

# **CCT Generator Manual**

22 August 2023

©2023 Perforce Software

# Contents

<b>1</b>	<b>Installation</b>	<b>2</b>
1.1	Windows platform . . . . .	2
1.2	Linux platform . . . . .	2
<b>2</b>	<b>Usage</b>	<b>2</b>
2.1	Command line help . . . . .	2
2.2	Typical use . . . . .	3
2.3	Source language selection . . . . .	4
2.4	Target framework version . . . . .	4
2.5	Overwrite CCT . . . . .	5
2.6	CCT name . . . . .	5
2.7	CCT destination path . . . . .	6
2.8	Compiler hierarchy . . . . .	6
2.9	Verbose console output . . . . .	6
2.10	Current working directory . . . . .	6
2.11	Build command . . . . .	7
2.12	Compiler command . . . . .	7
2.12.1	Obtaining the compiler command . . . . .	8
2.12.2	Cygwin support . . . . .	8
2.12.3	Green Hills notes . . . . .	9
2.12.4	ARMCC notes . . . . .	9
2.12.5	Diab notes . . . . .	9
2.12.6	GNU notes . . . . .	10
2.12.7	QNX notes . . . . .	11
2.12.8	Clang notes . . . . .	11
2.12.9	HighTec TriCore notes . . . . .	11
2.12.10	Microchip notes . . . . .	11
2.12.11	Texas Instruments Code Composer Studio notes . . . . .	12
2.12.12	IAR Embedded Workbench notes . . . . .	12
2.12.13	TASKING notes . . . . .	12
2.12.14	Renesas notes . . . . .	13
2.12.15	System header paths . . . . .	13
2.13	Helix QAC trace log file . . . . .	14
2.14	Klocwork trace output file . . . . .	15
2.15	Blacklisting binaries . . . . .	15
<b>3</b>	<b>Troubleshooting</b>	<b>15</b>
3.1	Generator command line options . . . . .	15
3.2	Environment settings . . . . .	16
3.3	Interpreting console output . . . . .	16
3.4	Reporting problems . . . . .	16

# 1 Installation

Extract the archive in any location.

The generator is a Python script that operates from the command line.

## 1.1 Windows platform

Run `cmd.exe` to obtain a command prompt window in which to run the generator. To run the python script using the Helix QAC python component, use

```
C:\Perforce\Helix-QAC-<version>\components\python-<version>\python
    CCT_Generator.py <options>
```

A 64-bit compiled version of the python script (`CCT_Generator.exe`) and 32-bit version (`CCT_Generator32.exe`) are provided for convenience.

## 1.2 Linux platform

Open an `xterm` window to obtain a command prompt window in which to run the generator. A wrapper script named `CCT_Generator.sh` is provided, which searches a Python interpreter and uses that to run the generator script.

# 2 Usage

## 2.1 Command line help

The generator provides a number of command line options to control its behavior. The following options (short and long form)

```
-h
--help
```

print a brief usage message:

```
usage: CCT_Generator.exe [-h] [-lang {c,cpp}] [-qv QACVERSION] [-o] [-n NAME]
                        [-op OUTPUTPATH] [-vb]
                        [-ch {arm,diab,ghs,gnu,iar,renesas,tasking,ti}]
                        [-qtrace [QACTRACE]] [-kwtrace KLOCWORKTRACE]
                        [-bl [BLACKLIST]] [-c ...] [-b ...]
```

Generate CCT for compiler command.

optional arguments:

```
-h, --help          show this help message and exit
-v, --version       show program's version number and exit
-lang {c,cpp}, --language {c,cpp}
                    specify source language
-qv VERSION, --qacversion QACVERSION
                    specify target framework version
-o, --overwrite     overwrite existing CCT
-n NAME, --name NAME CCT name
-op OUTPUTPATH, --outputpath OUTPUTPATH
                    CCT destination path
-vb, --verbose      show compiler output on error and keep sync projects
-ch {arm,diab,ghs,gnu,iar,renesas,tasking,ti},
--hierarchy {arm,diab,ghs,gnu,iar,renesas,tasking,ti}
                    specify compiler hierarchy
-qtrace [QACTRACE], --qactrace [QACTRACE]
                    qacli sync trace log file
-kwtrace KLOCWORKTRACE, --klocworktrace KLOCWORKTRACE
                    Klocwork kwinject trace output file
-bl [BLACKLIST], --blacklist [BLACKLIST]
                    blacklist binaries for which CCT generation failed
                    BLACKLIST times
-cwd CURRENTWORKINGDIRECTORY, --currentworkingdirectory CURRENTWORKINGDIRECTORY
                    current working directory for build or compiler
                    command
-c ..., --command ...
                    compiler binary and target or language options (must
                    be specified as final option)
-b ..., --build ... project build command (must be specified as final
                    option)
```

## 2.2 Typical use

All generator options are optional and it has ways to derive values according to the compiler or build command.

The generator provides options to generate the CCT from log files that contain compiler commands. See section 2.13 Helix QAC trace log file and section 2.14 Klocwork trace output file. These options can be used when build integration has already been performed and don't require passing of commands. For example:

```
CCT_Generator -qtrace
CCT_Generator -kwtrace kwtrace.log
```

The generator options for specifying commands are `--build` and `--command`. See section 2.11 Build command and section 2.12 Compiler command respectively.

These options must both be specified as final command as all following options will be treated as compiler options. The command must not be enclosed in quotes. If the compiler binary path or any option contains spaces, they must be enclosed in quotes. Some examples are given below.

```
CCT_Generator -b make clean all
```

This will generate a CCT for the first compiler command encountered in the Makefile build. This command will perform a sync of the build command using a special CCT that captures all compiler commands which will slow down the build considerably. Therefore, the most efficient way to use this option will be to only remove one object file for a source file and then specify an incremental build command:

```
CCT_Generator -b make all
```

This will then only compile that single source file so that the overhead of build monitoring will be minimal. Another way to speed up the process is to use blacklisting to not produce CCT's for binaries that failed previously. See section 2.15 Blacklisting binaries.

When using option `--command`, it's important that the build command is taken from the build of the project because there are compiler options that affect CCT settings and the generator will pick these up. If such options are not specified, the CCT will not be correct which may result in incorrect analysis of the code. An example of this is GNU option `-std` for setting the C/C++ language version. If this is not specified, the CCT will be for the default language version of the compiler instead of the version specified in the build.

```
CCT_Generator -c  
"C:\\Program Files (x86)\\Renesas Electronics\\CS+\\CC\\CC-RX\\V3.01.00\\bin\\ccrx.exe"  
-cpu=rx600 -dbl_size=8
```

## 2.3 Source language selection

```
-lang c  
--language cpp
```

In most cases the source language can be derived from either the compiler name or from the source file extension when a source file is specified in the compiler command. If not, this option can be used to specify `c` for C language and `cpp` for C++ language.

## 2.4 Target framework version

The generator will by default generate the CCT in the CCT section of the latest Helix QAC user data store. This option is only needed to specify a specific version in case a different version is required, for example:

```
-qv 2019.1
--qacversion 2.4.0
```

It also determines the tool version used for build synchronization (option `--build`, see section 2.11 Build command ).

In Helix QAC 2021.2, option `-si` was added to the `qac` component for specifying system include paths. Option `-qv` will determine whether `qac` component option `-si` or `-i` will be used for system include paths found by the generator.

Note that on Linux, the old style three digit version numbering must be used. To find this version use

```
$ qaccli -v
Helix QAC 2019.1
Build: 2.5.0-10131 Apr 15 2019 for Windows 64-bit
Copyright (C) 2019 Programming Research Ltd., a Perforce company
```

The version is found on the second line, in this case 2.5.0.

## 2.5 Overwrite CCT

By default the generator will not overwrite existing CCT files or directories and will give a warning if this would happen. By specifying

```
-o
--overwrite
```

the existing files will be overwritten without warning.

## 2.6 CCT name

By default, the generator will produce a name for the CCT by combining a number of its properties to obtain a name that is sufficiently unique in case the CCT is stored in the user data store.

If the CCT is produced in some other place, the name mangling can result in excessively long file system paths. This option can be used to provide some other name for the CCT:

```
-n cct
--name cct
```

## 2.7 CCT destination path

By default, the CCT is generated in the Helix QAC user data store, but it is possible to generate the CCT in any existing location. This can be achieved by specifying the desired destination path with:

```
-op my/cct/dir  
--outputpath my/cct/dir
```

If the leaf directory of the specified path does not exist, it will be created.

## 2.8 Compiler hierarchy

The generator recognizes the usual compiler names used in the compiler command. In case it is unable to determine the compiler hierarchy, make sure that the compiler executable path has been specified correctly and is enclosed in double quotes if it contains spaces. The following option can be used to explicitly specify the hierarchy.

```
-ch {arm,diab,ghs,gnu,iar,renesas,tasking,ti}  
--hierarchy {arm,diab,ghs,gnu,iar,renesas,tasking,ti}
```

Note that this option overrides the name recognition of the generator, so must be used only in case the recognition fails or selects the wrong hierarchy.

## 2.9 Verbose console output

```
-vb  
--verbose
```

With this option the generator also displays compiler command and its standard and error output for any compiler invocation that fails. See section 3.3 Interpreting console output. It also shows logging about blacklisting which is explained in section 2.15 Blacklisting binaries.

## 2.10 Current working directory

```
-cwd /path/to/build/dir  
--currentworkingdirectory ../relative/path/to/build/dir
```

This option sets the working directory for execution of the build and compiler command.

## 2.11 Build command

Note: this feature requires Helix QAC 2.4.0 or newer.

Since there are many compiler options that affect analysis configuration, the generator is best used with information obtained from the build system of the project.

```
-b <build command>  
--build <build command>
```

This option will run the provided build command and monitor its execution. Each compiler command encountered in the build will be processed until a CCT has been generated for the specified language.

Note that build monitoring can cause a substantial slowdown of the build. Therefore, the most efficient way to use this option will be to only remove one object file for a source file and then specify an incremental build command. This will then only compile that single source file so that the overhead of build monitoring will be minimal. Another way to speed up the process is to use blacklisting to not produce CCT's for binaries that failed previously. See section 2.15 Blacklisting binaries.

Also note that when specified, this option must be the last option on the command line since everything following it will be considered part of the compiler command and will not be processed as option of the generator. The command must not be surrounded by quotes.

## 2.12 Compiler command

```
-c <compiler path> {<target option>,<language option>}*  
--command <compiler path> {<target option>,<language option>}*
```

Note that if the compiler executable path contains spaces, the path needs to be enclosed in double quotes.

This option is similar to `--build` but only specifies a single command. The command consists of the compiler binary and any options affecting target or C/C++ language features.

These options should be taken from the build of the project. The generator filters out options that may interfere with its operation so that it is possible to provide a complete compilation command as argument.

The generator also maps compiler options that control certain C(++) language features to the according QAC(++) parser options.

Note that when specified, this option must be the last option on the command line since everything following it will be considered part of the compiler command and will not be processed as option of the generator. The command must not be surrounded by quotes.

### 2.12.1 Obtaining the compiler command

Note that section 2.13 Helix QAC trace log file describes a generator option that automates CCT generation from commands found in a Helix QAC log file. This section remains to explain the approach.

For Helix QAC 2.4.0 or newer, the process monitor method described before is the most convenient method to obtain compiler commands.

For earlier versions, it is still possible to use the Helix QAC process monitor to capture compiler commands. To do this, increase debug level to highest setting

```
qacli admin --debug-level TRACE
```

and synchronize the build of the project as explained in the framework manual. After this the log file which have been created in the app/logs subdirectory of the Helix QAC user data store and will be named

qaframework\_<timestamp>\_<process id>.log. This file will contain an entry for every compiler command detected including all options provided, e.g.

```
10:23:06.129 9560 QAF.debug: PRQA::QAAPI::QAProcessMonitor::  
  WindowsProcessMonitor::Monitor: Command line run:  
C:\cygwin64\bin\gcc.exe -DHAVE_CONFIG_H -I. -I.. -I../h -g  
  -O2 -Wall -funsigned-char -c getopt.c PID(7240) PPID(15280)
```

so the command to pass in would be

```
-c C:\cygwin64\bin\gcc.exe -DHAVE_CONFIG_H -I. -I.. -I../h -g \  
  -O2 -Wall -funsigned-char -c getopt.c
```

### 2.12.2 Cygwin support

The generator supports Cygwin in that the compiler path and include paths may be absolute Cygwin paths. When executed in a Cygwin environment the generator will use `cygpath` to convert such paths to Windows paths.

The python script must not be executed using the Cygwin python interpreter, but the python interpreter that is shipped with Helix QAC can be used to run the python script on Cygwin. Also, the Windows compiled versions of the generator can be used on Cygwin.

If the compiler depends on the Cygwin DLL, it will be easiest to use the generator in a Cygwin terminal so that the DLL will be in the path. In case such terminal is not available, extend the PATH environment variable to include the path where the DLL is located, for instance:

```
set PATH=C:\cygwin64\bin;%PATH%
```

### 2.12.3 Green Hills notes

For Green Hills compilers, which are typically located in a directory named `comp_<version>` and named `cc<target>` for C and `cx<target>`, for C++ or, for Integrity compilers, `ccint<target>` and `cxint<target>` for C respectively C++, it is required to specify the target option.

For Integrity targets this is the `-bsp=<target>` option that also requires an additional `-os_dir=<integrity path>` option. Both must be specified in the options of the compiler command.

For other compilers the `-cpu=<target>` option must be provided.

Other options that are relevant for C language standard selection are `-C99`, `-c99`, `-ANSI`, `-ansi`, `-gcc` and `-gnu99`.

Options that are relevant for C++ language standard selection are `--STD`, `--std`, `--C++11`, `--c++11`, `--C++14`, `--c++14`, `--arm`, `--g++`, `--e` and `--ee`

Options relevant for C++ library support are `--stdl`, `--stdle`, `-eel`, `--eele`, `--el` and `--ele`.

If any of these options are used in the build of the project, they need to be passed to the generator.

### 2.12.4 ARMCC notes

The compiler is named `armcc`. The generator works for compilers developed by ARM and Keil (ARM acquired Keil Software in 2005). Recent ARM compilers are based on Clang, which is supported as part of the GNU support of the generator. See section 2.12.8 Clang notes.

Note that there are many other compilers from other vendors for ARM targets. The generator works for many of those too and will use the name of the compiler to select the appropriate compiler hierarchy. In case of doubt, run the generator without specifying a compiler hierarchy.

The `--cpu=<target>` option must be provided to specify the target architecture.

The generator uses the `-J` compiler option, `ARMINC` environment variables and registry settings in that order to determine the system header file paths.

The options for setting language standard are `--c90`, `--c99`, `--cpp` and `--cpp11`.

### 2.12.5 Diab notes

The generator supports Wind River Diab compilers. They are typically named `dcc` for C and `dpplus` for C++ and use the `-t<tof:env>` option for target configuration where

- `t` is the target processor

- o is the object file format
- f is the floating point support
- env is the execution environment

For command line operation, there is the `wrenv` utility which has option `-p` for specifying the platform name. The platform name can be found in the `.wrproject` file which will have a line like this:

```
<properties platform="Standalone" platform_name="standalone-6.06" ...
```

In this case the platform name is `standalone-6.06` and the generator should be invoked as follows:

```
wrenv -p standalone-6.06 CCT_Generator ...
```

### 2.12.6 GNU notes

This generator is intended for use with any GNU based compiler, including cross compilers. GNU cross compiler use the following naming convention `arch-vendor-(os-)abi` where

- `arch` is for architecture: e.g. `arm`, `mips`, `x86`, `i686`.
- `vendor` is tool chain supplier: e.g. `apple`, `Codesourcery`, `Linux`.
- `os` is for operating system: e.g. `linux`, `none` (bare metal),
- `abi` is for application binary interface convention: e.g. `elf`, `eabi`, `gnueabi`, `gnueabihf`.

Options which are known to affect compiler settings are:

- `-std=<lang>`: sets the source language standard.
- `-fsigned-char` / `-funsigned-char`: sets `char` behavior (only used for C).

GNU based compilers allow specification of system include paths (i.e. paths that are searched for `#include <...>` directives) using option `-isystem`. These options will be picked up by the generator and reflected in the system include paths of the generated CCT. See also section 2.12.15 System header paths for more details about this. Note that other include options such as `-I` and `-iquote` are considered project include paths which are already picked up by Helix QAC project population methods and are for that reason ignored by the generator.

### 2.12.7 QNX notes

Since QNX compilers are based on GNU, the generator supports them using its GNU support. See section 2.12.6 GNU notes. The compilers are usually called `qcc` for C and `q++` for C++ and use the `-V` option for target selection.

Note that these compilers require setting of the environment variables `QNX_HOST` and `QNX_TARGET`. The generator will fail if either is not set correctly because the compiler will then fail to produce settings.

### 2.12.8 Clang notes

Since Clang compilers use a GCC-based preprocessor, the generator supports them using its GNU support. See section 2.12.6 GNU notes.

On Windows, Clang requires Microsoft Visual Studio header files. The generator supports this with a variant that is based on a Visual Studio CCT.

Clang language options are the same as for GNU. Use option `--target=<value>` to specify the target architecture which uses the same triplet naming convention as GNU. See section 2.12.6 GNU notes for treatment of system include paths.

### 2.12.9 HighTec TriCore notes

These compilers are also based on GNU so are supported by the generator using its GNU support. See section 2.12.6 GNU notes.

Note that the environment needs to be setup so that the compiler license can be accessed. This typically entails setting environment variable `RLM_LICENSE` to point to the license server and adding the location of the license manager to the search path.

### 2.12.10 Microchip notes

The MPLAB XC32 C/C++ compilers (`xc32-gcc` and `xc32-g++`) are based on GNU and supported by the generator using its GNU support. See section 2.12.6 GNU notes.

They uses the option `-mprocessor=<target>` to specify the target architecture which also affects setting of the system include path and inclusion of the appropriate target settings when using the system headers.

The MPLAB C30 (`pic30-gcc`) and XC16 (`xc16-gcc`) compilers are based on GNU and supported by the generator. Both use the option `-mcpu=<target>` to specify the target architecture which also affects setting of the system include path and inclusion of the appropriate target settings when using the system headers.

The latest MPLAB XC8 compiler (`xc8-cc`) includes Clang, GNU and proprietary front-ends for different targets and language versions. It is supported by the generator as a GNU variant. It uses the option `-mcpu=<target>` to specify the target architecture which will also affect setting of the system include path and inclusion of the appropriate target settings when using the system headers. Its precursor (`xc8`) which is based on the HI-TECH C Compiler is also supported and uses the option `--CHIP=<target>` to select target device.

### 2.12.11 Texas Instruments Code Composer Studio notes

These compilers typically have a name that starts or ends with `c1`, for example `armc1` and `c1430`. Also the system include path is usually specified explicitly as a command line option, e.g.

```
-I=C:\ti\ccsv5\tools\compiler\arm_5.1.1\include
```

The generator will pick this up and use it in the CCT generated. Note that no project include paths should be specified since that will result in messages from those paths being suppressed.

The compiler already is for a specific target architecture so no further options are required for target selection.

The same compiler supports both C and C++ based on extension of the source file. The `--cpp` option can be used to force C++ compilation.

### 2.12.12 IAR Embedded Workbench notes

These compilers typically have a name that starts with `icc` followed by the target identifier, for example `iccarms`.

The same compiler supports both C and C++. It has options such as `--c++` to compile C++, or `--ec++` for Embedded C++ or `--eec++` for Extended Embedded C++.

### 2.12.13 TASKING notes

The supported C compilers are `ctc`, `carm`, `cmcs` and `c166` and C++ compilers `cptc`, `cparm`, `cpmcs` and `cp166`. The options used for target selection are `--cpu=<architecture>` and `--core=<core>`.

Note that the compiler installations also include control programs (`cctc`, `ccarm`, `ccmcs` and `cc166` respectively), which themselves are not compilers but rather utilities that invoke the corresponding compiler, assembler and linker in the correct order. As such, they are used for both C and C++. The generator is not supported for these control programs but most control program options are also compiler options.

However, the target selection control program option `--cpu=<cpu>` is not a compiler option but instead converted to macro symbol settings when calling the compiler. For convenience, the converter treats this option as a compiler option and converts it to compiler options in the same way as the control program, so that `--cpu=<cpu>` can be used in the compiler command passed to the generator.

#### 2.12.14 Renesas notes

The generator supports the CC compilers (`ccrx`, `ccrh` and `ccr1`) and CA/CX compilers (`cx`, `ca850`, `cc78k0` and `cc78k0r`) shipped with the CubeSuite+ IDE.

The compilers use different options to specify target CPU or chip. In addition, a path may need to be specified for the target device file location. The binary device file contains definitions of target features that may be used in the program. The converter translates this device file to a C header file with the according definitions. This support for device files is only available in the Windows compiled versions of the generator. Below table shows the options for target selection and device path specification.

compiler	target option	device path option
<code>ccrx</code>	<code>-cpu=&lt;cpu&gt;</code> or <code>-isa=&lt;arch&gt;</code>	-
<code>ccrh</code>	<code>-Xcpu=&lt;core&gt;</code>	-
<code>ccr1</code>	<code>-cpu=&lt;core&gt;</code>	<code>-dev=&lt;path&gt;</code>
<code>cx</code>	<code>-C&lt;device&gt;</code>	<code>-Xdev_path=&lt;path&gt;</code>
<code>ca850</code>	<code>-cpu &lt;device&gt;</code>	<code>-devpath=&lt;path&gt;</code>
<code>cc78k0</code>	<code>-c&lt;device&gt;</code>	<code>-y&lt;path&gt;</code>
<code>cc78k0r</code>	<code>-c&lt;device&gt;</code>	<code>-y&lt;path&gt;</code>

When using the Build>Build Option List action on a project in the CubeSuite+ IDE, the Build Tool tab will show the complete compiler invocation including all options. Since there are also many other options that affect the CCT, it is recommended to use this command to produce the CCT.

#### 2.12.15 System header paths

The converter uses the same approach for dealing with system header paths for all compiler hierarchies, which is different from traditional CCT's where this step always occurs during the generation of the CIP file.

Instead, the generator determines the paths at the time of generation of the CCT and produces a file named `syshdr.lst` in the Stub directory of the CCT. After generating a CCT, verify that the contents of this file are correct.

The generated CCT will not invoke the compiler to find out the system include paths but instead directly use the paths found in `syshdr.lst`. This means that the compiler is no longer required when using the generated CCT.

Since there may be need for using the generated CCT with compilers installed in different locations, it is possible to specify root paths in `syshdr.lst` that will be used as root for following relative paths, e.g.:

```
:ROOT:  
C:\workspace\project\tools\gnuc\  
4.8.2\lib\gcc\m68k-elf-bes2\4.8.2\include-fixed  
4.8.2\m68k-elf-bes2\include  
4.8.2\lib\gcc\m68k-elf-bes2\4.8.2\include  
4.8.2\m68k-elf-bes2\include\c++\4.8.2  
4.8.2\m68k-elf-bes2\include\c++\4.8.2\m68k-elf-bes2
```

When there is a need to use this for a compiler installed in a different location, only the second line needs to be adjusted.

However, it is also possible to use the generator in a deployment script where it would be invoked just before the framework project creation command and the framework project creation command uses the generated CCT.

## 2.13 Helix QAC trace log file

This option produces CCT's for compiler commands found in a trace log file produced by Helix QAC as outlined in section 2.12.1 Obtaining the compiler command.

```
-qtrace  
-qtrace Helix-QAC_20200407T101508_16616.log  
--qactrace qaframework_20200407T103632_3048.log
```

To obtain the log file, increase logging level to at least DEBUG using one of:

```
qacli log --set-level TRACE  
qacli admin --debug-level TRACE
```

In order to synchronize the build, a Helix QAC project will need to be created for which a CCT needs to be specified. Choose the closest CCT that is shipped with Helix QAC or the SyncCCT that is shipped with the generator.

Blacklisting can be used to improve performance. See section 2.15 Blacklisting binaries.

If no log file is specified, the generator will process all log files found in the user data store of the specified Helix QAC version starting with the most recent log, until a CCT is generated. This will only work when logs are not stored in the project. To disable project logging to be local to the project use:

```
qacli log --project-logging disable
```

When the CCT has been generated it can be used in the Helix QAC project instead of the initial temporary choice and logging can be set back to the minimum using one of:

```
qacli log --set-level NONE
qacli admin --debug-level NONE
```

## 2.14 Klocwork trace output file

This option produces CCT's for compiler commands found in a Klocwork trace output file produced by `kwinject --trace-out`.

```
-kwtrace kwtrace.log
--klocworktrace kwtrace.log
```

Blacklisting can be used to improve performance. See section 2.15 Blacklisting binaries.

## 2.15 Blacklisting binaries

```
-bl [level]
--blacklist [level]
```

If the generator failed to produce a CCT for a binary more times than the specified level, the binary will be blacklisted and no further attempts will be made to produce a CCT for it. The default level is one. When the option is not specified, no blacklisting is performed.

# 3 Troubleshooting

## 3.1 Generator command line options

Section 2 Usage explains all command line options of the generator. There are no required options; the generator is able to derive all information from the compiler command. In particular option `--hierarchy` is not required and the generator includes logic for deriving the hierarchy from the compiler binary name for all supported compiler hierarchies. See section 2.8 Compiler hierarchy.

Note that compiler hierarchy `arm` is intended only for the Keil `armcc` compiler (ARM acquired Keil Software in 2005), and not for any other compilers for ARM targets from other compiler vendors. See section 2.12.4 ARMCC notes. The latest ARM compilers are named `armclang` and are based on Clang so are supported by the generator using its GNU support. See section 2.12.8 Clang notes.

Other options that are not required are `--version` and `--outputpath`. See section 2.4 Target framework version and section 2.7 CCT destination path for their usage.

## 3.2 Environment settings

Some compilers depend on certain environment variable settings. All known variables are mentioned in the applicable subsections of Section 2.12 Compiler command. Please make sure to read all sections that are relevant for the compiler that you are using the generator on.

## 3.3 Interpreting console output

The generator produces console output that can be used to investigate issues. It provides basic output for incorrect use of generator options.

Most console output is produced for compiler invocations. The generator tries to keep console output to a minimum. By default it will only report non-zero compiler exit codes in case it didn't succeed in obtaining the expected information from calling the compiler. The meaning of these exit codes is completely dependent on the compiler and can be looked up in the compiler documentation. When generator option `--verbose` is specified, in these cases also the complete compiler invocation and both its standard and error output are displayed, which may help in adjusting the environment or compiler command to solve the error.

Note that for several compiler hierarchies the generator will try multiple option combinations to obtain the required information and it is normal that some of these combinations will fail. This does not affect the correctness of the CCT that is generated for the correct option combination.

When the `--build` option is used, the generator will attempt to produce a CCT for each command seen in the build. See section 2.11 Build command. Since most commands typically are not compiler invocations this will cause a lot of spurious console output that can be ignored.

## 3.4 Reporting problems

Since the generator supports a wide range of compiler hierarchies that all require specific attention, there are many different ways in which it can fail. Section 2.12 Compiler command provides a lot of compiler specific information that is relevant for the use of the generator. Please consult the applicable sections in case of problems.

Any remaining questions or problems should be reported at <https://www.perforce.com/support/request-support>.

Please always submit the complete command line of the invocation of the generator and the complete console output of that invocation. In order to ease the investigation and reproduction of issues, please submit these in plain text instead of a screenshot.