# XCC User Manual

**V1.00**

Pizer.Fan

## Contents

## Preface

The XCC system is used to further improve the quality of software code and strengthen the quality construction of process management. In particular, it has significant help for quality management in the software development and verification stages.

This article will discuss how to quickly use the XCC system in Chapter 1. In the following chapters, each tool of the XCC system, such as XCC, XCovPost, and XCovHTML, will be introduced in detail. At the end of this article, there are some related appendices, which include explanations of some terms. If you encounter some abbreviations that you don't understand, you can get help from the appendix section of terms.

Here, I would like to reiterate the functions of the XCC system, including:

(1) Assisted static scanning:
- ✓ Provide what mainstream static scanning tools do not have.
- ✓ Measure code maintainability.
- ✓ Code style specification.
- ✓ ASSERT line management.
- ✓ Function call stack overhead.
- ✓ Function dependency tree.

(2) TRACE scanning:
- ✓ Improve code density.
- ✓ Provide LOG performance.
- ✓ Provide forward-compatible ID database management, and realize compilation and linking without modifying the source code.

(3) ASSERT line scanning:
- ✓ Scanned using precompilation.
- ✓ Support white-list database, with alert function (alarm) after filtering white-list.

(4) Coverage measurement:
- ✓ Mainstream host operating environments: Windows, Linux (Ubuntu, CentOS, Debian, etc.).
- ✓ Various compilers on mainstream chip architectures are supported, such as ARMCC, ARMCLANG, IAREWARM, ARM-NONE-EABI-GCC, ARC, RISCV, etc.
- ✓ Provide branch coverage, function coverage, and detailed HTML reports.

(5) Ease of use of the XCC system:
- ✓ For example, it is very easy to use in conjunction with the building subsystem (specifically, the LOS platform's building subsystem).

Suggestions or opinions are welcome!

Finally, I wish you a reassuring and comfortable experience using the XCC system.

Author: Pizer.Fan

August 15, 2022

# 1 Quick Start

## 1.1 Starting from building

### 1.1.1 Based on the LOS building subsystem

If your project is already compiled and linked based on the LOS building subsystem, all you need to do to seamlessly support all the functions of XCC is to involve a specific version related to the LOS building subsystem.

#### *1.1.1.1 Modify project configuration file*

For example, if your project configuration file is "project_mysata.mak", and you want to perform assisted code scanning while building, just add the following line:

```
…
XCCCODESCAN := on
…
```

During the following build process, you will see the report of the assisted code scanning at the same time (either through confirmation or alarm):

```
Remove output_clang_cov/obj/utest_xcc_cscan_rule/
output_clang_cov/dep/utest_xcc_cscan_rule/ ...
Compiling and update dependency ../code/iLib/iFileOps.c
Compiling and update dependency ../code/iLib/iString.c
Compiling and update dependency ../code/iLib/iStream.c
Compiling and update dependency ../code/iLib/iRbtree.c
Compiling and update dependency ../code/iLib/iList.c
Compiling and update dependency ../code/iLib/iAllocator.c
XccCScan ../code/iLib/iFileOps.c ... Okay
XccCScan ../code/iLib/iList.c ... 0 Errors, 29 Warnings
  [XccCScan] Warning 6: The iList_Create function did NOT match the naming rule
@../code/iLib/iList.c:198
  [XccCScan] Warning 6: The iList_Destroy function did NOT match the naming rule
@../code/iLib/iList.c:210
  [XccCScan] Warning 6: The iList_GetName function did NOT match the naming rule
@../code/iLib/iList.c:216
  [XccCScan] Warning 6: The iList_GetCount function did NOT match the naming rule
@../code/iLib/iList.c:222
…
```

#### *1.1.1.2 Check the results of static scanning*

For example, if your project configuration file is "project_mysata.mak", check the "output_mysata" directory:

```
\
|- cov
|- dep
|- lint
|- obj
|- out
|- tsc
|- xcov
|- build.log
|- macroes.option
|- xcc-cscan.log
|- xcc-protodb.dat
```

In the "output_mysata" directory, you can see the file "xcc-cscan.log", whose content is similar to:

```
XccCScan ../code/xcc/tsc_strdb/tsc_strdb_lex.c : 0 Errors, 3 Warnings
  [XccCScan] Warning 1: Exceed cyclomatic complexity limit 70 > 15 in the tsc_strdb_yylex function
@../code/xcc/tsc_strdb/tsc_strdb_lex.c:9615
  [XccCScan] Warning 7: The tsc_strdb_yyleng global variable did NOT match the naming rule @../code/xcc/tsc_strdb/tsc_strdb_lex.c:215
  [XccCScan] Warning 7: The tsc_strdb_yyout global variable did NOT match the naming rule @../code/xcc/tsc_strdb/tsc_strdb_lex.c:264
XccCScan ../code/test_cov/test_cov.c : 0 Errors, 1 Warnings
  [XccCScan] Warning 1: Exceed cyclomatic complexity limit 55 > 15 in the func function @../code/test_cov/test_cov.c:172
XccCScan ../code/utest_clang_case1.c : 0 Errors, 7 Warnings
  [XccCScan] Warning 1: Exceed cyclomatic complexity limit 47 > 15 in the utest_base function @../code/utest_clang_case1.c:294
```

### 1.1.2 Based on Makefile building

#### *1.1.2.1 Modify the compilation process*

For example, to modify the compilation process, typically find the rule "% .0->% .c":

```
$(PROJECT_OUTDIR)/obj/$(MOD)/%.o : %.d
$(PROJECT_OUTDIR)/obj/$(MOD)/%.o : %.c
	$(ECHO) Compiling $<
	$(CC_EXEC) …
```

Then replace the compiler command (refer to the modification marked with the following colors):

- ✓ First configure the path of the XCC executable: XCC_EXEC := xcc.out (if in a Windows environment, it is xcc.exe, and the path should be configured carefully)
- ✓ Then replace the $(CC_EXEC) compiler command with: CC_EXEC := $(XCC_EXEC) $(XCC_CMD_OPTS) $(CC_EXEC)

```
# XCC
XCC_EXEC := xcc.out
ifeq ($(XCCCODESCAN),on)
XCC_CMD_OPTS += --cmd=codescan  --resultfile=./xcc-cscan.log --protodbfile=./xcc-protodb.dat
endif
ifeq ($(XCCCOV),on)
XCC_CMD_OPTS += --cmd=cov
endif
ifeq ($(XCCDRYSCAN),on)
XCC_CMD_OPTS += --cmd=dryscan
endif
ifneq ($(wildcard $(XCC_EXEC)),)
ifneq ($(XCC_CMD_OPTS),)
CC_EXEC := $(XCC_EXEC) $(XCC_CMD_OPTS) $(CC_EXEC)
endif
endif

$(PROJECT_OUTDIR)/obj/$(MOD)/%.o : %.d
$(PROJECT_OUTDIR)/obj/$(MOD)/%.o : %.c
	$(ECHO) Compiling $<
	$(CC_EXEC) ...
```

#### 1.1.2.2 *Linking process modification (only for gcc on x86 architecture)*

For example, to modify the linking process:

```
$(TARGET_OUT) : ...
	$(ECHO) Link $@ ...
	$(LINK_EXEC) ...
	$(ECHO) Done
```

Replace the linking command (refer to the modifications marked in color below):

```
$(TARGET_OUT) : ...
	$(ECHO) Link $@ ...
ifneq ($(wildcard $(XCC_EXEC)),)
	$(XCC_EXEC)  $(LINK_EXEC) ...
else
	$(LINK_EXEC) ...
endif
	$(ECHO) Done
```

### 1.1.3 Building based on IDE

Not yet supported.

## 1.2 Target board coverage data output

### 1.2.1 Output adaptation development

XCC system provides basic library files such as xcov_lib.c and xcov_lib_arm.c. To output coverage data on the target board, adaptation development is required:

- Redirect the data output to UART or save it to storage media such as RAM or Flash.
- Implement the trigger dump process by calling xcov_dump.

### 1.2.2 Critical adaptation

For single-core critical adaptation, disabling interrupts is usually sufficient. For systems with RTOS, avoiding RTOS-level MUTEX is recommended for performance reasons. For multi-core critical adaptation, disabling interrupts + spin lock are used. If an RTOS (SMP) is available, avoiding its MUTEX mechanism is also recommended.

### 1.2.3 Other suggestions

To shield the impact of coverage functionality on startup and performance modules, add the following line to the source file:

```
#pragma XCov off
```

If using makefile, for example, based on LOS to build a subsystem, configure XCCCOV = off in the module makefile to abandon coverage function for all files in the module, such as:

```
TARGET := test_cov$(OUT_DEF_SUFFIX)

TARGET_OUT_DEPS_LIBS :=
TARGET_OUT_DEPS := $(TARGET_OUT_DEPS_LIBS)

CODE_ROOT_DIR := ../code
XCCCOV := off
```

## 1.3 Cost of coverage functionality

The cost of the coverage functionality is:

- ROM cost: to be determined
- RAM cost: to be determined

# 2 XCC Interfaces

## 2.1 Help for command

```
>xcc -h
Usage: xcc [options] compiler [compiling-options]
 General options:
  -h/--help
   Prints this message
  -v/--version
   Prints the version info of the tool.
  --silent
   Disable outputing verbose info.
  --list-env
   List supported compilers.
  --color on|off
   Enable/disable outputing message with multi-color.

 Command options:
  --cmd <command>
   Execute XCC command:
   * dryscan      - just test XCC command.
    codescan     - scan codes to check if there're any warnings or not.
    assertscandb - scan assert lines database of C syntax, into precompiled database.
    assertscan   - scan assert lines to check if they're verified or not.
    tracescan    - convert constant strings to identifiers in the logging invocations.
    cov          - insert veneer codes for coverage.

 Command 'assertscandb' & 'assertscan' options:
  --assertfunc <assert-func-name>
   Specify the fucntion name of assert.
   For example :
    --assertfunc=my_assert1 --assertfunc=my_assert2
    or :
    --assertfunc=my_assert1:my_assert2
  --assertdbfile <precompiled-db-path>
   Specify the path of precompiled database.
  --assertresultfile <result-file>
   Specify the result file.

 Command 'codescan' options:
  --rulefile <rule-file>
   Specify the rule file.

   Rule options:
    --disable <rule-check-point>
    --enable <rule-check-point>
  Rule check point:
      cyclomatic_complexity_limit
      code_lines_limit
      for_max_depth_limit
      if_max_depth_limit
      include_files_limit
      loop_include_files
      func_naming_rule
      glb_var_naming_rule

    --cyclomatic_complexity_limit=<value>
    --code_lines_limit=<value>
    --for_max_depth_limit=<value>
    --if_max_depth_limit=<value>
    --include_files_limit=<value>

  --resultfile <result-file>
   Specify the result file.
  --protodbfile <protodb-file>
   Specify the prototype database file.

 Command 'tracescan' options:
  --tscdbfile <tsc-db-file>
   Specify the database file for trace scan.
  --create-mod <module list>
   Specify the module list to be created to the database.
   For example: "mod1 mod2 ..."
    The character of module name shall be one of [A-Z/a-z_0-9]
  --ls-log-header-file <file path>
   Specify the header file of LS_LOG/LS_LOG_ID macroes.

 Command 'cov' options:
  --zidata-section <section-name>
   Specify the section for the new XCov .bss data.
  --rodata-section <section-name>
   Specify the section for the new XCov .rodata data.

This XCC tool is using to scan codes for searching the possible defects, convert constant strings to
identifiers, and insert veneer codes for coverage.

################################################################################
XCC Release V1.00 Version, 2 July 2022.
Copyright (C) 2022 (Any question, please contact me: lzfzmpizer@sina.com).
################################################################################
```

## 2.2 Static Scanning

### 2.2.1 The format of the rule file

The format of the rule file content is:

```
 --rulefile <rule-file>
  Specify the rule file.

  Rule options:
    --disable <rule-check-point>
    --enable <rule-check-point>
      Rule check point:
        cyclomatic_complexity_limit
        code_lines_limit
        for_max_depth_limit
        if_max_depth_limit
        include_files_limit
        loop_include_files
        func_naming_rule
        glb_var_naming_rule
    --cyclomatic_complexity_limit=<value>
    --code_lines_limit=<value>
    --for_max_depth_limit=<value>
    --if_max_depth_limit=<value>
    --include_files_limit=<value>
```

For example, to input rule file ./xcc-cscan-rule.txt:

```
--disable cyclomatic_complexity_limit
--disable code_lines_limit
--disable for_max_depth_limit
--disable if_max_depth_limit
--disable include_files_limit
--disable loop_include_files
--disable func_naming_rule
--disable glb_var_naming_rule

--enable cyclomatic_complexity_limit
--enable code_lines_limit
--enable for_max_depth_limit
--enable if_max_depth_limit
--enable include_files_limit
--enable loop_include_files
--enable func_naming_rule
--enable glb_var_naming_rule

--cyclomatic_complexity_limit=20
--code_lines_limit=2000
--for_max_depth_limit=4
--if_max_depth_limit=4
--include_files_limit=256
```

### 2.2.2 The option specifying the rule file

If building a subsystem based on LOS, only need to configure in the project configuration file:

```
XCCCODESCAN_RULEFILE := ./xcc-cscan-rule.txt
```

If built based on Makefile, need to add the following command options in the process of replacing compilation commands:

```
--rulefile=./xcc-cscan-rule.txt
```

### 2.2.3 Disabling rules in code

If you want to disable certain static scanning rules in a code file, you can add comments such as:

```
#ifdef XccCScan
#pragma XccCScanSwitch off(maintainability_limit,code_lines_limit)
#endif
…
#ifdef XccCScan
#pragma XccCScanSwitch on
#endif
```

If you want to suppress all rules of static scanning in a certain code file, for example:

```
#ifdef XccCScan
#pragma XccCScanSwitch off(*)
#endif
…
#ifdef XccCScan
#pragma XccCScanSwitch on
#endif
```

### 2.2.4 Alerts

#### *2.2.4.1 ERROR alert*

ERROR 0 - Different function prototype

#### *2.2.4.2 WARNING alert*

WARNING 128 - Exceed cyclomatic complexity limit in the function

WARNING 129 - Exceed code line limit in the function

WARNING 130 - The maintainability was bad in the function

WARNING 131 - The foreach maximum depth limit in the function

WARNING 132 - The branch maximum depth limit in the function

WARNING 133 - The function did NOT match the naming rule

WARNING 134 - The global variable did NOT match the naming rule

WARNING 135 - There's loop including codes

WARNING 136 - There's too many including files

#### *2.2.4.3 INFO alert*

INFO 4096 - The %s global variable is recommended to rename with '_g' NOT 'g_'

## 2.3 ASSERT LINE Scanning

ASSERT LINE scanning is based on the ASSERT LINE whitelist database, and uses source files waiting for compilation with the compilation method for scanning.

Step 1: First build the ASSERT LINE whitelist database, for example, create assertlinedb.c, and create an assertdb() function in it to add each ASSERT LINE:

```
#include <stdio.h>

void assertdb()
{
  MY_ASSERT(x > 0);
  MY_ASSERT2(x > 0);
}
```

Step 2： Pre-parse the ASSERT LINE whitelist database to output the parsed whitelist database, such as parsing assertscandb.c to assertscandb.c.i.

- xcc --cmd=assertscandb, used to preprocess the C syntax ASSERT LINE whitelist database.
- --assertdbfile=assertscandb.c.i, used to indicate the preprocessed whitelist database.
- --assertfunc used to indicate the assert function, such as my_assert, my_assert2.
- armcc -c assertscandb.c indicates that the target assertscandb.c is compiled using the armcc compiler, along with other compilation options.

```
> xcc --cmd=assertscandb --assertfunc=my_assert --assertfunc=my_assert2
--assertdbfile=assertscandb.c.i armcc -c assertscandb.c
```

Step 3: Create a scan result file, for example, result.txt.

Step 4: Compile all target source files.

- We need to replace <CC> in the original '<CC> -c xxxx.c' compile command with 'xcc --cmd=assertscan --assertdbfile=assertscandb.c.i --assertfunc=my_assert <CC>'.
- xcc --cmd=assertscan, used for assert line scanning.
- --assertdbfile=assertscandb.c.i, used to indicate the preprocessed whitelist database.
- --assertfunc used to indicate the assert function, such as my_assert, my_assert2.
- --assertresultfile=result.txt used to specify the file for storing the assert scanning results.

```
> xcc --cmd=assertscan --assertfunc=my_assert --assertfunc=my_assert2 --
assertdbfile=assertscandb.c.i --assertresultfile=result.txt armcc -c
xxxx.c
```

If an ASSERT line appears in the source code but cannot be found in the whitelist, an alert will be raised as follows:

```
XccAssertScan xxxx.c ... 1 Warnings, 2 Passes
  [XccAssertScan] Pass : "my_assert((x>0))" @xxxx.c(25)
  [XccAssertScan] Pass : "my_assert2((x>0),"x > 0")" @xxxx.c(26)
  [XccAssertScan] Warning : "my_assert2((x!=0),"x != 0")" @xxxx.c(27)
```

If the --assertresultfile parameter specifies the result file to be saved, the above alert will be saved in that file.

## 2.4 LOG ID Scanning

Some important command options for LOG ID:

```
Command 'tracescan' options:
  --tscdbfile <tsc-db-file>
    Specify the database file for trace scan.
  --create-mod <module list>
    Specify the module list to be created to the database.
    For example: "mod1 mod2 ..."
      The character of module name shall be one of [A-Z/a-z_0-9]
  --ls-log-header-file <file path>
    Specify the header file of LS_LOG/LS_LOG_ID macroes.
```

### 2.4.1 Specifying the database file path

It is recommended to specify the database file for LOG ID, so that newly added IDs can be appended in a forward-compatible way. For example:

```
--tscdbfile=./tsc_db/tsc_db.cxx
```

If the database file is not specified, the XCC tool will default to ./tsc_db/tsc_db.cxx as the current database file.

### 2.4.2 Specifying the redirection file for LOG API

Considering that LS_LOG/LS_LOG_ID may have different redirection implementations in different projects, developers are required to specify the LOG API redirection file. For example:

```
--ls-log-header-file=tools/runtime/xcc_ls_log.h
```

### 2.4.3 Creating LOG module

LS LOG supports modularity. Considering that LOG modules cannot be created automatically at will, they need to be manually created. The XCC tool is compatible with some TSC functions, such as creating LOG modules (which will be updated in the database). For example:

```
> tools\xcc.exe --cmd=tracescan --create-mod="ABC CDE"
XccTraceScan : Warning: The ABC module was already existed!
XccTraceScan : Warning: The CDE module was already existed!
```

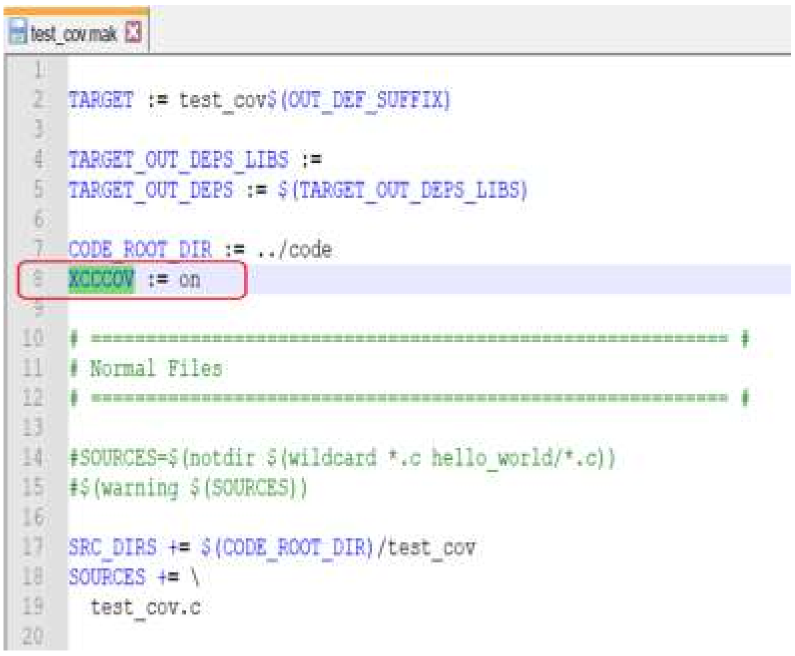
It is worth mentioning that:

- It is recommended to use uppercase letters, underscores, and numbers for the module name to be created.
- If the module has already been created, XCC will report relevant warnings.

## 2.5 Code coverage instrumentation

### 2.5.1 Project or module configuration

To instrument code coverage, simply configure XCCCOV=on in the module makefile (based on LOS building subsystem), for example:



```
TARGET := test_cov$(OUT_DEF_SUFFIX)

TARGET_OUT_DEPS_LIBS :=
TARGET_OUT_DEPS := $(TARGET_OUT_DEPS_LIBS)

CODE_ROOT_DIR := ../code
XCCCOV := on

# ============================================================ #
# Normal Files
# ============================================================ #

#SOURCES=$(notdir $(wildcard *.c hello_world/*.c))
#$(warning $(SOURCES))

SRC_DIRS += $(CODE_ROOT_DIR)/test_cov
SOURCES += \
  test_cov.c
```

```
>make p=clang_cov m="test_cov2 test_cov" build job=8
Update makefile dependency for the test_cov2 module ...
Remove output_clang_cov/obj/test_cov2/ output_clang_cov/dep/test_cov2/ ...
Update makefile dependency for the test_cov module ...
Remove output_clang_cov/obj/test_cov/ output_clang_cov/dep/test_cov/ ...
Compiling and update dependency ../code/test_cov/test_cov_filem.c
Compiling and update dependency ../code/test_cov/test_cov_file2.c
Compiling and update dependency ../code/test_cov/test_cov_file1.c
XccCov ../code/test_cov/test_cov_file2.c ... Okay [1 checkpoints]
XccCov ../code/test_cov/test_cov_filem.c ... Okay [1 checkpoints]
XccCov ../code/test_cov/test_cov_file1.c ... Okay [3 checkpoints]
Link output_clang_cov/out/test_cov2.exe ...
Done
Compiling and update dependency ../code/test_cov/test_cov.c
XccCov ../code/test_cov/test_cov.c ... Okay [122 checkpoints]
Link output_clang_cov/out/test_cov.exe ...
Done
Generate image ...
Making binaries ...
Done
```

### 2.5.2 Code masking

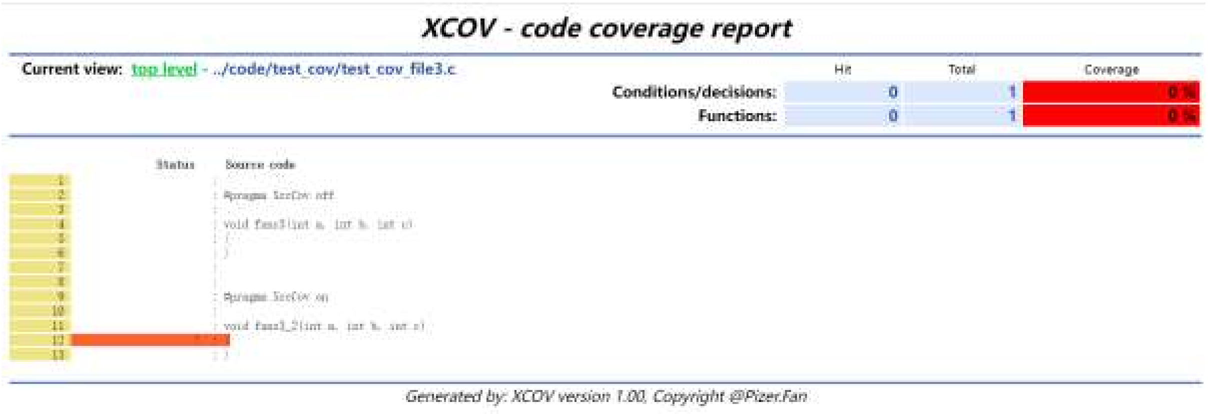
If you want to mask coverage instrumentation in a particular code file, for example:

```
#ifdef XccCov
#pragma XccCovSwitch off
#endif

void func3(int a, int b, int c)
{
}


#ifdef XccCov
#pragma XccCovSwitch on
#endif

void func3_2(int a, int b, int c)
{
}
```



**XCOV - code coverage report**

Current view: top level - ../code/test_cov/test_cov_file3.c

| | Hit | Total | Coverage |
|---|---|---|---|
| Conditions/decisions: | 0 | 1 | 0 % |
| Functions: | 0 | 1 | 0 % |

```
Status    Source code

          #pragma XccCov off

          void func3(int a, int b, int c)
          {
          }


          #pragma XccCov on

          void func3_2(int a, int b, int c)

          }
```

Generated by: XCOV version 1.00, Copyright @Pizer.Fan

*Figure 1 Code coverage instrumentation report after masking*

If you expect to only shield a certain section of code without affecting other code.:

```
#ifdef XccCov
#pragma XccCovSwitch push(off)
#endif

void func3(int a, int b, int c)
{
}


#ifdef XccCov
#pragma XccCovSwitch pop
#endif

void func3_2(int a, int b, int c)
{
}
```

Alternatively, if you only expect to enable coverage instrumentation for a certain section of code without affecting other code:

```
#ifdef XccCov
#pragma XccCovSwitch push(on)
#endif

void func3(int a, int b, int c)
{
}


#ifdef XccCov
#pragma XccCovSwitch pop
#endif

void func3_2(int a, int b, int c)
{
}
```

### 2.5.3 Specifying section for instrumented data

If you want to put the instrumented data in a specific section, refer to the relevant parameters of cov:

```
 Command 'cov' options:
   --zidata-section <section-name>
     Specify the section for the new XCov .bss data.
   --rodata-section <section-name>
     Specify the section for the new XCov .rodata data.
```

For example, in the project configuration makefile:

```
# project_test.mak
XCC_CMD_OPTS := --zidata-section=”MYSEC” --rodata-section=”MYSEC”
XCCCOV := on
```

Or enter relevant parameters on the command line:

```
 xcc --cmd=cov --zidata-section=”MYSEC” --rodata-section=”MYSEC” …
```

If you expect to put instrumentation data for a specific file into a specified section, you can do the following:

```
#ifdef XccCov
#pragma XccCovSection rodata = "cov.rodata", zidata="cov.zidata"
#endif
```

For example, if we add the above line to a certain file, after linking is complete, we can view its section information:

```
>nm test.exe | grep cov\.
0000000140017000 r cov.rodata
0000000140014000 d cov.zidata
```

### 2.5.4 Bank/Overlay Linking

In embedded systems, Bank or Overlay Link refers to the partitioning of static memory into multiple different parts (called banks, overlays, etc.) and allowing programs to switch between these parts at runtime. This method is often used to solve memory capacity issues while also improving program efficiency.

- In Bank Linking, programmers manually divide static memory into multiple banks and store different parts of the program in different banks. Then, the linker allocates each module to the corresponding bank according to the programmer's specified bank allocation plan. The program only needs to switch banks when needed.
- In Overlay Linking, programmers divide the code into multiple functional modules and place each module in a different area of memory. Then, the linker needs to be able to identify each module and determine which modules can share the same memory area. For example, if two modules' memory spaces have no overlapping parts, they can be allocated to the same memory area. When the program is executed, only the module currently needed will be loaded into memory. If you need to switch to another module, the current module will be unloaded from memory, and the required module will be loaded.

Both Bank Linking and Overlay Linking require support from the linker and compiler to correctly allocate code to different banks or overlays and maintain jump relationships between programs. Depending on the specific application scenario, programmers can choose to use one of these technologies or combine them to achieve the best effect in memory management.

- Under Bank/Overlay Linking, the XCOV data of Bank needs to be placed in resident memory.
- During the program start-up process, the XCOV segment of Bank needs to be copied to the corresponding location where the execution attempt is made.

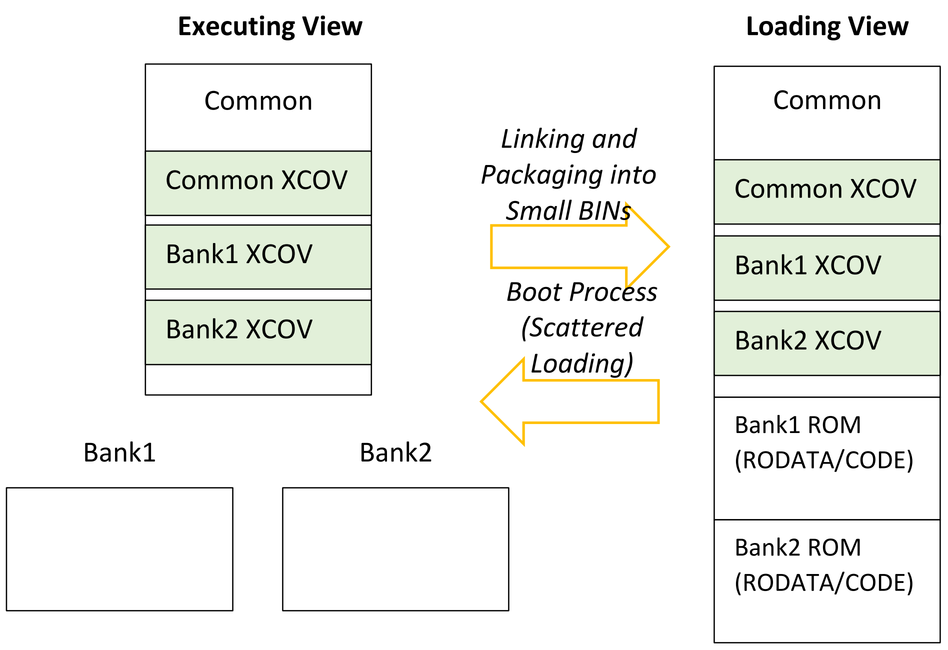


*Figure 2 The XCOV data of Bank under Bank/Overlay Linking should be placed in resident memory.*

## 2.6 Adapting lib-xcov library to target environment

### 2.6.1 Target environment for ARM/ARC/RISCV and other environments

The XCC system provides basic library files such as xcov_lib.c and xcov_lib_arm.c. When the target environment is ARM/ARC/RISCV (based on xcov_lib_arm.c), there are three functions in the lib-xcov library that need to be adapted:

- xcov_dump_write: outputs coverage data to the host, which can be temporarily stored in the target's flash or RAM and retrieved in subsequent processes, or directly output via UART and other buses.
  *static void xcov_dump_write(const void *data, int len) { … }*
- xcov_lock/xcov_unlock: for single-core critical adaptation, interrupt switching is generally sufficient (note possible nesting, such as in the case of interrupt disablement). If there is an RTOS, avoid using the RTOS-level MUTEX mechanism based on performance considerations. For multi-core critical adaptation, use interrupt switching + spin lock. If there is an RTOS (SMP) support, also avoid using its RTOS-level MUTEX mechanism.
  *static void *xcov_lock() { … }*
  *static void xcov_unlock(void *status) { … }*

Finally, the trigger coverage data dump process needs to be implemented - call xcov_dump.

### 2.6.2 Target environment for HOST

For example, when compiling the target through gcc on Windows or Linux, the XCC automatically provides (links) the lib-xcov code library under the host environment. The library function no longer calls xcov_dump_write but instead directly xcov_dump_file to the host file system, and xcov_dump is automatically called when the process exits. Currently, the absolute path of outputting coverage data to the .xcovdat file is determined at compile time, so it is recommended to copy all .xcovdat files and executable programs generated during compilation to another host environment:

- Establish the same path and copy all .xcovdat files generated during compilation and executable programs.
- After execution is complete, merge the .xcovdat of various hosts through XCovPost.

# 3 XCovPost Interfaces

## 3.1 Help for command

```
>xcovpost -h

Usage: xcovpost [options] <input-data-file>
  General options:
    -h/--help
      Prints this message
    -v/--version
      Prints the version info of the tool.
    -i <directory>
      Specify the input *.xcovdat directory to be merged.
    -o <directory>
      Specify the target directory of the database.
    --export[-force]
      Use exporting mode instead of merging one.
      In the exporting mode, the *.xcovsrc/*.xcovdat will be exported.
      --export-force: the existed files will be replace without prompt.
  <input-data-file>:
    Specify the input coverage data file path, it may be encoded or a .xcovdat file.

This XCovPost tool is using to merge the coverage data to the database.

##################################################################################
XCovPost Release V1.00 Version, 2 July 2022.
Copyright (C) 2022 (Any question, please contact me: lzfzmpizer@sina.com).
##################################################################################
```

## 3.2 Exporting function

If you need to backup the *.xcovsrc/*.xcovdat files after compiling, you can use the export function to do so. For example: *xcovpost.exe -i <input-dir> -o <export-dir> --export-force*

The export function also has a merge feature. For example, if a project is divided into several independent sub-parts for coverage instrumentation, they can be merged together using the export function: *xcovpost.exe -i <dir1> -i <dir2> -i <dir3> -o <export-dir> --export*

```
>xcovpost.exe -i output_clang_cov –o out_clang_export --export-force
XCovPost Scanning the input directories ...
  [XCovPost] Input 64 valid .xcovdat files
  [XCovPost] Export to myexport\obj\clang\clang_allocator.xcovdat ...
  [XCovPost] Export to myexport\obj\clang\clang_compound_statement_stat.xcovdat ...
  [XCovPost] Export to myexport\obj\clang\clang_func_body_stat.xcovdat ...
  [XCovPost] Export to myexport\obj\clang\clang_ir_common.xcovdat ...
  [XCovPost] Export to myexport\obj\clang\clang_ir_declarator.xcovdat ...
  [XCovPost] Export to myexport\obj\clang\clang_ir_expression.xcovdat ...
...
  [XCovPost] Export to myexport\obj\utest_xcc_main_general_6_listenv\test_api.xcovdat ...
  [XCovPost] Export to myexport\obj\xcc_lib\xcc_crc16.xcovdat ...
  [XCovPost] Export complete (64)!
```

### 3.3 Merging the coverage data from target board

To incorporate the coverage data of the target board, which can be redirected to UART or FLASH, its coverage data for each file can be saved separately or merged into one file.

We need to use the XCovPost command to merge the above coverage data into the coverage database generated during construction, namely the *.xcovsrc and *.xcovdat files. The XCovPost command provides automatic searching for the target database file by specifying the directory where the database is located, for example:

```
>xcovpost.exe -o output_clang_cov test_cov_data.txt
XCovPost Scanning the target directories ...
  [XCovPost] Merge the "test_cov_data.txt" to ... [58 valid .xcovdat target files]
  [XCovPost] Merge the "test_cov_data.txt" ... Okay [1 files]
```

### 3.4 Merging .xcovdat file

Especially for applications on HOST that may be tested on multiple servers, with their respective coverage data output to each server (.xcovdat file), the coverage data needs to be aggregated to generate a final unified report.

The XCovPost command supports both merging a single .xcovdat into the target directory and merging all .xcovdat files in a source directory into the target directory. The XCovPost command will automatically equip them with their compilation times to ensure correct merging.

```
>xcovpost.exe -o output_clang_cov -i output_clang_cov
XCovPost Scanning the input directories ...
  [XCovPost] Input 58 valid .xcovdat files
XCovPost Scanning the target directories ...
  [XCovPost] Merge 58 .xcovdat files to ... [58 .xcovdat target files]
  [XCovPost] Merge output_clang_cov\obj\clang\clang_allocator.xcovdat ... Okay
  [XCovPost] Merge output_clang_cov\obj\clang\clang_compound_statement_stat.xcovdat ... Okay
  [XCovPost] Merge output_clang_cov\obj\clang\clang_func_body_stat.xcovdat ... Okay
  [XCovPost] Merge output_clang_cov\obj\clang\clang_ir_common.xcovdat ... Okay
  [XCovPost] Merge output_clang_cov\obj\clang\clang_ir_declarator.xcovdat ... Okay
  [XCovPost] Merge output_clang_cov\obj\clang\clang_ir_expression.xcovdat ... Okay
  [XCovPost] Merge output_clang_cov\obj\clang\clang_ir_func.xcovdat ... Okay
  [XCovPost] Merge output_clang_cov\obj\clang\clang_ir_tp.xcovdat ... Okay
...
```

# 4 XCovHTML Interfaces

## 4.1 Help for command

```
>xcovhtml -h

Usage: xcovhtml [options]
  General options:
    -h/--help
      Prints this message
    -v/--version
      Prints the version info of the tool.
    -d <directory>
      Specify the input directory.
    -o <directory>
      Specify the output directory.

This XCovHtml tool is using to generate HTMLs reports for coverage info.

##################################################################################
XCovHtml Release V1.00 Version, 2 July 2022.
Copyright (C) 2022 (Any question, please contact me: lzfzmpizer@sina.com).
##################################################################################
```

## 4.2 Outputting an HTML report of the coverage database directory

XCovHTML provide the ability to output an HTML report of the coverage database directory, which is generally divided by module. For example:

```
>xcovhtml.exe -d output_clang_cov -o output_clang_cov/xcov
XCovHtml There's 42 .xcovdat files found!
  [XCovHtml] processing file ../code/clang/clang_func_body_stat.h ... decisions 0/10, functions: 0/6
  [XCovHtml] processing file ../code/clang/clang_ir.h ... decisions 0/145, functions: 0/143
  [XCovHtml] processing file ../code/clang/clang_ir_expression.h ... decisions 0/3, functions: 0/3
  [XCovHtml] processing file ../code/clang/clang_pars.h ... decisions 0/9, functions: 0/9
…
  [XCovHtml] processing file c:\mingw\include\wchar.h ... decisions 0/1, functions: 0/1
  [XCovHtml] processing file c:\mingw\include\winnt.h ... decisions 0/4, functions: 0/4
  [XCovHtml] processing file c:\mingw\include\winsock.h ... decisions 0/14, functions: 0/4
  [XCovHtml] total ... decisions 2679/4882, functions: 514/913
…
```

## 4.3 Filtering configurations for outputting HTML reports

Filtering configurations can be specified in the xcovhtml-skips.conf file in the same directory as the XCovHTML command. The prefix directories to be filtered can be specified in this file. For example:

```
c:/msys64/mingw64/
c:/MinGW/include/
c:/MinGW/lib/
```

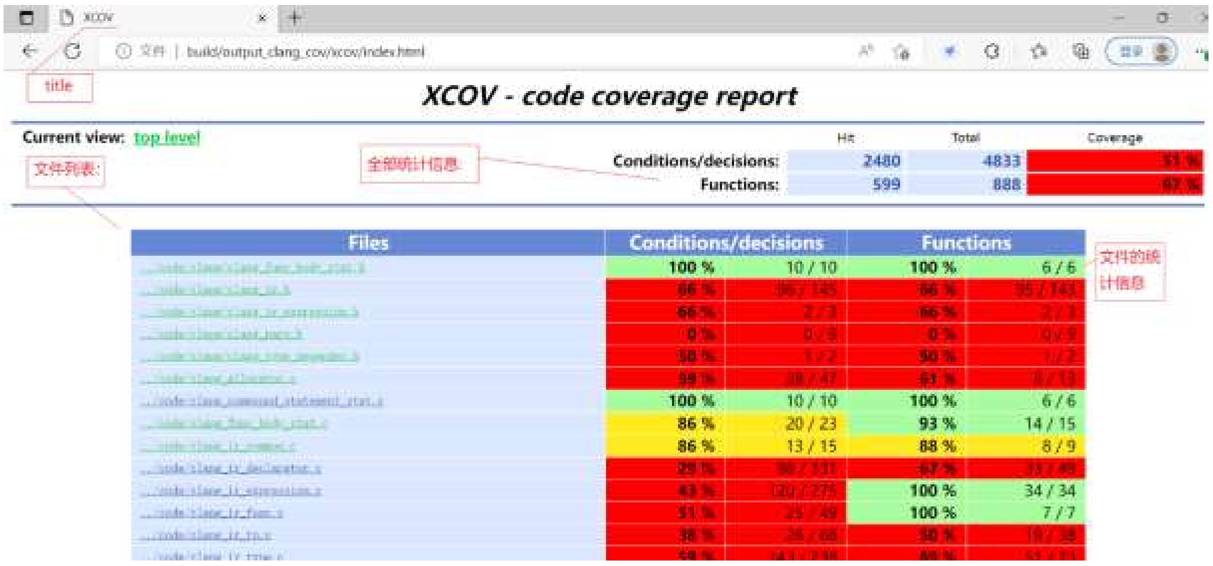
## 4.4 View HTML report

The HTML report generated by the XCovHTML command is placed in the specified output directory (or the current working directory if unspecified), where the index.html file can be found and opened using a browser.

### 4.4.1 The overview of the HTML report

The index.html file can be opened using a browser after it has been generated by the XCovHTML command.

- The overall summary report title shows "XCOV," while the titles of each detailed file show the source path.
- The overall summary provides the total information for all files' coverage conditions/decisions as well as the coverage status of all functions.
- The file list summarizes all sources (source files or header files with inline functions) and their coverage status for conditions/decisions and functions.



**XCOV - code coverage report**

Current view: top level

| | Hit | Total | Coverage |
|---|---|---|---|
| Conditions/decisions: | 2480 | 4833 | 51 % |
| Functions: | 599 | 888 | 67 % |

| Files | Conditions/decisions | | Functions | |
|---|---|---|---|---|
| [illegible] | 100 % | 10 / 10 | 100 % | 6 / 6 |
| [illegible] | 66 % | 96 / 145 | 66 % | 95 / 143 |
| [illegible] | 66 % | 2 / 3 | 66 % | 2 / 3 |
| [illegible] | 0 % | 0 / 9 | 0 % | 0 / 9 |
| [illegible] | 50 % | 1 / 2 | 50 % | 1 / 2 |
| [illegible] | 59 % | 28 / 47 | 61 % | 8 / 13 |
| [illegible] | 100 % | 10 / 10 | 100 % | 6 / 6 |
| [illegible] | 86 % | 20 / 23 | 93 % | 14 / 15 |
| [illegible] | 86 % | 13 / 15 | 88 % | 8 / 9 |
| [illegible] | 29 % | 38 / 131 | 67 % | 33 / 49 |
| [illegible] | 43 % | 120 / 275 | 100 % | 34 / 34 |
| [illegible] | 51 % | 25 / 49 | 100 % | 7 / 7 |
| [illegible] | 38 % | 26 / 68 | 50 % | 19 / 38 |
| [illegible] | 59 % | [illegible] | 69 % | [illegible] |

*Figure 3 The overview of the HTML report*

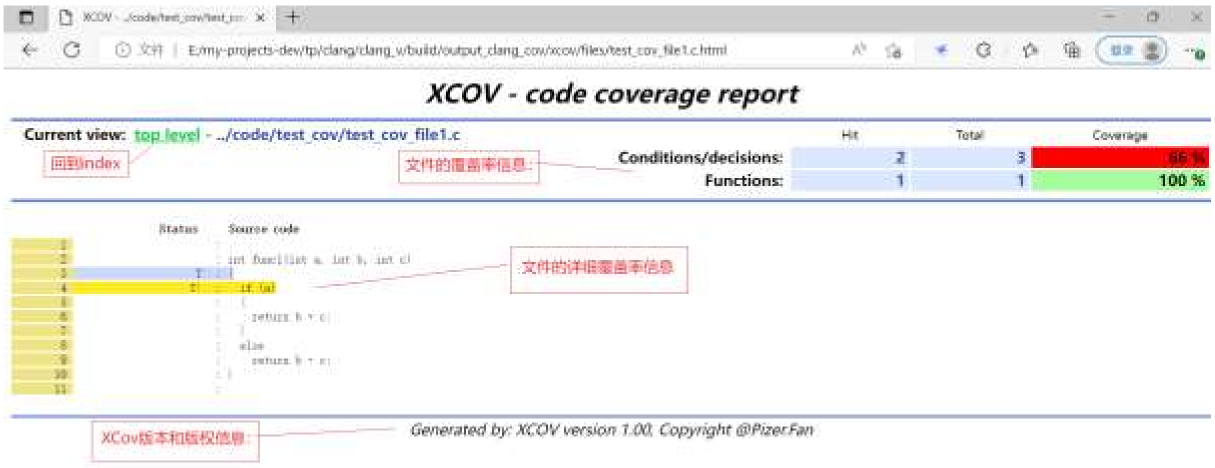
### 4.4.2 The detailed HTML report for files

After opening index.html, select the specified file to open:

- Coverage statistics information: the coverage status of the file's conditions/decisions and functions.
- Detailed coverage information of the file:
  - ✓ Code line column: corresponding to the line number of the source code.

- ✓ Status column: There are three types of statuses - 'T', 'F', and '!'. 'T' represents that the condition for true judgment has been covered, 'F' represents that the condition for false judgment has been covered, and '!' represents that some kind of judgment for this condition has not been covered. For a single status, it generally means that the true judgment has not been covered, and for a paired status, it alternates between TF to indicate which judgment has not been covered.
- ✓ Source code column: the specific content of the source code.

- ● XCov version and copyright information: indicates the XCov version information and copyright information.



E:/my-projects-dev/tp/clang/clang_v/build/output_clang_cov/xcov/files/test_cov_file1.c.html

**XCOV - code coverage report**

Current view: top level - ../code/test_cov/test_cov_file1.c

| | Hit | Total | Coverage |
|---|---|---|---|
| Conditions/decisions: | 2 | 3 | 66 % |
| Functions: | 1 | 1 | 100 % |

```
Status   Source code
         :
         : int func1(int a, int b, int c)
     T   : {
    T!   :   if (a)
         :   {
         :     return b + c;
         :   }
         :   else
         :     return b + c;
         : }
         :
```

*Generated by: XCOV version 1.00, Copyright @PizerFan*

Figure 4 The detailed HTML report for files

# 5 XCScanView Interfaces

## 5.1 Help for command

```
> xcscanview -h
  General options:
    -h/--help
      Prints this message
    -v/--version
      Prints the version info of the tool.
    -i <directory>
      Specify the input *.cscandat directory.
    --top <top-level>
      Specify the top level, such as 10.
    --by <mt|cl|cc|all>
      Sort by:
       *mt - maintainability
        cl - code lines
        cc - cyclomatic complexity

This XCScanView tool is using to view the codescan info.

#####################################################################################
XCScanView Development V1.00 Version, 2 July 2022.
Copyright (C) 2022 (Any question, please contact me: lzfzmpizer@sina.com).
#####################################################################################
```

## 5.2 View the maintainability ranking of functions

View the maintainability ranking of functions, for example:

```
>xcscanview -i output_clang_cov --top=20 –by=mt
XCScanView Scanning the input directories ...
  [XCScanView] Top 20 by maintainability
    [01]  0 : clang_pars_yyparse in ../code/clang_yacc.c
    [02]  6 : yylex_ in ../code/clang_lex.c
    [03] 33 : tsc_strdb_yylex in ../code/xcc/tsc_strdb/tsc_strdb_lex.c
…
    [20] 56 : yy_get_next_buffer in ../code/clang_lex.c
```

The lower the maintainability value of a function, the greater the difficulty in maintaining it, so the higher its ranking - used as a warning.

## 5.3 View the code line ranking of functions

View the code line ranking of functions, for example:

```
>xcscanview -i output_clang_cov --top=20 –by=cl
XCScanView Scanning the input directories ...
  [XCScanView] Top 20 by code lines
    [01] 2549 : clang_pars_yyparse in ../code/clang_yacc.c
    [02] 1168 : yylex_ in ../code/clang_lex.c
    [03] 400 : tsc_strdb_yylex in ../code/xcc/tsc_strdb/tsc_strdb_lex.c
…
    [20]  87 : iRbtree_TestMain in ../code/iLib/test_irbtree.c
```

The larger the number of code lines in a function, usually indicating greater difficulty in maintenance, so the higher its ranking - used as a warning.

## 5.4 View the cyclomatic complexity ranking of functions

View the cyclomatic complexity ranking of functions, for example:

```
>xcscanview -i output_clang_cov --top=20 –by=cc
XCScanView Scanning the input directories ...
  [XCScanView] Top 20 by cyclomatic complexity
    [01] 395 : clang_pars_yyparse in ../code/clang_yacc.c
    [02] 196 : yylex_ in ../code/clang_lex.c
    [03] 77 : utest_type_depender in ../code/utest_clang_case1.c
 …
    [20] 15 : clang_relational_expr_get_value in ../code/clang_ir_expression.c
```

The higher the cyclomatic complexity value of a function, usually indicating greater difficulty in maintenance, so the higher its ranking - used as a warning.

## Appendix 1 Terminology

| NO. | Abbreviation | Comment |
|---|---|---|
| 1 | XCC | eXtended C Compiler, which is the name of the target tool designed in this article. It also refers to the XCC system, which is a collection of related toolchains. |
| 2 | LOS | Love Operating System - refers to the operating system designed by the author Pizer.Fan (supports polling scheduling, single-core real-time preemptive scheduling, SMP preemptive scheduling, etc.). LOS usually refers to the entire LOS platform - in addition to the LOS microkernel, it also includes various basic software modules, including memory management, inter-core communication, debugging components, finite state machine engines, and other middleware. |
| 3 | C99 | C99 is the informal name for ISO/IEC 9899:1999, released in 1999, and adopted by ANSI in March 2000. |
| 4 | ANSI C | ANSI C is a standard for the C language launched by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). ANSI C standardized existing implementations, while adding some content from C++ and supporting multiple character sets. The ANSI C standard also defines the standard C runtime library routines. Specifically, ANSI C refers to C89, which is called ANSI C because this standard was published by the American National Standards Institute (ANSI). |
| 5 | TSC | Trace SCan - specifically refers to the TSC tool, a TRACE ID scanning tool on the LOS platform, implemented by modifying the source code. |
| 6 | XCScanView | XCC Code Scan View - a tool used to view the intermediate database of code scanning. |
| 7 | XCov | XCC Coverage - specifically refers to test coverage, such as code line coverage, branch coverage, etc. |
| 8 | XCovPost | XCC Coverage Post - merges the runtime fragment data of the coverage output by the test target into the coverage database. |
| 9 | XCovHtml | XCC Coverage Html generator - generates an HTML format coverage report from the coverage database. |
| 10 | XAssertScan | XCC Assert Scan - a tool used to view ASSERT lines in the code, with a reminder function to support filtering white lists. |
| 11 | CC | In this article, there are two abbreviations:<br>● C Compiler: The C language compiler.<br>● Cyclomatic complexity: Also known as conditional complexity, it is a standard for measuring code complexity. |

*table 1 Terminology*

# Appendix 2 TSC Tool

The TSC (TraceSCan) tool converts log printing in the source code into "TRACEID" format, with the goal of saving the code size of the target program, improving logging performance and so on. For detailed guidance on the TSC tool, please refer to the "LS-TSC Tool User Manual".

Here is a simple example of using the TSC tool:

```
> ls-tsc.exe --db-path "..\tsc_db\tsc_db.cxx" -l ftl_file_list.txt
```

(1) ls-tsc.exe: the TSC executable program

(2) --db-path "..\tsc_db\tsc_db.cxx": specifies the ID database, which is a C++ language file that records all "TRACEID" converted by TSC. For forward compatibility, this database can generally only be appended.

(3) -l ftl_file_list.txt: specifies all files of the FTL module through ftl_file_list.txt. Of course, the files to be scanned can also be specified directly.

The above command will convert LS_LOG in the files listed in ftl_file_list.txt to ID printing, such as in a certain file:

```
/* ftl_sample.c */
...
  LS_LOG(CRIT, FTL, "Test FTL LOG!");
  LS_LOG(CRIT, FTL, "Test FTL LOG with p1 = %d!", 10);
->
  LS_LOG_ID(CRIT, FTL, "Test FTL LOG!", 0,(), Test_FTL_LOG_);
  LS_LOG_ID(CRIT, FTL, "Test FTL LOG with p1 = %d !", 1,(10), Test_FTL_LOG_with_p1__d_);
...
```

At this point, if you check the ID database - "..\tsc_db\tsc_db.cxx", you will find the added "TRACEID".

## Appendix 3 Code Metrics - Range and Meaning of Maintainability Index [Microsoft]

Originating from the official website of Microsoft, the maintainability index is a computed value ranging from 0 to 100, representing the relative difficulty of maintaining code. Higher values indicate greater maintainability. Color-coded ratings are used to quickly identify trouble spots in codebases: green ratings between 20 and 100 indicate good maintainability, yellow ratings between 10 and 19 indicate moderate maintainability, and red ratings between 0 and 9 indicate low maintainability. For further details, please refer to the range and meaning of maintainability index in code metrics (see Code metrics - Maintainability index range and meaning - Visual Studio (Windows) | Microsoft Docs).

The metric originally was calculated as follows:

```
Maintainability Index = 171 - 5.2 * ln(Halstead Volume) - 0.23 * (Cyclomatic
Complexity) - 16.2 * ln(Lines of Code)
```

The use of this formula meant that it ranged from 171 to an unbounded negative number. As code tended toward 0, it was clearly hard to maintain code and the difference between code at 0 and some negative value was not useful. As a result of the decreasing usefulness of the negative numbers and a desire to keep the metric as clear as possible, we decided to treat all 0 or less indexes as 0 and then rebase the 171 or less range to be from 0 to 100. For this reason, the formula we use is:

```
Maintainability Index = MAX(0,(171 - 5.2 * ln(Halstead Volume) - 0.23 * (Cyclomatic
Complexity) - 16.2 * ln(Lines of Code))*100 / 171)
```

In addition to that, we decided to be conservative with the thresholds. The desire was that if the index showed red then we would be saying with a high degree of confidence that there was an issue with the code. This gave us the following thresholds:

For the thresholds, we decided to break down this 0-100 range 80-20 to keep the noise level low and we only flagged code that was suspicious. We've used the following thresholds:

- 0-9 = Red
- 10-19 = Yellow
- 20-100 = Green

## Appendix 4 Brief introduction to Cyclomatic Complexity.

Refer to Code metrics - Cyclomatic complexity - Visual Studio (Windows) | Microsoft Docs 或者 Introducing Cyclomatic Complexity in C# | CodeGuru.com ， Cyclomatic complexity can be calculated as follows:

$$M = E - N + 2 * P$$

Where:

- M = cyclomatic complexity
- E = number of edges of the graph
- N = number of nodes of the graph
- P = number of connected components

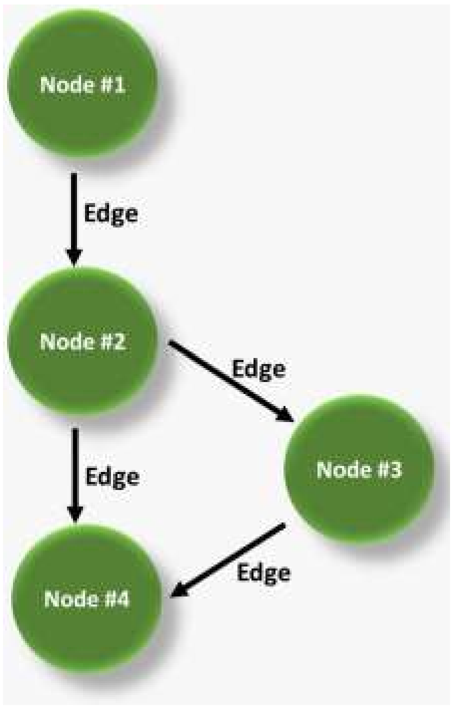


*图 1 Illustrating Cyclomatic Complexity*

## Appendix 5 Measurement standard for Cyclomatic Complexity from an online source

A quote from an online source states: "Code with low complexity is not necessarily good, but code with high complexity is definitely not good. The cyclomatic complexity is associated with the code condition, testability, and maintenance cost" The measurement standard for cyclomatic complexity is as follows:

| Cyclomatic Complexity | Code Condition | Testability | Maintenance Cost |
|---|---|---|---|
| 1 - 10 | Clear, Structured | High | Low |
| 10 - 20 | Complex | Medium | Medium |
| 20 - 30 | Very complex | Low | High |
| >30 | Unreadable | Untestable | Very high |

*table 2 Measurement standard for Cyclomatic Complexity*

Note: "Testability" refers to the ease of testing or verifying the correctness of the code.