of the code. 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 three 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 Residual Bugs (STPTH-based est.)

Hopkins, in Hatton & Hopkins[6] 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.

$$STPBG = \log_{10}(STPTH)$$

STPDN Path Density

This is a measure of the number of paths relative to the number of executable lines of code.

STPDN = STPTH / STXLN
STPDN is computed as zero when STXLN is zero.

STPTH Estimated Static Program Paths

This is similar to Nejmeh's[7] NPATH 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 NPATH value for a sequence of statements at the same nesting level is the product of the NPATH values for each statement and for the nested structures. NPATH is the product of:

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

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 */
}
```

Since NPATH cannot be reliably computed if there are *goto* statements in the function, they are ignored by QA·C for the purposes of calculating NPATH.

The following code example has a static path count of 26.

```
int n;
if ( n )
{
} /* block 1, paths 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] Nejmeh, B.A. (1988), *NPATH: A Measure of Execution Path Complexity and its Applications*, Comm ACM, 31, (2), p. 188-200.

```
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:

$$\text{cyclomatic complexity} \leq \text{true path count} \leq \text{static path count}$$

Static path count is one of the three standard metrics used by QA·C for demographic analysis.

STRET Number of Return Points in Function

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 should 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 the functions that do not follow the Structured Programming paradigm. For example, those with switch statements with returns in many or every branch.

STST*x* 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, see STCAL.

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;
    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

[8] Brandl, D.L. (1990), *Quality Measures in Design*, ACM Sigsoft Software Engineering Notes, vol 15, 1.

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 or Non-Reused Variables

An unused variable is one that has been defined but which is never referenced. A non-reused variable is a variable that has a value by assignment but which is never used subsequently.

Such variables are generally clutter and are often evidence of “software ageing”, which is the effect of a number of programmers making changes. The code below has 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 (TU). They are listed alphabetically here for convenience. They are not calculated in this sequence. because of their derivation from each other:

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

STBMO Organic Programmer Months
STBMS Semi-detached Programmer Months
STBME Embedded Programmer Months

The COCOMO metrics are produced for each source code file. You can display an 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 estimate the number of programmer-months required to create the source code in the respective environments.

STBME = 3.6 * (STTPP / 1000) 1.20
STBMO = 2.4 * (STTPP / 1000) 1.05
STBMS = 3.0 * (STTPP / 1000) 1.12

STBUG Residual Bugs (token-based estimate)
$STBUG = 0.001 * STEFF^{2/3}$

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 Hopkins[9].

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 value of STCDN indicates that there may be too many comments, which can make a module difficult to read. A small value indicates that there may not be enough comments, which can make a module difficult to understand.

The code below has 28 visible characters in comments and 33 visible characters in the code. The resulting STCDN value is 0.85.

```
int test()
/* This is a test */
{
    int x;
    int y;
/* This is another test */
    return (x + y);
}
```

The value of STCDN is affected by how QA·C counts the comments. QA·C can count comments in three possible 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

[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.

line of code (comments that are on the same line as code at file scope).

You can determine which counting method is used in Comment Count on the Metrics tab of the Analyzer Personality or by setting the –co option on the command line.

STDEV Estimated Development (programmer-days)

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.

STDIF = STVOL / ((2 + STVAR) * $\log_2$ (2 + STVAR))

STECT Number of External Variables Declared

This is a measure of the number of data objects (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 Programmer 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.

STEFF = STVOL * STDIF

STFNC Number of Functions in File

This metric is a count of the number of function definitions in the file.

STFCO Estimated Function Coupling

See Brandl[10]. 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 from STFNC and the STSUB values of the component functions in the translation unit:

STFCO = ∑(STSUB) – 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:

STZIP = (STOPN+STOPT) * (0.5772+ln (STOPN+STOPT))
STHAL = STOPT * $\log_2$ (STOPT) + STOPN * $\log_2$ (STOPN)

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.

[10] Brandl, D.L. (1990), *Quality Measures in Design*, ACM Sigsoft Software Engineering Notes, vol 15, 1.

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. A file header is defined as tokens preceeding the first code token or preprocessor directive 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;
}
```

STOPN Number of 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 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 Number of Distinct Operators

This covers any source code tokens not supplied by the user, such as keywords, operators, and punctuation. STOPT is used in the calculation of a number of other metrics.

The code 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 -> } */
```

STSCT Number of Static Variables Declared

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 recognized 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:

$STSHN = STZIP * \log_2 (\sqrt{(STOPN+STOPT)} + \ln (STOPN+STOPT))$

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 elapsed time in months required to develop the source code in the respective environments.

$STTDE = 2.5 * STBME^{0.32}$
$STTDO = 2.5 * STBMO^{0.38}$
$STTDS = 2.5 * STBMS^{0.35}$

STTLN Total Preprocessed Code Lines

This metric is a count of the total amount of lines in the translation unit after pre-processing. The pre-processed file will reflect the processing of include files, pre-processor 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 Unpreprocessed Code Lines

This metric is a count of the total number of source lines in the file before pre-processing.

STVAR Total Number of Variables

The metric represents the total number of distinct identifiers. The code 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. It is used to calculate various Halstead vocabulary metrics.

The following is the calculation for the program volume:

STVOL = STTOT * $\log_2$ (STOPN + STOPT)

STZIP Zipf Prediction of STTOT

STZIP = (STOPN+STOPT) * (0.5772+ln (STOPN+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 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 final column indicates the executables to which each return code applies.

Analysis Status Reported	Return Code	Explanation	Notes
Completed	0	Successful completion	(1)
Failed: parser hard error	1	Hard errors were encountered during analysis #error directive encountered	(2)
Failed: parser configuration	2	#include file not found Recursive calls to files No end to comment block	(2)
Failed: configuration	10	general configuration problem missing internat.rsc file	(1)
Failed: fatal error	19	Memory allocation error, or untrapped exception	(1)
Failed: licensing error	11	Product licensing failure	(2)
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, errwrt.exe, and errsum.exe)

(2) Applicable to qac.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
```

This record is generated whenever a #include statement is encountered. The *includingfile* may be either a source file or another included file.

```
<R>C callingfunction  calledfunction lineno
```

This record is generated for every instance of a function call with either internal or external linkage.

```
<R>X callingfunction  identifier  lineno  flag
```

This record is generated for every reference to an identifier with external linkage. It 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
```

This record is generated for every definition of an object with external linkage.

<EXTN> external_identifier filename lineno

This record is generated for every declaration of an object with external linkage.

<EXTV> identifier external_identifier lineno

This record is generated for every definition of an object which is initialized using an object with external linkage.

<EXTT> identifier filename lineno

This record is generated for every tentative definition of an object. A tentative definition of an object is generated by declaring an identifier at file scope without an initializer and without a storage class specifier, or with the storage-class specifier static. The first tentative definition is followed by an <EXTN> record.

Example:

Line Number	File Scope Declarations	Met File External Entries
1	int count = 0;	<EXTD> count file.c 1
2	int *pcount = &count;	<EXTD> pcount file.c 2
		<EXTV> pcount count 2
3	extern int major;	<EXTN> major file.c 3
4	static int ifunc();	
5	extern int efunc();	<EXTN> efunc file.c 5
6	int (*pfi)() = ifunc;	<EXTD> pfi file.c 6
7	int (*pfe)() = efunc;	<EXTD> pfe file.c 7
		<EXTV> pfe efunc 7
8	int tdef;	<EXTN> tdef file.c 8
		<EXTT> tdef file.c 8
		<EXTD> 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
identifier		Identifier name (“-“ for unnamed bit-field)
filename		File in which declaration appears
lineno		Line at which declaration appears
inc	I	Included file
	N	Not included, i.e. main source file
def	DF	Definition
	DC	Declaration
	DR	Re-declaration
	DI	Implicit definition
space	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
scope	N	Function scope (for labels)
	F	File scope
	B	Block scope
	P	Function prototype scope
	-	Not applicable (macros)
linkage	I	Internal
	X	External
	N	none – local or typedef
	-	not applicable (macros)
Type	See table below	Symbolic type code, a shorthand notation used to describe a datatype.
Flag	F	Function-like macro
	O	Object-like macro
	S	static storage duration (local and global declarations. Not the same as static keyword.)
	R	Function parameter
	-	no further information

Symbolic Type Code	**Datatype**
nc	signed char
uc	unsigned char
_c	char
ns, ni, nl, nll	short, int, long, long long
us, ui, ul, ull	unsigned short, unsigned int, unsigned long, unsigned long long
fs, ff, fl	float, double, long double
pX	pointer to X
(Y,A1,..,An)	function returning Y taking A1.. An
[N,X]	N-element array of X
g	pointer to void

.	void, as in ' void f(int)' or 'int f(void)'
{sNAME}	struct NAME
{uNAME}	union NAME
{eNAME}	enum NAME
ne	enumeration constant
nb	signed bit-field
ub	unsigned bit-field
:	ellipsis, as in 'void f(int, ...)'
=X	const X
^X	volatile X

Example source code:

Line Number	Source Code	Met file <DEFINE> entries
1	#define M(x) ((x) + 10)	<DEFINE> M file.c 1 N DF OM - - - F
2	int a = 0;	<DEFINE> a file.c 2 N DC OV F X ni S
		<DEFINE> a file.c 2 N DF OV F X ni S
3	extern int b(char *c);	<DEFINE> c file.c 3 N DC OV P N p_c R
		<DEFINE> b file.c 3 N DC OF F X (ni,p_c) -
4	typedef double D;	<DEFINE> D file.c 4 N DC OT F N ff -
5	static int e(D f);	<DEFINE> f file.c 5 N DC OV P N ff R
		<DEFINE> e file.c 5 N DC OF F I (ni,ff) -
6	extern int g;	<DEFINE> g file.c 6 N DC OV F X ni S
7	static float h (int i)	<DEFINE> i file.c 7 N DC OV B N ni R
		<DEFINE> h file.c 7 N DC OF F I (fs,ni) -
		<DEFINE> h file.c 7 N DF OF F I (fs,ni) -
8	{	
9	int j;	<DEFINE> j file.c 9 N DC OV B N ni -
		<DEFINE> j file.c 9 N DF OV B N ni -
10	struct k	<DEFINE> k file.c 10 N DC TG B N {sk} -
		<DEFINE> k file.c 10 N DF TG B N {sk} -
11	{	
12	int l;	<DEFINE> l file.c 12 N DC MB B N ni -
13	char m;	<DEFINE> m file.c 13 N DC MB B N _c -

14	} n;	<DEFINE> n file.c 14 N DC OV B N {sk} -
		<DEFINE> n file.c 14 N DF OV B N {sk} -
15	j = i + g;	
16	return(j);	
17	}	

Control Graph Records

Example:

```
<CTLG> functionname 1
<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 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 <S>STNAM record is generated for each file or function, containing its name, before outputting its metrics.

A <S>STFIL record is also generated for each file or function, containing the name of the analyzed 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
```

This record is generated for all #pragma directives.

Literal Records

This record type identifies all constants used in the source code.

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 in the following table:

Field	Content	Description
literal		Literal reproduced verbatim, excl suffix
suffix		Suffix reproduced verbatim
sign	P	Positive
	N	Negative
	-	Not applicable
filename		Fully qualified filename
lineno		Line number in source code
colno		Column number in source code
inc	I	Included file
	N	Not an included file
lit-type	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”)
type		Symbolic type code
scope	F	File scope
	B	Block scope
	P	Prototype scope
context-flag	M	macro constant
	D	array dimension
	B	Bitfield size
	E	enum constant
	O	scalar object initializer

	A	array initializer
	S	Structure initializer
	U	union initializer
	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 customized 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> 155 314 (close pairs versus total names)
<CLOSENESS> 0.003154
```

<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_sort | r_uniq | r_close -report
```

will produce the following output:

The following identifiers might cause confusion because of their similarity:

```
arg1 arg2
bdiag diag
bdiag fdiag
bmax dmax
bmax fmax
bmid bmin
bmid fmid...
```

34 pairs of close names were found in 239 names.

The "closeness" metric is 0.001195.

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:

```
cat alloca.c.met | r_grep -p"<DEFINE>" | r_fields +4 +1 +7 | r_uniq
```

will extract the line number, identifier and name space as follows:

```
30      SCCSid  OV
51      pointer OT
54      NULL    OM
56      free    OF
57      xmalloc OF
70      STACK_DIRECTION OM...
```

You could produce the same results by removing the unwanted fields. For example:

```
cat alloca.c.met | r_grep –p”<DEFINE>” | r_fields  -0 -2 -3 -5 -6 -8 -9 -10 -11 | r_uniq
```

will produce:

```
SCCSid  30      OV
pointer 51      OT
NULL    54      OM
free    56      OF...
```

These fields could be rearranged by adding:

```
| r_fields +1 +0 +2
```

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.c.met
```

will display the function name and path count metrics for the file alloca.c.met as follows:

```
<S>STNAM  find_stack_direction(.,)
<S>STPTH  3
<S>STNAM  alloca(puc,ui)
<S>STPTH  12
<S>STNAM  alloca.c
```

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.c.met | r_grep –v –p”alloca.c”
```

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 that copies standard input to standard output, omitting the duplicated lines. The input must be sorted.

APPENDIX H

Code-based 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 cross-module 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 its use is deprecated.

Atomics of Diagnostics

All diagnostic messages have as basic information the 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:

tag_name	is a special tag that can be used by other annotations to refer to this location.
location_specifier	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, the suppression applies from the current line until the first location tag appearing in the current file.

Continuous Suppression Syntax

A special syntax is provided as a direct replacement to the deprecated #pragma PRQA_MESSAGES_[ON|OFF] syntax. 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:

message specifier	represents the set of messages this suppression should apply to.
++	specifies that the set of messages in message_specifier be added to the current set of suppressed messages from this line onwards.

--	specifies that the set of messages in message_specifier be removed from the current set of suppressed messages 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 behavior 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 behavior, 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.

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: All instances of message 100 will be suppressed on the line of the declaration of i.

Suppression of a blank line

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

Effects: All instances of message 100 will be suppressed 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, e.g. 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 starts and ends 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 correct.

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 the 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 foo.h, 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 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 2

// 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 comment as above about suppressing i also 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 while processing 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
// PRQA L:A_1
```

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.

Typically, the Name Checker is configured to run from the GUI, but it can also be run from the command line. It analyzes 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 inputs:

- Source file and corresponding .met and .err files from Primary Analysis.
- A configuration file specifying a set of naming rules – this is normally specified in *Message Personality > Secondary Analysis set-up*.

It will also be necessary to provide any additional messages mentioned in the configuration file, through a user message file. Message 4800 is reserved in the standard message file as a default configuration entry.

Configuration File

This file, also referred to as the name rules file, contains the rules to be used by the Name Checker. As a minimum, each rule should specify the pattern (Regular Expression) that is to be applied to each type of identifier (macros, functions, typedefs, variables, etc). You can have different rules for different types of identifier.

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 http://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 hash (#) are ignored. Each rule must be on a separate line, and may not be split across two lines.

All the Rule Values are strings, except for the following:

- Invert is a Boolean (specified with 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.

Filename

```
Name: filename
Value: string, regular expression
```

This object is used to specify particular files to that the rule applies to. It is a Regular Expression. Only a file whose name matches the regular expression has 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 to 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 main 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

When omitted, the rule applies to all definitions and declarations. 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
TG	structure, union or enumeration tag
MB	member of structure or union
OT	typedef
OF	function
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
F	file
B	block
P	function prototype

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 utilizing a symbolic shorthand notation as described in Appendix F: Metric Output File. A few examples are given here:

Value	Description
nc	signed char
uc	unsigned char
ni	signed int
g	pointer to void
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 an identifier with any flag value, including none.

Value	Description
F	function like macro
O	object-like macro
S	static storage duration
R	function parameter

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. 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, the results of the matches will not be inverted.

Message Number

```
Name: message
Value: number
```

This specifies the message number to issue if the identifier fails the pattern match. If this is omitted, the default message (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.

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 (which we simply refer to as Regular Expressions) and you are 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 letters 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

The default is 1: that is, where no number of matches is specified, the pattern must match exactly once. Otherwise, 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, but 'velocity' does not match.

'[AEIOU][a-z]+' matches any 2 letter or longer word starting with a capital vowel followed by lower case letter(s). So 'Add' 'An' 'Egg' all 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 times, 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 our 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 such as 'i'. The Regular Expression can be constructed to match those 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, it 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 '^.[$()|*+?{\'.

For instance, the parenthesis may be needed in a type pattern, as this designates a function, or the open square bracket may be needed in order to indicate that the type is an array. But in the Regular Expression, these characters will be interpreted as having their special meaning. This means that the desired matching will not occur.

To overcome this, the special characters must be escaped with a backslash '\'. However, the backslash itself needs to be escaped.

So, to specify a left parenthesis in a Regular Expression, use '\\(' (that's two backslashes and the parenthesis). To specify a backslash, use '\\\' (three backslashes).

Configuration File Example

A simple example file is shown below along with a description for each rule:

#1 internal function identifier consists of words separated by _ and with initial letter capitalized
```
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 with b_
```
rule={'space':'OV','scope':'B','linkage':'N','pattern':'^[ij]$|^(b_.*)$','message':48
      00}
```

#3 macros must be all capitals, underscores are 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

The error log is registered in a NameCheckErrors.txt text file, which is written in the output directory. If a project has more than one output directory, the same file will appear in each output directory corresponding to a 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 given in the Message Personality. Check that the correct filename has been given 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 entries 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 Rule Names in the Appendix I 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. 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, which is denoted by ‘X’, rather than ‘E’.

Regular Expressions

The Regular Expressions for filename, pattern and type are compiled as the configuration file is processed. Any errors that occurred will be reported.

For example:

```
Line 29 Pattern Regular expression '[[:ower:]]*' compile fail: bad class at offset 3
   {'pattern':'[[:ower:]]*'}
```

The offset shows the location of the problem in the Regular Expression. In some cases, such as mismatched brackets, problems are reported at the end of the string.

Appendix J

Baselining Details

For an overview of baselining, see Chapter 5.

How to Generate a Baseline

There is more than one method of generating a baseline. The menu option is simplest to use. However, this method is not as flexible as either the Custom Report method or the command-line method.

In all cases, you need to consider whether you will need a backup of your baseline source for use when creating the difference information later. You can take your original source from a version control system if you have one, or you can keep a backup copy of your original source. The complete source tree is needed, starting from the root folder default source path.

Before the complete baseline is generated initially, the baseline source should be completely analyzed, including secondary analysis and cross-module analysis.

In some cases, you may want to baseline certain files only, rather than the complete source. If this is the case, before generating the baseline, use the menu item *Analyze* > *Remove Analysis Output* to remove the analysis output of the files you do not wish to be included in the baseline. Only files with current analysis output will be included in the baseline.

Generating a Baseline via Menu

With the default configuration settings, using the QA·C menu item *Analyze* > *Generate Analysis Baseline* will

- generate a suppression file in the project root folder source path called `project_name.sup`;
- create a log file called `project_name.log` in the same place;
- add an entry called BaselineFile to the project file, giving the full path to the suppressions file.

Generating a Baseline via Custom Report

If you are setting up a Custom Report or using the command line, you will need to supply the path and name of the executable program, and the appropriate parameters (see below).

The name of the executable program is `genBaseline.exe`, and it is installed in the bin directory.

When you have a suitable Custom Report set up, selecting it from *Reports* > *Custom Report: "project"* starts the baseline generation.

Generating a Baseline from the Command Line

The command is

```
genBaseline QAC <parameters> -list <filename>
```

where:

<parameters> is the parameter list: see table and text below.

<filename> is the list file for the analysis that is being baselined.

Editing the Command Line

If a path contains spaces, the string must be enclosed in double quotation marks ("…").

Quotes within quotes will need to be escaped, for example:

```
"value with \"quotes\""
```

Example:

Suppose that the path to the `prqavcs` helper application contains spaces. For instance, say that the full path and filename is

```
C:\Program Files\PRQA\QAC\bin\prqavcs.exe
```

As this path contains a space, it must be quoted when prqavcs is called on the command line, thus:

```
"C:\Program Files\PRQA\QAC\bin\prqavcs.exe"
```

Nested quotes will be needed when using prqavcs as a parameter to –ident, for example:

```
-ident "\"C:\Program Fil[…]in\prqavcs.exe\" -status"
```

Shortcuts

There are some useful shortcuts available, which can save you much of the typing that is otherwise needed for command-line entry.

Two of these are widely used elsewhere by PRQA tools, namely %Q for the product and %L+ for the filelist. In addition, for baselining on a Windows platform, you can use %D, %O and %J, where:

%D is the default path to the project root source folder.
%O is the default path to the project root output folder.
%J is the project name.

Table of Baseline Parameters

Parameter	Long form (if any)	Description
%Q		the product type
-sf	-suppressionfile	the full path to the suppression file
-sop	-sourcepath	the root of the baseline source tree
-po		an optional log file for events from the generation process
%L+	-list	the list of baseline source files and their paths
-ident		the identifier of the source being baselined
-sdt	-suppressiondifftool	to provide difference information

-sop

The root of the baseline source tree should match that of the working version of source. The path specified here is used to generate a non-development-specific location within the suppressions file. This means that the suppressions file can be shared by different users with different development locations.

Unless you are using a version control system, –sop is a required parameter.

-po

Note that this parameter has more than one use:

(1) When generating a baseline, this parameter can be used to specify a logfile:

```
-po baseline::log="<filename>"
```

Using a dash for the <filename> will cause the log to be sent to STDOUT.

(2) When diagnostics have been concealed by the Hide/Show Messages facility, they are not baselined, just hidden from view in the Message Browser. This parameter can be used if you wish to apply the hidden diagnostics directly to the suppressions file.

```
-po baseline::GenFromHoc (case-sensitive)
```

Any diagnostics in those files output from the last analysis which appear as *hidden* will be baselined, but those diagnostics which appear as *shown* will remain active.

-ident

This parameter has two modes of operation. You can use either

```
-ident "<cmd>"
```

or

```
-ident "::tag"
```

(1) With `-ident "<cmd>"` the process indicated by `"<cmd>"` will be run to obtain the source file identifier. For example, the command could query a version control system to obtain the version:

```
-ident "prqavcs -status"
```

You can use the shortcut %BF as part of the command, and this will be replaced with the full path to the source filename.

(2) The second mode of operation `-ident "::tag"` simply uses the specified tag as the identifier.

-sdt / -suppressiondifftool

This parameter supplies information related to the differencing function. The replacements listed below can occur when this parameter is used.

$ROOTPATH is an identifier used internally.

Parameter	Replaced by	Example
%BF	full path and filename	C:\devel\proj\src\file.c
%Bf	filename	file.c
%BD	full path	C:\devel\proj\src
%BR	full path and filename with rootpath applied	$ROOTPATH\proj\src\file.c
%Br	full path with rootpath applied	$ROOTPATH\proj\src
%BT	user-specified identifier supplied by `-ident`	BASELINE_TAG
%BB	the path and filename with rootpath omitted	proj\src\file.c
%Bb	the path with rootpath omitted	proj\src

Note that, since diffs are done on files relative to the root folder default source path, if you change this setting on the working project, the diff will no longer be correct and baselining will not operate correctly.

-updateerr

This optional parameter, available on the command line, can be used to modify the .err files during the baseline generation process. The default is that baseline suppression records are not added to the .err file when the baseline is generated.
To override the default, use

```
-updateerr+
```

Help

To produce this help information from the command line, navigate to the directory where `genBaseline.exe` has been installed and enter:

```
genBaseline -help
```

on the command line.

Generating a Baseline via Secondary Analysis

The above methods of baseline generation handle a complete project including CMA output. In the later stages of your project development you may need to update baseline data for a single source file or a small part of a project. Rather than recreating the baseline for a complete project, you can recreate the relevant part using Secondary Analysis.

To do this, set up a Message Personality with the Secondary Analysis settings for the executable and parameters as indicated in the Table of Parameters. Apply this to your current folder, or set of folders. If necessary, apply it to the root folder, and then to all sub-folders using the *Edit>Propagate Changes to Subfolders* menu item.

A Message Personality which is used to generate a baseline will differ from the one you use for interactive update and re-analysis of your code. Once your baseline has been generated satisfactorily, you should revert to your normal Message Personality.

Modifying Source

Copying the Original Source and Baseline

If you are not using a Version Control System, then after the initial baseline is generated, there is one more action to perform before you can start working on changes to your source. Copy the entire source tree (starting from the root folder default source path) to a safe location. This copy will include the newly-generated suppressions file, so use the *File* > *Project Properties* menu item to set the new location of the suppressions file.

At this point, you should delete the suppressions file from its original location.

Now you can modify your source code as usual.

Synchronizing the Baseline Data with Working Source

It is necessary that the baseline data is maintained so that it is in line with your changes to your source. This is done by adjusting the recorded locations of the baselined messages to match the corresponding locations in the new source file.

The original and current versions of source and the original baselined suppressions are used together. Using a differences utility between the two source versions, the original baseline entries have their locations recalibrated and appended to the .err records for each source file.

Applying the Baseline to Working Source

If you are applying baselining through the Windows GUI and there is a suppressions file selected in the Project Properties, this process will be semi-automatic.

Should you need or wish to use the command-line instead of the GUI, the name of the executable is suppressionTransform.exe. Entering this on the command-line followed by –help will display the appropriate help information, but the usage and parameters are similar to those applicable to generating the original baseline.

You can specify a suppressions file in *File* > *Project Properties* and this file will be applied to all file analysis, automatically, at the end of any secondary analysis process.

The baseline entry is specified in the project file as

```
BaselineFile=<path_to_baseline_sup_file>
```

The entry uses Relative Path and Environment Variable operation, although using a relative path is slightly non-intuitive. The baseline is intended to belong to a different version of code and when using the copy source approach the suppressions file is normally placed at the root of that other code version.

The default path of the project's Root Folder in the GUI must contain the location of the root of the source tree.

Modifying the Baseline

This is generally carried out through what is called Baseline Admin Mode. It can also be done after using the Hide/Show Messages facility.

Hide/Show Messages

If you wish to hide some diagnostics from view temporarily, without including them in the baseline, you can do it with Hide/Show messages, which only affects the visibility of the messages within the Message Browser.

If you right-click on a message group in the active list, you are offered the option to *Hide* it, that is, to remove the group from the active list. To make a Hidden group visible in the *Active MG* list again, right-click on the group where it appears under the *Messages Group* tab, and accept the option to *Show* message.

Under the menu item Suppressions > Display, you can toggle the visibility of messages suppressed in the Baseline, by Comment based annotation in code, or those which are hidden through Hide message.

However, if you would like to baseline some diagnostics which you have hidden using the Hide/Show functionality within the Message Browser, you should use the command line switch –po baseline::GenFromHoc when calling genBaseline.

Baseline Admin Mode

Baseline Administration Mode is used to make manual modifications to the suppressions file - and it follows that you cannot use Baseline Admin Mode until you have generated a suppressions file.

This mode can be reached from the main GUI *File* > *Project Properties* and also from the Message Browser's *File* > *Baseline Admin Mode* menu items.

You have the opportunity to browse to the suppressions file required. You can also specify the location of the source code. The default is that it is in the same folder as the suppressions file.

If you are using a Version Control System, you will need to have checked out the original baselined source code.

Restrictions

The Hide/Show messages facility is not available in the Baseline Admin Mode.

No parameter or sub-message information is stored in the suppressions file, which could give rise to the appearance of duplicate diagnostics. To avoid confusion, diagnostics differing only by parameter are shown in the suppressions file "de-duplicated" to a single instance. Neither parameter information nor sub-message information is shown in the Baseline Admin Mode.

Marking Message Groups

To mark a suppressed (baselined) group for re-activation, right-click on it to show the option to *Remove from baseline* and then left-click on the option. The procedure to mark an active group for suppression is similar.

Updating the Baseline

When you have finished marking groups, the baseline will need to be updated with your changes. This is done from the Baseline Admin Mode screen, using the menu item *Suppressions* > *Update Baseline*.

If you just try to close Baseline Admin Mode, omitting the updating stage, you will be asked if you wish to save your changes.

- *Proceed* will save your changes and close Baseline Admin Mode.
- *Discard* will discard your changes and close Baseline Admin Mode.
- *Cancel* will cancel the instruction to close Baseline Admin Mode, leaving your changes as they were, unsaved but still available for saving.

Using a Version Control System with Baselining

In the case where more than one developer is working on the code, it is preferable to use a Version Control System (VCS) to keep the various versions of source distinct from each other, rather than relying on manually copying the source.

Requirements

For baselining to be used with a VCS, the following are needed:

- the VCS name of the file; that is, the name used for it in the source file repository,
- the version number of the file,
- a command to check the differences between the current working file and the version identified.

The commands used to carry out the above vary between different control systems, but PRQA supply a helper application.

Version Identification

When the baseline is generated using a VCS, -ident must be used to identify the version of each file.

This may be specified directly as -ident "::version" where version is the identifier, or indirectly by using -ident "myexe %F" to call a program to obtain the identifier. This could be, for example, a wrapper command to query the VCS.

To specify an –ident through the GUI, you have to navigate to *Configuration* > *Options* > *Extensions*. On that screen, highlight *Baseline Diff Parameter* and then *Edit Command Line*.

See also the notes on –ident and on Editing the Command Line, above.

PRQA Helper Application

PRQA supply a helper application called prqavcs.exe and an XML configuration file called prqavcs.xml. These are used to customize the interrogation of the VCS.

It is important that the code in the working directory is up-to-date and that it is synchronized with the original code.

Usage Details: PRQA Helper Application

```
prqavcs [-config <file>] [parameters] <filename>
```

This command specifies the XML configuration file to be used with the given parameters. See the table below for details of the parameters.

The default location for prqavcs.xml is in the same directory as prqavcs.exe.

Parameter	Action
-f,-file <filename>	returns the repository name of the file specified
-s,-status <filename>	returns the current version of the file specified
-d,-diff <version> <filename>	returns the differences between the repository version of the file specified and the currently-checked-out version (see below)
-c,-code <version> <filename>	retrieves the source code of the version of the filename from the repository
-v,-version	writes the program version to STDOUT and exits
-h,-help	writes out the help text and exits

Alternatively, the differences between two versions in the repository can be checked, as opposed to the differences between the specified version and the currently-checked-out version. The command for this is:

```
prqavcs -diff <version1> -diff <version2> <filename>
```

Setting up Baselining with VCS

First, identify the correct XML file for your VCS from the <QAC>\bin\vcs directory. For example, you may want clearcase.xml or cvs.xml.

Copy the file you selected to the prqavcs.xml file in the <QAC>\bin directory. By default, the cvs.xml file is shipped as prqavcs.xml.

If you have a VCS other than CVS or Clearcase, the prqavcs.xml file can be modified by hand, to specify the necessary configuration options.

In QA·C, on the *Configuration* > *Options* > *Extensions Tab*, set the parameters as follows:

Generate Analysis Baseline:

```
%Q %L+ -ident "prqavcs -status" -sf "%O/%J.sup" -sop "%D" -po
baseline::log="%O/%J.log"
```

Baseline Diff Parameter:

```
prqavcs -diff %BT %BF
```

From the VCS repository, check out the version of the code to be baselined (say, version 1.0).

Analyze this code. After that, run *Analyze* > *Generate Analysis Baseline.*

Make the modifications to the source code (or check out, say, version 2.0 into the working directory). Analyze the project. When you open the Message Browser, you will see only the analysis results from the modified code. The analysis results from the baselined code will not be visible at this stage.

If opening the Message Browser in Baseline Admin Mode from the command line, the -sop option must also be set to point at the baselined source.

INDEX

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z