LAUTERBACH
DEVELOPMENT TOOLS

Application Note for Trace-Based Code Coverage

Release 09.2024

Application Note for Trace-Based Code Coverage

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents .... 🗁

Code Coverage .... 🗁

Application Note for Trace-Based Code Coverage 1

History 6

Intended Audience 7

Introduction 8

Supported Code Coverage Metrics 8

Code Coverage and Certification 9

Trace-Based Code Coverage 11

Introduction to the Approach 11

Processors/Chips Suitable 13

Code Coverage Measurement 14

Evaluation of the Code Coverage Measurement 14

Report Generation 15

MC/DC, Condition and Decision Coverage 16

Multiple Code Coverage Modes 16

Preconditions for a Trace-Based Code Coverage 16

The Individual Code Coverage Modes 17

A Comparison of the Different Code Coverage Modes 19

Causes for Observability Gaps: An Overview 20

Evaluation of Switch Case Statements 21

Code Coverage Workflows 22

Workflows for Source Code Metrics 22

General Procedure 22

Statement Coverage Workflow 23

Condition Coverage Workflow 28

Decision Coverage Workflow 30

MC/DC Workflow 32

Function Coverage Workflow 34

Call Coverage Workflow 36

Workflows for Address-Based Metrics 38

General Procedure 38

Object Code Coverage Workflow 39

Object Code Based (ocb) Decision Coverage Workflow 41

Build Process 43

Introductory Notes 43

General Recommendations for the Build Toolchain 43

Build Process Requirements for All Code Coverage Metrics at a Glance 43

Verification of Alignment with Production Code 45

Build Process Call Coverage 46

Build Process MC/DC, Condition and Decision Coverage 47

Decision Making Process 47

Build Process for Code Coverage Mode Targeted Instrumentation/No Instrumentation 52

Build Process Code Coverage Mode Breakpoint Assisted 57

Build Process Code Coverage Mode Full Instrumentation 58

Trace Data Collection Overview 60

SMP Multicore Systems 60

iAMP Multicore Systems 60

TRACE32 Tool Configurations 61

Choose the Appropriate Trace Data Collection Variant 62

Best Practices for Trace Recording 64

Reduce the Amount of Trace Data 64

Ensure a Fault-Free Trace Recording 65

Disable Timestamps for Trace Streaming 65

Steps in Preparation for Trace Data Collection 67

Notes on the Individual Test Variants 67

Preparation for Statement, Function, Object Code, ocb Decision Coverage 68

Preparation for Call Coverage 69

Preparation for MC/DC, Condition and Decision Coverage 70

Preparation for Targeted Instrumentation/No Instrumentation 70

Preparation for Code Coverage with Breakpoints 72

Preparation for Full Instrumentation 75

Trace Data Collection and Code Coverage Measurement 77

Incremental Code Coverage 77

Data Collection 77

Example Script 79

Summary 79

Incremental Code Coverage in STREAM Mode 80

Data Collection 80

Example Script 83

Summary 83

RTS Mode Code Coverage 84

Data Collection 84

Example Scripts 87

Summary 89

SPY Mode Code Coverage 90
Operation States 90
Data Collection 92
Example Script 94
Summary 95
Code Coverage with Virtual Targets 96
ART Mode Code Coverage 98
Data Collection 99
Example Script 100

Code Coverage Evaluation in TRACE32 101
Object Code Coverage Evaluation in TRACE32 101
Evaluation 101
Example Script 105
Object Code Based (ocb) Decision Coverage Evaluation 106
Evaluation Strategy 106
Evaluation 107

Comment Your Results 112

TRACE32 Merge and Report Tool 114

Appendix A: TRACE32 Coverage Report Utility 117

Appendix B: Assemble Multiple Test Runs at Address Level 119
Save and Restore Code Coverage Measurement 119
Save and Restore Trace Recording 121

Appendix C: Assembler-Only Functions and Code Coverage 123
Object Code Coverage 123
Source Code Metrics 124

Appendix D: Data Coverage 126
Trace Data Collection 126
Evaluation 127

Appendix E: Trace Decoding in Detail 130
Trace Decoding for Static Applications 130
Decoding in Stopped State for Static Applications 130
Decoding in Running State for Static Applications 130
RTS Decoding for Static Applications 131
Trace Decoding for Applications Using a Rich OS 132
Decoding in Stopped State (Rich OS) 132
Decoding in Running State (Rich OS) 132
RTS Decoding (Rich OS) 132

Appendix F: Coding Guidelines 134

Appendix G: Object Code Coverage Tags in Detail 137
Standard Tags 137

Tags for Arm-ETMv1/v3/v4 for Arm/Cortex Architecture 138

Appendix E: Data Coverage in Detail 140

Version 05-Oct-2024

History

08-Aug-2024	Chapter 'Build Process' revised.
08-Aug-2024	Since statement coverage, decision/condition/MCDC coverage, and function/call coverage are preferably evaluated in a web browser, the evaluation chapters for TRACE32 have been removed.
10-Jul-2024	Description of --filelist parameter added to chapter 'TRACE32 Merge and Report Tool'.
02-Jul-2024	“Notes on Branch Coverage” added to chapter 'Code Coverage and Certification'.
02-Jul-2024	“Notes on Statement Coverage” added to chapter 'Statement Coverage Workflow“.
19-Jun-2024	Chapter 'Introduction' revised.
29-May-2024	Subchapter 'Evaluation of Switch Case Statements' added to chapter 'MC/DC, Condition and Decision Coverage'.
26-Jan-2024	The manual has been completely revised to integrate the new code coverage modes targeted and full instrumentation.
07-Sep-2023	EN50128 (railway) added to 'Trace-Based Code Coverage and Certification'. The chapter now also lists the safety levels and the TRACE32 tool classification of the individual standards.
19-Aug-2020	Initial version of the manual.

Intended Audience

This manual is intended for the following users:

- Those who create executable files for measuring code coverage
- Those who perform code coverage measurements
- Those who evaluate code coverage measurements
- Those who generate code coverage reports

Although this is a general manual, the screenshots were taken using a TriCore™ AURIX™ TC297T, unless stated otherwise. Your screen may look different.

You only need to read the relevant chapters of this manual. Reading the entire manual may result in some repeated information.

Supported Code Coverage Metrics

TRACE32 supports the following code coverage metrics:

- **Statement coverage**

  Statement coverage ensures that every statement in the program has been invoked at least once.

- **Condition coverage**

  All conditions in the program have evaluated both true and false.

- **Decision coverage**

  Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once.

- **MC/DC coverage**

  Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once. And each condition in a decision is shown to independently affect the outcome of that decision.

- **Function coverage**

  Every function in the program has been invoked at least once.

- **Call coverage**

  Every function call has been executed at least once.

- **Cbject code coverage**

  Object code coverage ensures that each object code instruction was executed at least once and all conditional instructions (e.g. conditional branches) have evaluated to both true and false.

Code Coverage and Certification

Measuring code coverage is a prerequisite for certification in order to evaluate the completeness of test cases and to prove that no unintended functionality is present. TRACE32 supports the following standards:

- **DO-178C (avionics)**

  Safety integrity levels: five levels from E to A, with level A being the highest level

  Tool classification for TRACE32 code coverage: TQL-5

  Supported code coverage metrics: statement coverage, decision coverage, MC/DC

- **EN 50128 (railway)**

  Safety integrity levels: five levels, SIL 0 to 4, with SIL 4 being the highest level

  Tool classification for TRACE32 code coverage: T3

  Supported code coverage metrics: statement coverage, branch coverage (decision coverage in TRACE32), compound condition coverage (condition coverage in TRACE32)

- **IEC 61508 (industrial)**

  Safety integrity levels: five levels, basic integrity, SIL 1 to 4, with SIL 4 being the highest level

  Tool classification for TRACE32 code coverage: T3

  Supported code coverage metrics: statement coverage, branch coverage (decision coverage in TRACE32), condition coverage, MC/DC as well as function coverage

- **IEC 62304 (medical)**

  Safety integrity levels: three levels, class A to C, with class C being the highest level

  Tool classification for TRACE32 code coverage: T3

  Supported code coverage metrics: the standard does not contain any directives in this regard; select suitable subset according to software development plan

- **ISO 26262 (automotive)**

  Safety integrity levels: five levels, QM, ASIL A to D, with ASIL D being the highest level

  Tool classification for TRACE32 code coverage: TCL2/3

  Supported code coverage metrics: statement coverage, branch coverage (decision coverage in TRACE32), condition coverage, MC/DC as well as function coverage.

For those whose application requires tool qualification, Lauterbach offers a Tool Qualification Support Kit (TQSK for short). It contains everything needed to qualify a TRACE32 tool for use in safety-critical projects. If you are interested, refer to the **TRACE32 customer portal**.

Standards like ISO 26262 require branch coverage. Due to the similarity between branch coverage and decision coverage, Lauterbach considers it justified to offer only decision coverage. How does Lauterbach justify this? Let's first take a look at the definitions for the two metrics.

Definition of Branch Coverage: Branch coverage measures whether all possible branches of every conditional statement in the source code have been executed.

Definition of Decision Coverage: Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once.

Decision coverage is somewhat stricter as it must consider decisions within assignments, such as `a = b || (c && d);`. Although the two metrics differ in the calculation of the reported coverage rate, this simplification can be justified with regard to the definitions.

Trace-Based Code Coverage

Introduction to the Approach

Before we delve into TRACE32 trace-based code coverage, let's first examine conventional code coverage.

Conventional code coverage operates by instrumenting the source code so that coverage data is stored in the target's RAM during test execution. Once the test run is complete, the conventional code coverage tool retrieves and processes this data for code coverage analysis.

Now, let's move on to TRACE32 trace-based code coverage which requires two main conditions:

1. The core(s)-under-test must have the capability to generate trace data to monitor the program flow.
2. TRACE32 works best with low compiler optimization levels for easier object-to-source code mapping. Consequently, TRACE32's trace-based code coverage cannot be conducted on production code.

   The code coverage analysis in TRACE32 relies on object code, as only this is captured in the program flow trace recording. Source code lines are tagged for code coverage based on an appropriate mapping between the object code and the source code. This mapping works better when a lower level of compiler optimization is used.

During testing, trace data on the program flow is collected. TRACE32 retrieves and processes this data for code coverage analysis.

For complex metrics such as Modified Condition/Decision Coverage (MC/DC), condition coverage, and decision coverage, it may be necessary to instrument individual lines of source code. TRACE32's lightweight instrumentation has only a minimal impact on code size and timing behavior.

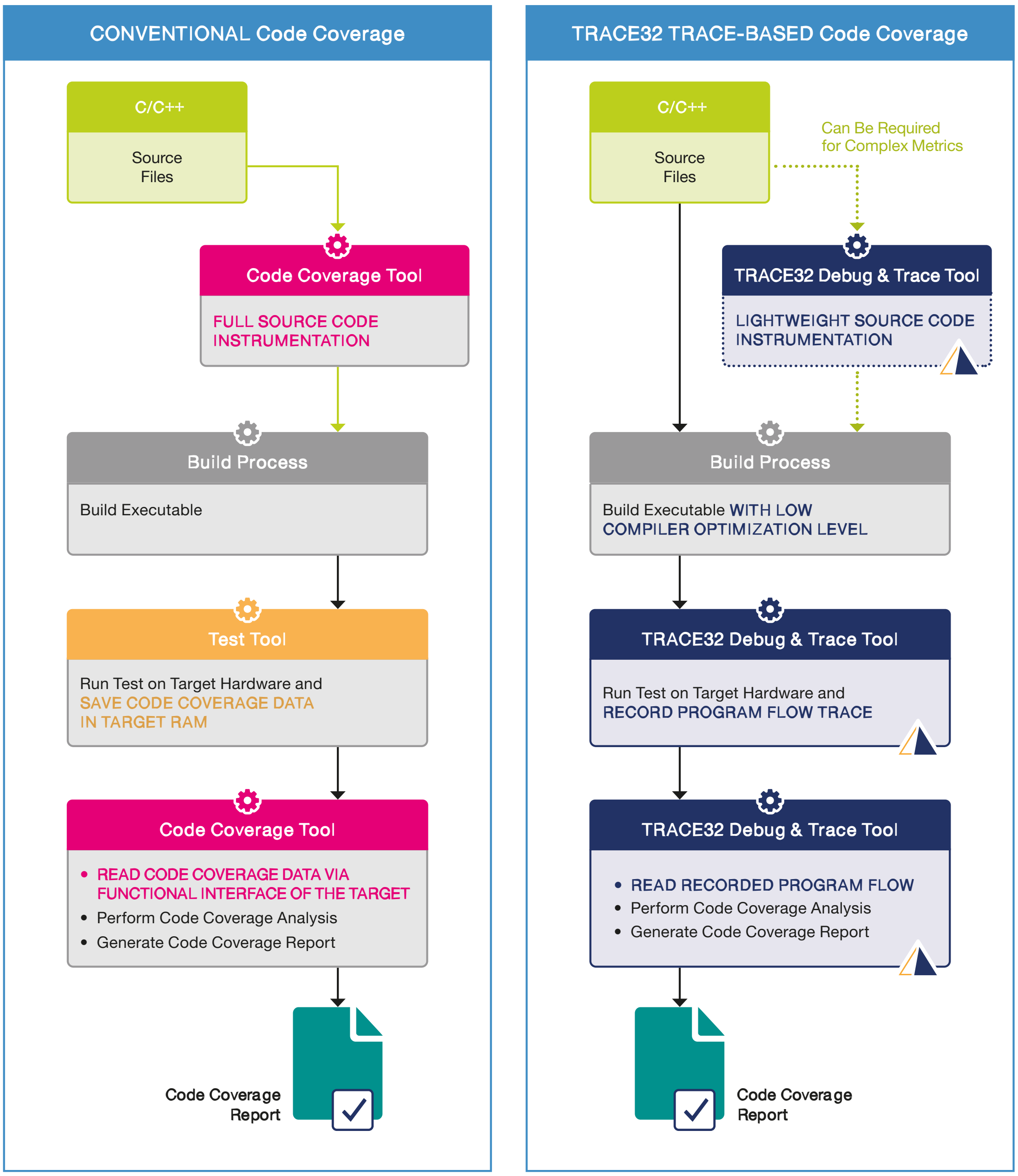

CONVENTIONAL Code Coverage
C/C++
Source Files
Code Coverage Tool
FULL SOURCE CODE INSTRUMENTATION
Build Process
Build Executable
Test Tool
Run Test on Target Hardware and SAVE CODE COVERAGE DATA IN TARGET RAM
Code Coverage Tool
• READ CODE COVERAGE DATA VIA FUNCTIONAL INTERFACE OF THE TARGET
• Perform Code Coverage Analysis
• Generate Code Coverage Report
Code Coverage Report
TRACE32 TRACE-BASED Code Coverage
C/C++
Source Files
Can Be Required for Complex Metrics
TRACE32 Debug & Trace Tool
LIGHTWEIGHT SOURCE CODE INSTRUMENTATION
Build Process
Build Executable WITH LOW COMPILER OPTIMIZATION LEVEL
TRACE32 Debug & Trace Tool
Run Test on Target Hardware and RECORD PROGRAM FLOW TRACE
TRACE32 Debug & Trace Tool
• READ RECORDED PROGRAM FLOW
• Perform Code Coverage Analysis
• Generate Code Coverage Report
Code Coverage Report

Figure: Workflow comparison, conventional code coverage vs. TRACE32 trace-based code coverage.

TRACE32 trace-based code coverage is characterized by the following:

- No additional target resources are required beyond the program flow trace.
- Lightweight instrumentation results in minimal code and time overhead.
- It supports a wide range of code coverage metrics.
- It can be used in all test phases.
- It supports both C and C++.
- It can be used to generate comprehensive reports.
- Complete test automation is possible with TRACE32 PRACTICE, Python, or the TRACE32 Remote API.

Processors/Chips Suitable

The question now arises: which processors/chips have a trace interface suitable for code coverage measurement with TRACE32?

- **All processors/chips with an off-chip trace interface are suitable**

  You can find these processors/chips on the page **https://www.lauterbach.com/supported-platforms/chips**, where they are tagged with "Off-Chip Trace" in the "Supported TRACE32 Solutions" column.

- **Some processors/chips with an on-chip trace are suitable**

  Processors/chips with on-chip trace are tagged with "On-Chip Trace" in the "Supported TRACE32 Solutions" column on the page **https://www.lauterbach.com/supported-platforms/chips**. The on-chip trace should be at least 1 MB in size so that it makes sense for the TRACE32 code coverage.

- **Some chips that allow debugging and tracing via the USB stack are suitable**

  You can find these processors/chips on the page **https://www.lauterbach.com/supported-platforms/chips**, where they are tagged with "USB Direct" in the "Supported TRACE32 Solutions" column. However, it's always best to consult with Lauterbach's sales team to confirm compatibility.

For the processors/chips mentioned above, code coverage measurement is conducted on the target hardware. In the early stages of testing, code coverage measurement can also be performed using simulators or virtual targets. The safety standards allow this for the test phases software unit and module integration testing. See also **TRACE32 Instruction Set Simulator and ISO 26262**. However, virtual targets become slow due to tracing and require a significant amount of memory.

If Lauterbach does not offer a TRACE32 Instruction Set Simulator for the core architecture you are using, you can also use the **TRACE32 Advanced Register Trace** (**Trace.METHOD ART**). This is a single-step trace, which makes program execution very slow. This method is therefore only suitable for unit testing.

Code Coverage Measurement

TRACE32 offers two variants for code coverage measurement:

Incremental Code Coverage

In this variant, the two steps of RECORDING and PROCESSING are repeated consecutively until sufficient measurement data has been recorded and processed.

RECORDING: Run the program and record the program flow in the trace memory. Stop the program execution when the trace memory is full.

PROCESSING: Read the trace content, process it, and save the code coverage results in the TRACE32 internal code coverage system.

Continuous Code Coverage

This variant allows code coverage processing while the program is running. However, it is only compatible with processors/chips that support off-chip trace. The following steps occur simultaneously:

- Execute the program and record the program flow in the trace memory.
- Stream the recorded program flow to the host.
- Process the program flow and save the code coverage results in the TRACE32 internal code coverage system.

Continuous code coverage requires simpler scripts and is naturally faster due to the simultaneity of the steps. However, it is only effective for off-chip trace up to a certain bandwidth. Incremental code coverage, on the other hand, requires more complex scripts and is slower, but it has the advantage of being consistently reliable.

Evaluation of the Code Coverage Measurement

TRACE32 offers two variants for code coverage evaluation:

- **Web Browser Variant**

  This variant is recommended for code coverage metrics such as statement, decision, condition, MC/DC, call, and function coverage.

  Typically, code coverage is not measured in a single test run but is approached incrementally. This requires consolidating multiple code coverage measurements into a summarized report. This is done in two steps:

  - Export the result of each individual code coverage measurement stored in TRACE32's internal coverage system to a JSON file.

  - Merge the JSON files and generate an HTML file to evaluate a specific code coverage metric.

- **Evaluation in TRACE32**

  The individual code coverage measurement results for address-based code coverage metrics, including object code and object code-based decision coverage, can be merged and evaluated within the TRACE32 PowerView GUI.

Report Generation

The HTML file generated for evaluating code coverage measurements in a web browser can also serve as a report.

When evaluating in the TRACE32 PowerView GUI, you can generate an HTML report at any time using the TRACE32 Coverage Report Utility.

MC/DC, Condition and Decision Coverage

Mastering these metrics presents a slightly greater challenge.

- Achieving complete code coverage may require the instrumentation of individual lines of source code or marking them with breakpoints. Lauterbach offers multiple code coverage modes for this purpose.
- TRACE32 must convert case statements into if-then expressions to perform code coverage.

Multiple Code Coverage Modes

This chapter needs you to know exactly what a decision and a condition are. So, just to make sure, here's an explanation.

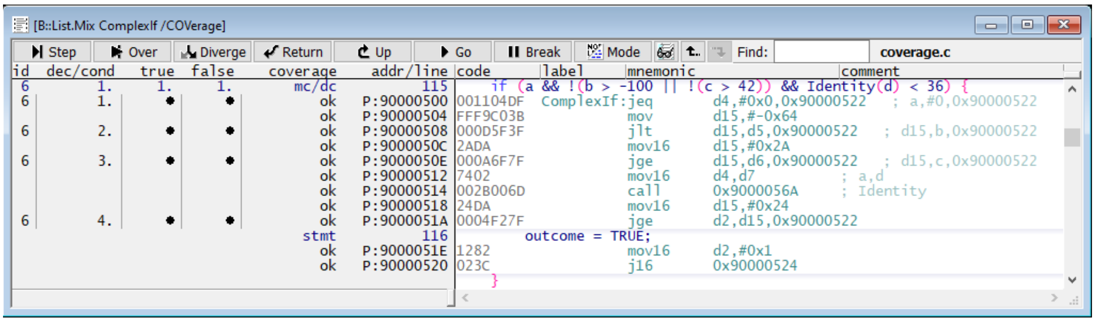
```
while ((( !(Identity(a) >= -45) && Identity(b)) && Identity(c) || d
```

- A condition (yellow in the line above) is a logical indivisible, atomic expression. It can only be true or false.
- A decision (framed by turquoise rectangle) is a logical expression which can be composed of several (atomic) conditions separated by logical operators such as ||, &&, !. It results in true or false.

Preconditions for a Trace-Based Code Coverage

For MC/DC, condition, or decision coverage evaluation to be conducted based on the recorded program flow, four criteria must be met:

1. TRACE32 needs to understand the structure and location of conditions and decisions within the source code. Since the compiler-generated debug information does not include these details, Lauterbach provides a Clang-based command-line tool called t32cast. This tool analyzes the C/C++ source files and generates an extended code analysis (.eca) file for each source file, supplying the required condition and decision details.
2. Decisions consist of one or more atomic conditions. Each condition in the source code must be represented by a conditional branch or a conditional instruction at the object code level.
3. An exact mapping between the conditions/decisions in the source code and the conditional branches/instructions in the object code is required.
4. The conditional branches or instructions in the recorded program flow trace must enable the observation of whether a source code condition was evaluated as true or false.



```
id  dec/cond  true  false   coverage    addr/line code      label       mnemonic    comment
6        1.     1.     1.      mc/dc          115     if (a && !(b > -100 || !(c > 42)) && Identity(d) < 36) {
6        1.      ●      ●         ok  P:90000500 001104DF  ComplexIf:jeq     d4,#0x0,0x90000522   ; a,#0,0x90000522
                                  ok  P:90000504 FFF9C03B            mov     d15,#-0x64
6        2.      ●      ●         ok  P:90000508 000D5F3F            jlt     d15,d5,0x90000522   ; d15,b,0x90000522
                                  ok  P:9000050C 2ADA                mov16   d15,#0x2A
6        3.      ●      ●         ok  P:9000050E 000A6F7F            jge     d15,d6,0x90000522   ; d15,c,0x90000522
                                  ok  P:90000512 7402                mov16   d4,d7              ; a,d
                                  ok  P:90000514 002B006D            call    0x9000056A         ; Identity
                                  ok  P:90000518 24DA                mov16   d15,#0x24
6        4.      ●      ●         ok  P:9000051A 0004F27F            jge     d2,d15,0x90000522
                                stmt          116          outcome = TRUE;
                                  ok  P:9000051E 1282                mov16   d2,#0x1
                                  ok  P:90000520 023C                j16     0x90000524
                                                       }
```

Figure: This screenshot illustrates the mapping between conditional branches in the object code, tagged as derived from the program flow trace, to the respective source code lines representing a decision, thereby tagging the decision line for MC/DC coverage.

Experience has demonstrated that criteria 2, 3, and 4 are not consistently met in all test scenarios. This results in gaps in code coverage. Lauterbach refers to these gaps as observability gaps.

The Individual Code Coverage Modes

The observability gap refers to a condition in the source code that TRACE32 cannot determine whether it was evaluated as true or false. Consequently, no code coverage result can be displayed for the corresponding decision. Condition, decision and MC/DC coverage becomes incomplete if these gaps are not addressed.

To prevent these gaps, it's helpful to write code in a way that's friendly to code coverage (please refer to **“Appendix F: Coding Guidelines”**, page 134 for details). Moderate optimization also enhances the clarity and intuitiveness of the code coverage analysis for the user.

Lauterbach offers several code coverage modes to address observability gaps, with Targeted Instrumentation being the most commonly used in practice. The choice of mode primarily depends on the number of gaps detected.

- **Code coverage mode *No Instrumentation***

  Selecting this mode assumes there are no observation gaps, allowing the build process to remain unchanged.

- **Code coverage mode *Targeted Instrumentation***

  If there are a moderate number of observability gaps, Lauterbach suggests initially reviewing them before deciding on their necessity for closure. Should you opt to address these gaps, employing the code coverage mode *Targeted Instrumentation* is advisable.

  Employing this code coverage mode can add complexity to your build process. It's good to know that for every observability gap within each function, a corresponding hook function pair is necessary, resulting in increased memory consumption. However, the effect on code size and application runtime remains small.

- **Code coverage mode *Breakpoint Assisted***

  If you aim to address a moderate number of observability gaps without any code instrumentation, you can opt for the *Breakpoint Assisted* code coverage mode. Here, observability gaps are identified prior to code coverage measurement and promptly handled. Breakpoints are strategically placed to stop the program execution, enabling status checks and recording of necessary information. This mode significantly impacts application runtime.

- **Code coverage mode *Full Instrumentation***

  Various factors can contribute to a significant number of gaps: high compiler optimization, unusual core architectures, or core/compiler combinations lacking support. For an exhaustive examination of these potential causes, refer to the chapter **"Causes for Observability Gaps: An Overview"**, page 20.

  When dealing with high compiler optimization levels, consider the following:

  - If maintaining a high optimization level is essential, Lauterbach recommends employing *Full Instrumentation* code coverage mode. This approach introduces numerous instrumentation points, moderately increasing both the program code size and runtime. However, from a technical standpoint, full instrumentation is straightforward, requiring only two hook functions, thereby allowing for further compiler optimizations.

    Full instrumentation, however, necessitates adjustments of the build process. But it offers high robustness and serves as a reliable fallback option.

  - Alternatively, reducing the compiler's optimization level may be considered. Although this increases the program's size and runtime slightly, it should reduce the number of observability gaps to a level where *Targeted Instrumentation* code coverage mode with fewer instrumentation points becomes viable.

  In some cases, using either full or targeted instrumentation modes can result in a similar program size, meaning they have essentially the same impact.

Please keep in mind that adding or modifying source code can create new observability gaps or close existing ones. Therefore, the transition between code coverage mode No Instrumentation and code coverage mode Targeted Instrumentation is particularly fluid.

TRACE32 utilizes body-less hook functions for instrumentation, which are visible in the recorded program flow. They monitor whether an instrumented source code condition has been evaluated as true or false.

TRACE32 instrumentation doesn't need any data memory.

A Comparison of the Different Code Coverage Modes

The following table provides an overview of what has been stated:

	No Instrumentation	**Full Instrumentation**	**Targeted Instrumentation**	**Breakpoint Assisted**
Number of Instrumentation Sites	No	High	Low	No
Instrumentation Technique	—	Two instrumentation hooks	A pair of instrumentation hooks per observability gap within each function	—
Code Size	Unchanged	Moderately larger	Slightly larger	Unchanged
Impact on Runtime	No	Modest	Small	High
Build Process	Unchanged	Simple adaptation	Complex adaptation	Unchanged
Code Coverage Analysis	Based on program flow	Based on program flow	Based on program flow	Based on program flow and status information

Causes for Observability Gaps: An Overview

Lastly, for those interested in wrapping up this chapter, here's an overview of what causes observability gaps.

No dedicated compiler support for the TRACE32 code coverage analysis

The large number of core architectures and the associated diversity of compilers represents a challenge for Lauterbach. An impressive number of cores offer the possibility to generate program flow trace. And there are a big number of compilers, especially for commonly used core architectures. The result is a large amount of possible core architecture/compiler pairings. There is no generic heuristic for mapping source code decisions to conditional branches/instructions at object code level that generates an exact result for every possible pairing. In practice, TRACE32 has to tailor the mapping to the core architecture/compiler combination. Much, especially for common core/compiler combinations is already tailored.

For not yet supported core architecture/compiler pairings, for which the generic heuristic of TRACE32 does not provide an exact result, criterion 3 described on **page 16** is not to be met. This results in observability gaps.

Macros

When a macro used in a decision or condition contains its own decisions or conditions, the compiler expands all macros before compiling, treating the expanded statement as one line of code. This causes the original locations of decisions or conditions within the macro to be lost. As a result, criterion 3 described on **page 16** is not met, and it becomes impossible to map the decisions inside the macro to the conditional branches or instructions. This results in observability gaps.

Highly-optimized code

Highly-optimized code is not recommended for trace-based code coverage. For one, individual conditions may not be represented by conditional branches/instructions at the object code level. Criterion 2 described on **page 16** is violated here. However, this can be remedied. Highly optimized code is particularly challenging because it may not possible to map the decisions/conditions exactly to the conditional branches/instructions. The violation of criterion 3 described on **page 16** cannot be resolved in all cases.

Limitations of the trace protocol

The instruction set for a core architecture may contain conditional instructions. The compiler uses these to implement source code conditions at object code level. For trace-based code coverage to work, the trace protocol used must generate details about the execution of these conditional instructions. Unfortunately, this is not always the case. Currently there is no option that advises the compiler not to use conditional instruction. Observability gaps in program tracing are therefore inevitable. Criterion 4 as described on **page 16** is violated.

If you're uncertain about the properties of your core/trace protocol, the **COVerage.INFO** command can offer clarity.

Instruction set complexity

The issues discussed mostly apply to cores using basic RISC architecture. But in complex Systems-on-Chips (SoCs), there are also special cores and coprocessors, like DSPs or customizable cores with user-defined instructions. These require TRACE32 to be adjusted for their instruction sets. So, it's wise to reach out to Lauterbach for help in such cases.

Evaluation of Switch Case Statements

To evaluate MC/DC, condition and decision for switch case statements, TRACE32 performs an implicit conversion into an equivalent if-then expression. The equivalent if-then expression has the property that in cases where several code paths lead to a single point, all code paths need to be executed at least once before full code coverage is achieved. The following code example illustrates this concept:

```
Switch case statement                 Equivalent if-then expression

switch (color) {                      if (color == RED) {
    case RED:                             offset = 10;
        offset = 10;                  }
        break;                        else if (color == BLUE) {
    case BLUE:                            offset = 8;
        offset = 8;                   }
        break;                        else if (color == ORANGE) {
    case ORANGE:                          offset = 6;
        offset = 6;                   }
        break;                        else if (color == YELLOW) {
    case YELLOW:                          offset = 2;
    case GREEN:                       }
        offset = 2;                   else if (color == GREEN) {
        break;                            offset = 2;
    default:                          }
        offset = -1;                  else {
        break;                            offset = -1;
}                                     }
```

Please note: In contrast to the original switch case statement, the converted if-then expression achieves complete code coverage only when color had both the values YELLOW and GREEN.

Code Coverage Workflows

Workflows for Source Code Metrics

This chapter addresses the code coverage metrics statement coverage, decision coverage, condition coverage, modified condition/decision coverage (MC/DC), and both call and function coverage.

General Procedure

The general procedure involves initially measuring the code coverage in TRACE32 and subsequently evaluating it in a web browser.

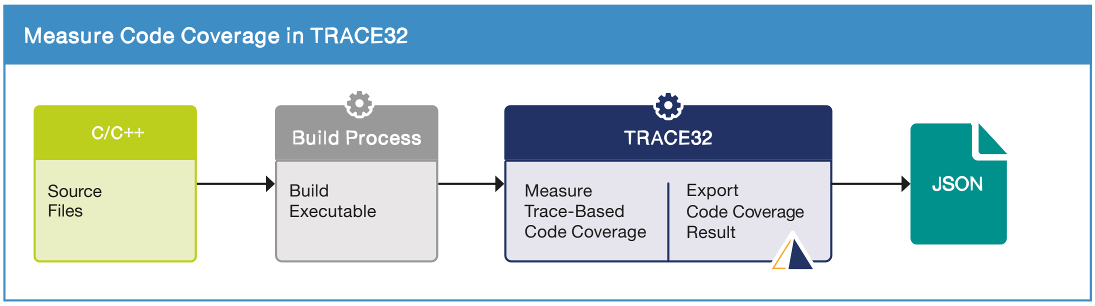

Measure Code Coverage in TRACE32
C/C++
Source Files
Build Process
Build Executable
TRACE32
Measure Trace-Based Code Coverage
Export Code Coverage Result
JSON

Figure: After generating the appropriate executable, code coverage measurement can be conducted in TRACE32. The resulting data must be exported as a JSON file.

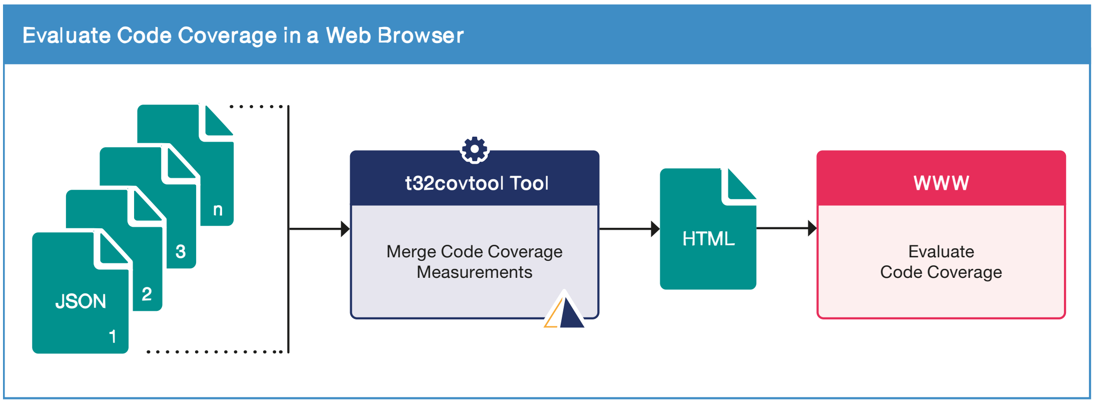

Evaluate Code Coverage in a Web Browser
JSON
1
2
3
n
t32covtool Tool
Merge Code Coverage Measurements
HTML
WWW
Evaluate Code Coverage

Figure: The data resulting from multiple code coverage measurements can be summarized in an HTML file and intuitively evaluated in a web browser.

Statement Coverage Workflow

To perform a statement coverage evaluation, follow these steps.

1. Build the Executable

Ensure to follow the guidelines in **"General Recommendations for the Build Toolchain"**, page 43.

2. Choose a trace data collection variant

Choose the variant that best fits your test scenario:

- Incremental code coverage measurement in Leash mode.

- Incremental code coverage measurement in STREAM mode.

- Continuous code coverage measurement RTS.

- Continuous code coverage measurement SPY.

Refer to **"Trace Data Collection Overview"**, page 60 for assistance in decision-making.

3. **Load necessary files**

Load the files relevant for statement coverage into TRACE32. See **"Preparation for Statement, Function, Object Code, ocb Decision Coverage"**, page 68.

4. Configure and perform code coverage measurement

Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

- General information on trace configuration and recording is in "**"Best Practices for Trace Recording"**, page 64.

- Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **"Trace Data Collection and Code Coverage Measurement"**, page 77.

5. **Export results**

After the code coverage measurement is complete, export the results to a JSON file using the command **COVerage.EXPORT.JSONE**.

6. Generate an HTML report

Generate an HTML report from one or more JSON files as described in **"TRACE32 Merge and Report Tool"**, page 114.

7. **Evaluate the statement coverage intuitively**

Evaluate the statement coverage intuitively using a web browser.

Statement coverage is achieved under the following conditions:

- **Single Object Code Block:** If only one block of object code is generated for a source code line, statement coverage is achieved when at least the first object code statement of this block is executed.

- **Multiple Non-Adjacent Object Code Blocks:** If several non-adjacent blocks of object code are generated for a source code line, statement coverage is achieved when at least the first object code state of each of these blocks is executed.

If you are unfamiliar with the term "Multiple Non-Adjacent Object Code Blocks," we recommend reading **"Debugging of Optimized Code"** in Training Basic Debugging, page 140 (training_debugger.pdf).

TRACE32 uses the following two tags to mark source code lines for statement coverage:

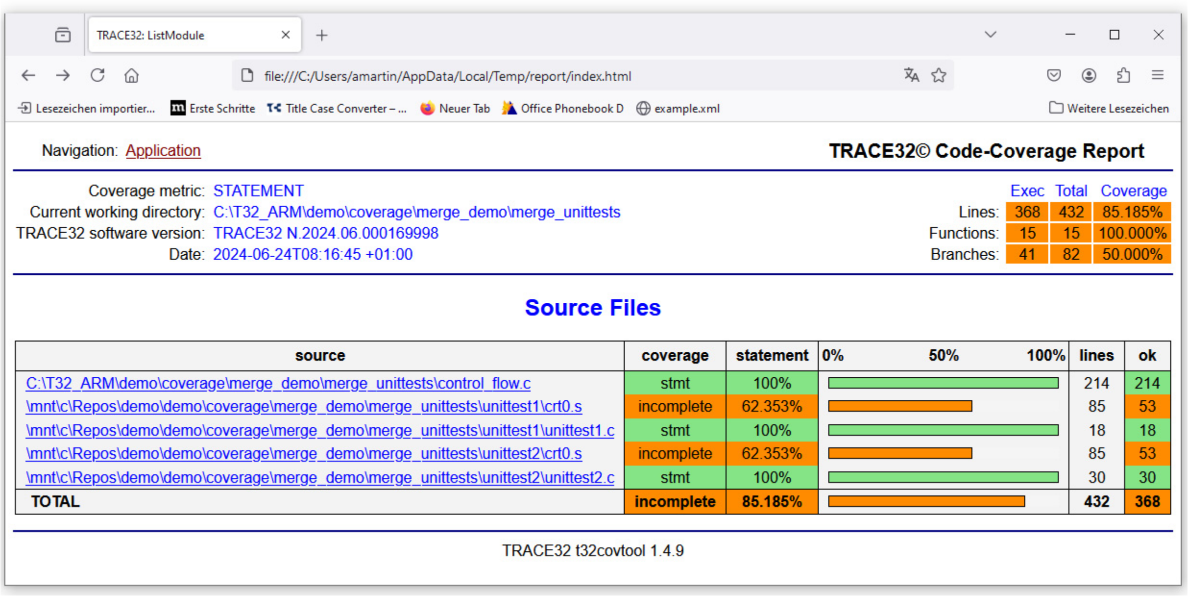
stmt: Statement coverage achieved.

incomplete: Statement coverage not achieved.

At the module and function levels, the tags used are:

stmt: All source code lines of the function/module are tagged stmt.

incomplete: At least one source code line of the function/module is tagged with incomplete.



Navigation: Application

TRACE32© Code-Coverage Report

Coverage metric: STATEMENT
Current working directory: C:\T32_ARM\demo\coverage\merge_demo\merge_unittests
TRACE32 software version: TRACE32 N.2024.06.000169998
Date: 2024-06-24T08:16:45 +01:00

	Exec	Total	Coverage
Lines:	368	432	85.185%
Functions:	15	15	100.000%
Branches:	41	82	50.000%

Source Files

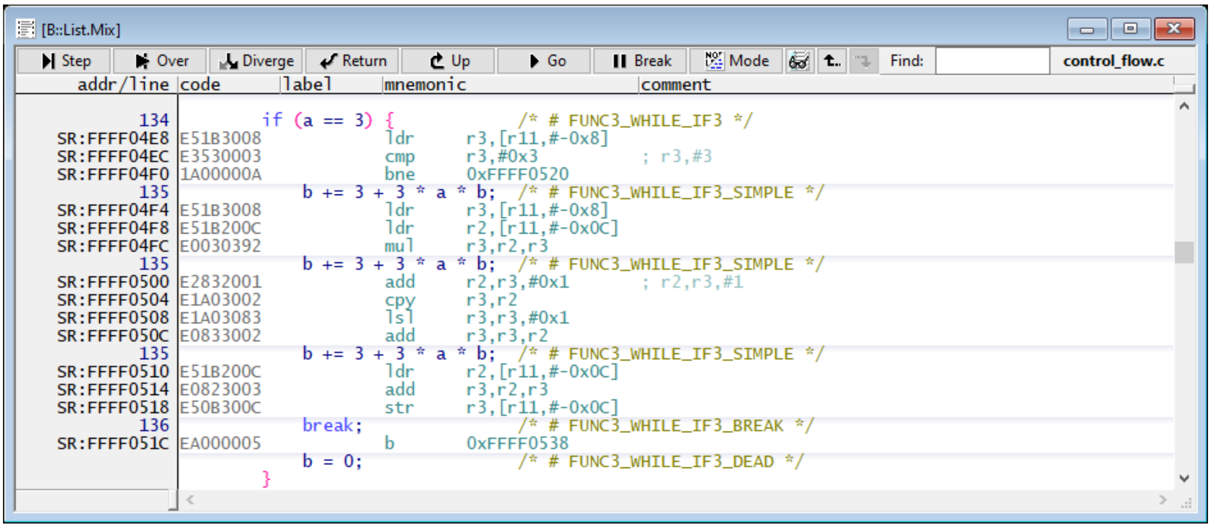
source	coverage	statement	0% 50% 100%	lines	ok
C:\T32_ARM\demo\coverage\merge_demo\merge_unittests\control_flow.c	stmt	100%		214	214
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\crt0.s	incomplete	62.353%		85	53
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\unittest1.c	stmt	100%		18	18
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\crt0.s	incomplete	62.353%		85	53
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\unittest2.c	stmt	100%		30	30
TOTAL	**incomplete**	**85.185%**		**432**	**368**

TRACE32 t32covtool 1.4.9

Figure: Statement coverage evaluation in a web browser.

In rare instances, TRACE32 Trace-Based Code Coverage may not provide precise measurements, especially with short if-blocks. Compiler optimizations can condense these blocks, potentially resulting in false positive statement coverage results. Let's first cover the basics and then go through a few examples.

Debug information, usually loaded with the executable, includes details about which object code corresponds to each source code line (command **sYmbol.List.LINE**). The **List.Mix** window displays this information. Optimizations may cause the compiler to omit object code for certain source code lines. TRACE32 does not display line numbers for these.



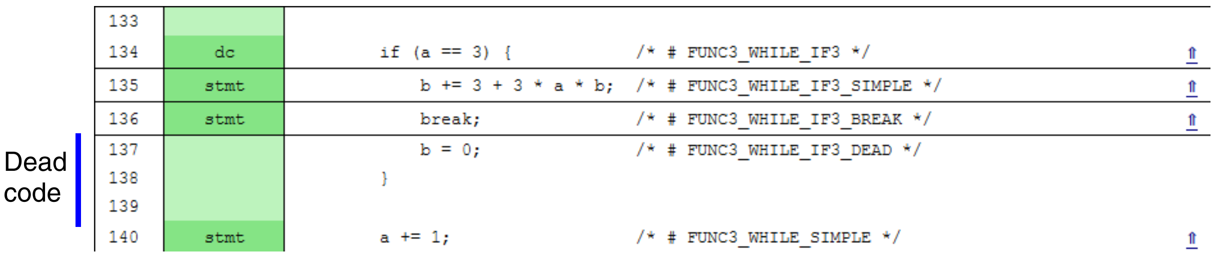
```
     addr/line code      label   mnemonic                comment
          134         if (a == 3) {              /* # FUNC3_WHILE_IF3 */
  SR:FFFF04E8 E51B3008            ldr      r3,[r11,#-0x8]
  SR:FFFF04EC E3530003            cmp      r3,#0x3          ; r3,#3
  SR:FFFF04F0 1A00000A            bne      0xFFFF0520
          135             b += 3 + 3 * a * b;   /* # FUNC3_WHILE_IF3_SIMPLE */
  SR:FFFF04F4 E51B3008            ldr      r3,[r11,#-0x8]
  SR:FFFF04F8 E51B200C            ldr      r2,[r11,#-0x0C]
  SR:FFFF04FC E0030392            mul      r3,r2,r3
          135             b += 3 + 3 * a * b;   /* # FUNC3_WHILE_IF3_SIMPLE */
  SR:FFFF0500 E2832001            add      r2,r3,#0x1       ; r2,r3,#1
  SR:FFFF0504 E1A03002            cpy      r3,r2
  SR:FFFF0508 E1A03083            lsl      r3,r3,#0x1
  SR:FFFF050C E0833002            add      r3,r3,r2
          135             b += 3 + 3 * a * b;   /* # FUNC3_WHILE_IF3_SIMPLE */
  SR:FFFF0510 E51B200C            ldr      r2,[r11,#-0x0C]
  SR:FFFF0514 E0823003            add      r3,r2,r3
  SR:FFFF0518 E50B300C            str      r3,[r11,#-0x0C]
          136             break;                /* # FUNC3_WHILE_IF3_BREAK */
  SR:FFFF051C EA000005            b        0xFFFF0538
                          b = 0;                /* # FUNC3_WHILE_IF3_DEAD */
                      }
```

Figure: In TRACE32, the source code statement `b = 0;` does not have line numbers.

TRACE32's code coverage analysis relies on the object code, as only the object code is recorded in the program flow trace. Source code lines are tagged for statement coverage through an appropriate mapping between the object code and the source code. However, TRACE32 ignores source code lines without line numbers/corresponding object code when performing statement coverage. Consequently, some statements are not invoked but are not explicitly tagged as incomplete in the TRACE32 statement coverage evaluation. Here are some illustrative examples.

Dead Code

As part of compiler optimizations, dead code elimination leads to no object code being generated for dead code at source code level. Since TRACE32 ignores source code lines without object code during statement coverage, it is advisable to review the code coverage report to identify any dead code. In the TRACE32 Code-Coverage Report, these source code lines are displayed next to the following line that has object code and are shown in a lighter color.



```
133
134   dc      if (a == 3) {            /* # FUNC3_WHILE_IF3 */
135   stmt        b += 3 + 3 * a * b;  /* # FUNC3_WHILE_IF3_SIMPLE */
136   stmt        break;               /* # FUNC3_WHILE_IF3_BREAK */
137               b = 0;               /* # FUNC3_WHILE_IF3_DEAD */
138           }
139
140   stmt    a += 1;                  /* # FUNC3_WHILE_SIMPLE */
```

Figure: In the TRACE32 Code-Coverage Report the statements `b = 0;` is displayed along with the next statement. It is shown in a lighter color.

To achieve complete statement coverage, these lines of source code must be removed.

Short if-Block (conditional branch)

Here is a small source code example where the compiler generates object code only for the statement `if (a == 5)`, but not for the `break;` statement. And the object code generated for the `if` statement includes a conditional branch.

```
if (a  == 5)
 CMP R3, #5
 BNE func_end
    break;

b = a+c
…
RETURN b;
```

TRACE32 interprets statement coverage as: "A source code line achieves statement coverage when at least the first object code statement generated for this line has been executed." Based on this, the `if` statement would achieve statement coverage as soon as the `CMP` instruction is executed, regardless of whether `a == 5` is true or not. This interpretation is incorrect.

For precise statement coverage, it is essential to verify that `a == 5` was evaluated both true and false. To achieve this, you need to inspect the object code coverage for the conditional branch `BNE` in case of this type of compiler optimization. As long as the conditional branch is only tagged with "taken" or '"not taken" statement coverage has not been achieved.

Short if-Block (conditional instruction)

Here is a small source code example where the compiler generates object code only for the statement `if (a == 5)`, but not for the `b = 7;` statement. And the object code generated for the `if` statement includes a conditional instruction.

```
if (a == 5)
  CMP   R3, #5
   MOVEQ R4, #7
     b = 7;
…
```

TRACE32 interprets statement coverage as: "A source code line achieves statement coverage when at least the first object code statement generated for this line has been executed." Based on this, the `if` statement would achieve statement coverage as soon as the `CMP` instruction is executed, regardless of whether `a == 5` is true or not. This interpretation is incorrect.

For precise statement coverage, it is essential to verify that `a == 5` was evaluated both true and false. To achieve this, you need to inspect the object code coverage for the conditional instruction `MOVEQ` in case of this type of compiler optimization. However, this is only possible if the trace protocol of the core under debug supports conditional instructions. You can use the **COVerage.INFO** command or the **CPU.Feature**(CONDTRACE) function to check this.

- If the trace protocol does not support conditional instructions, statement coverage cannot be verified for this type of compiler optimization.
- If the trace protocol supports conditional instructions indicating whether the condition code check passed or failed, you need to inspect the object code coverage. As long as the conditional instruction is only tagged with "only exec" or "not exec," statement coverage has not been achieved.

Condition Coverage Workflow

Before starting the evaluation for condition coverage, it is recommended to review chapter **“MC/DC, Condition and Decision Coverage”**, page 16.

To perform a condition coverage evaluation, follow these steps.

1. **Build the executable**

   When performing condition coverage analysis, it's possible to encounter observability gaps. TRACE32 offers various code coverage modes to address these, outlined in chapter **“The Individual Code Coverage Modes”**, page 17. Your choice of mode will depend on your application specifics. Refer to **“Decision Making Process”**, page 47 for guidance in selecting the appropriate mode.

   Generate all files needed for condition coverage, as detailed in chapter **“Build Process MC/DC, Condition and Decision Coverage”**, page 47. Each code coverage mode has a dedicated sub-chapter there. Ensure adherence to the guidelines provided in **“Build Process Call Coverage”**, page 46.

2. **Choose a trace data collection variant**

   Choose the variant that best fits your test scenario:

   - Incremental code coverage measurement in Leash mode.

   - Incremental code coverage measurement in STREAM mode.

   - Continuous code coverage measurement RTS.

   - Continuous code coverage measurement SPY.

   Refer to **“Trace Data Collection Overview”**, page 60 for assistance in decision-making.

3. **Load relevant files into TRACE32**

   Load the files relevant for the condition coverage into TRACE32, see **“Preparation for MC/DC, Condition and Decision Coverage”**, page 70. Read the sub-chapter on the code coverage mode that you decided to use in step 1.

4. **Configure and perform code coverage measurement**

   Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

   - General information on trace configuration and recording is in "**“Best Practices for Trace Recording”**, page 64.

   - Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **“Trace Data Collection and Code Coverage Measurement”**, page 77.

5. **Export results**

   After the code coverage measurement is complete, export the results to a JSON file using the command **COVerage.EXPORT.JSONE**.

6. **Generate an HTML report**

   Generate an HTML report from one or more JSON files as described in **“TRACE32 Merge and Report Tool”**, page 114

7. Evaluate the condition coverage intuitively

Evaluate the condition coverage intuitively via a web browser. TRACE32 uses the following two tags to mark source code condition statements for condition coverage:

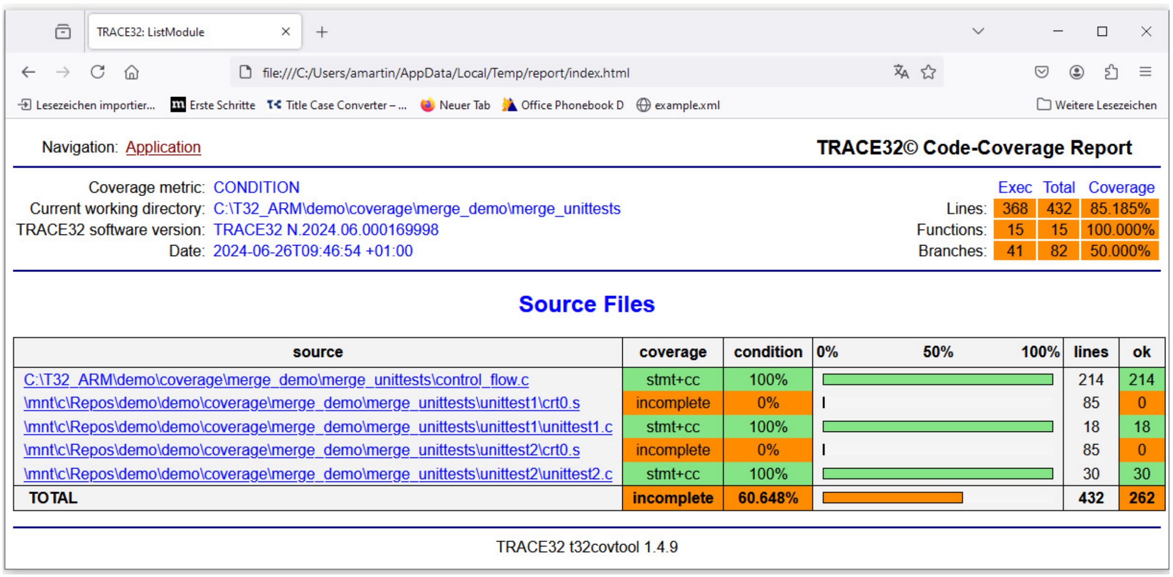
cc: Condition coverage achieved — both true and false evaluations for all conditions in the source code statement have been achieved.

incomplete: Condition coverage not achieved — at least one condition in the source code statement has not been evaluated for both true and false.

At the module and function levels, the tags used are:

stmt+cc: All source code lines of the function/module are tagged either with cc or stmt (statement coverage achieved).

incomplete: At least one source code line of the function/module is tagged with incomplete.



Navigation: Application

TRACE32© Code-Coverage Report

Coverage metric:	CONDITION
Current working directory:	C:\T32_ARM\demo\coverage\merge_demo\merge_unittests
TRACE32 software version:	TRACE32 N.2024.06.000169998
Date:	2024-06-26T09:46:54 +01:00

	Exec	Total	Coverage
Lines:	368	432	85.185%
Functions:	15	15	100.000%
Branches:	41	82	50.000%

Source Files

source	coverage	condition	0% 50% 100%	lines	ok
C:\T32_ARM\demo\coverage\merge_demo\merge_unittests\control_flow.c	stmt+cc	100%		214	214
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\crt0.s	incomplete	0%		85	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\unittest1.c	stmt+cc	100%		18	18
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\crt0.s	incomplete	0%		85	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\unittest2.c	stmt+cc	100%		30	30
TOTAL	**incomplete**	**60.648%**		**432**	**262**

TRACE32 t32covtool 1.4.9

Figure: Condition coverage evaluation in a web browser.

Decision Coverage Workflow

Before starting the evaluation for decision coverage, it is recommended to review chapter **“MC/DC, Condition and Decision Coverage”**, page 16.

To perform a decision coverage evaluation, follow these steps.

1. **Build the executable**

   When performing decision coverage analysis, it's possible to encounter observability gaps. TRACE32 offers various code coverage modes to address these, outlined in chapter **“The Individual Code Coverage Modes”**, page 17. Your choice of mode will depend on your application specifics. Refer to **“Decision Making Process”**, page 47 for guidance in selecting the appropriate mode.

   Generate all files needed for decision coverage, as detailed in chapter **“Build Process MC/DC, Condition and Decision Coverage”**, page 47. Each code coverage mode has a dedicated sub-chapter there. Ensure adherence to the guidelines provided in **“Build Process Call Coverage”**, page 46.

2. **Choose a trace data collection variant**

   Choose the variant that best fits your test scenario:

   - Incremental code coverage measurement in Leash mode.

   - Incremental code coverage measurement in STREAM mode.

   - Continuous code coverage measurement RTS.

   - Continuous code coverage measurement SPY.

   Refer to **“Trace Data Collection Overview”**, page 60 for assistance in decision-making.

3. **Load relevant files into TRACE32**

   Load the files relevant for the decision coverage into TRACE32, see **“Preparation for MC/DC, Condition and Decision Coverage”**, page 70. Read the sub-chapter on the code coverage mode that you decided to use in step 1.

4. **Configure and perform code coverage measurement**

   Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

   - General information on trace configuration and recording is in "**“Best Practices for Trace Recording”**, page 64.

   - Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **“Trace Data Collection and Code Coverage Measurement”**, page 77.

5. **Export results**

   After the code coverage measurement is complete, export the results to a JSON file using the command **COVerage.EXPORT.JSONE**.

6. **Generate an HTML report**

   Generate an HTML report from one or more JSON files as described in **“TRACE32 Merge and Report Tool”**, page 114.

7. Evaluate the decision coverage intuitively

Evaluate the decision coverage intuitively via a web browser. TRACE32 uses the following two tags to mark source code decision statements for decision coverage:

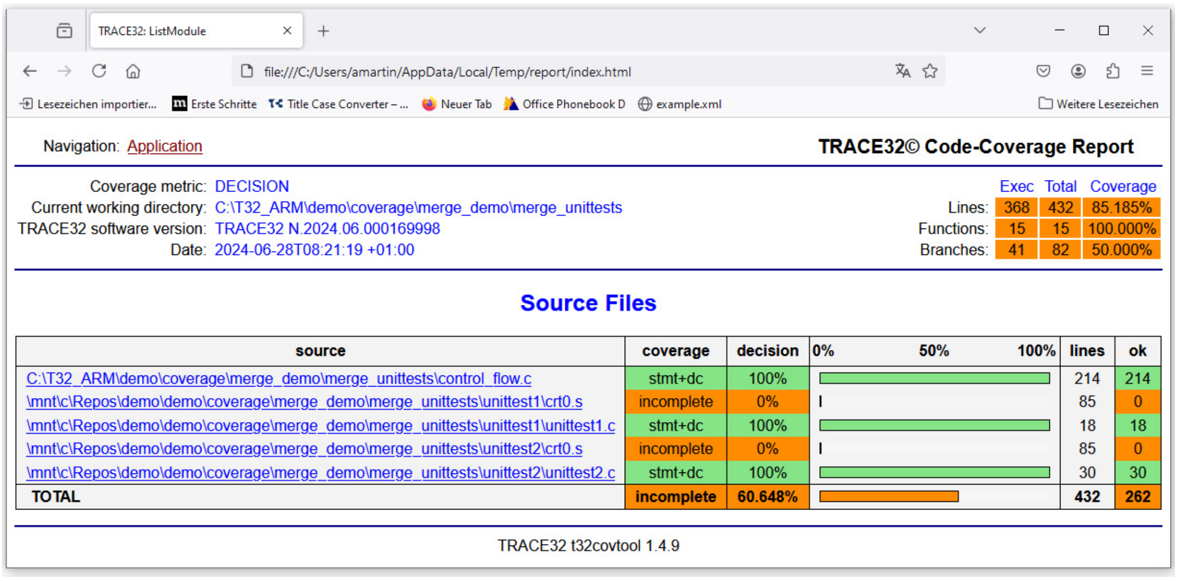
dc: Decision coverage achieved — the decision in the source code statement have taken all possible outcomes at least once.

incomplete: Decision coverage not achieved — at least one possible outcome is missing for the decision.

At the module and function levels, the tags used are:

stmt+dc: All source code lines of the function/module are tagged either with dc or stmt (statement coverage achieved).

incomplete: At least one source code line of the function/module is tagged with incomplete.



Navigation: Application

TRACE32© Code-Coverage Report

Coverage metric: DECISION
Current working directory: C:\T32_ARM\demo\coverage\merge_demo\merge_unittests
TRACE32 software version: TRACE32 N.2024.06.000169998
Date: 2024-06-28T08:21:19 +01:00

	Exec	Total	Coverage
Lines:	368	432	85.185%
Functions:	15	15	100.000%
Branches:	41	82	50.000%

Source Files

source	coverage	decision	0% 50% 100%	lines	ok
C:\T32_ARM\demo\coverage\merge_demo\merge_unittests\control_flow.c	stmt+dc	100%		214	214
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\crt0.s	incomplete	0%		85	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\unittest1.c	stmt+dc	100%		18	18
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\crt0.s	incomplete	0%		85	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\unittest2.c	stmt+dc	100%		30	30
TOTAL	**incomplete**	**60.648%**		**432**	**262**

TRACE32 t32covtool 1.4.9

Figure: Decision coverage evaluation in a web browser.

MC/DC Workflow

Before starting the evaluation for Modified Condition/Decision Coverage, it is recommended to review chapter **“MC/DC, Condition and Decision Coverage”**, page 16.

To perform an MC/DC, follow these steps.

1. **Build the executable**

   When performing MC/DC analysis, it's possible to encounter observability gaps. TRACE32 offers various code coverage modes to address these, outlined in chapter **“The Individual Code Coverage Modes”**, page 17. Your choice of mode will depend on your application specifics. Refer to **“Decision Making Process”**, page 47 for guidance in selecting the appropriate mode.

   Generate all files needed for MC/DC, as detailed in chapter **“Build Process MC/DC, Condition and Decision Coverage”**, page 47. Each code coverage mode has a dedicated sub-chapter there. Ensure adherence to the guidelines provided in **“Build Process Call Coverage”**, page 46.

2. **Choose a trace data collection variant**

   Choose the variant that best fits your test scenario:

   - Incremental code coverage measurement in Leash mode.

   - Incremental code coverage measurement in STREAM mode.

   - Continuous code coverage measurement RTS.

   - Continuous code coverage measurement SPY.

   Refer to **“Trace Data Collection Overview”**, page 60 for assistance in decision-making.

3. **Load relevant files into TRACE32**

   Load the files relevant for MC/DC into TRACE32, see **“Preparation for MC/DC, Condition and Decision Coverage”**, page 70. Read the sub-chapter on the code coverage mode that you decided to use in step 1.

4. **Configure and perform code coverage measurement**

   Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

   - General information on trace configuration and recording is in "**“Best Practices for Trace Recording”**, page 64.

   - Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **“Trace Data Collection and Code Coverage Measurement”**, page 77.

5. **Export results**

   After the code coverage measurement is complete, export the results to a JSON file using the command **COVerage.EXPORT.JSONE**.

6. **Generate an HTML report**

   Generate an HTML report from one or more JSON files as described in **“TRACE32 Merge and Report Tool”**, page 114

7. Evaluate MC/DC intuitively

Evaluate MC/DC intuitively via a web browser. TRACE32 uses the following two tags to mark source code decision statements for MC/DC:

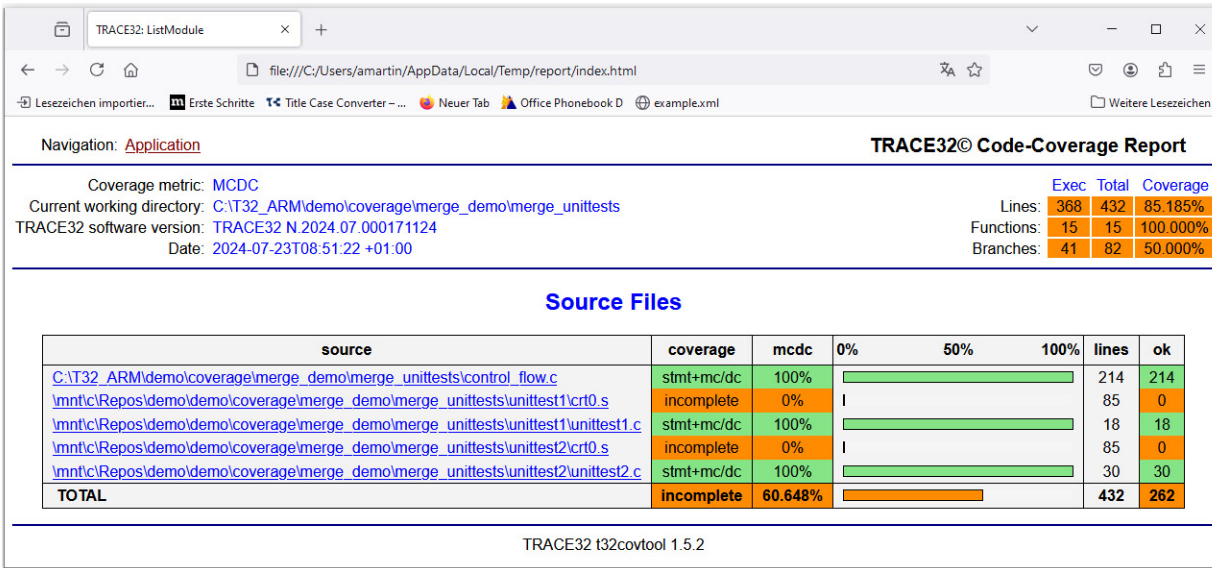
mc/dc: MC/DC achieved — Each condition in the decision is shown to independently affect the outcome of the decision.

incomplete: MC/DC not achieved — at least one condition in the decision has not yet been shown to independently affect the outcome.

At the module and function levels, the tags used are:

stmt+mc/dc: All source code lines of the function/module are tagged either with mc/dc or stmt (statement coverage achieved).

incomplete: At least one source code line of the function/module is tagged with incomplete.



Navigation: Application

TRACE32© Code-Coverage Report

Coverage metric: MCDC
Current working directory: C:\T32_ARM\demo\coverage\merge_demo\merge_unittests
TRACE32 software version: TRACE32 N.2024.07.000171124
Date: 2024-07-23T08:51:22 +01:00

	Exec	Total	Coverage
Lines:	368	432	85.185%
Functions:	15	15	100.000%
Branches:	41	82	50.000%

Source Files

source	coverage	mcdc	0% 50% 100%	lines	ok
C:\T32_ARM\demo\coverage\merge_demo\merge_unittests\control_flow.c	stmt+mc/dc	100%		214	214
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\crt0.s	incomplete	0%		85	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\unittest1.c	stmt+mc/dc	100%		18	18
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\crt0.s	incomplete	0%		85	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\unittest2.c	stmt+mc/dc	100%		30	30
TOTAL	**incomplete**	**60.648%**		**432**	**262**

TRACE32 t32covtool 1.5.2

Figure: MC/DC evaluation in a web browser.

Function Coverage Workflow

To perform function coverage evaluation, follow these steps.

1. **Build the executable**

   Create your executable file, ensuring that function inlining is disabled for clearer and more intuitive results. Be sure to follow the guidelines provided in **“Build Process Call Coverage”**, page 46.

2. **Choose a trace data collection variant**

   Choose the variant that best fits your test scenario:

   - Incremental code coverage measurement in Leash mode.

   - Incremental code coverage measurement in STREAM mode.

   - Continuous code coverage measurement RTS.

   - Continuous code coverage measurement SPY.

   Refer to **“Trace Data Collection Overview”**, page 60 for assistance in decision-making.

3. **Load necessary files**

   Load the files relevant for function coverage into TRACE32. See **“Preparation for Statement, Function, Object Code, ocb Decision Coverage”**, page 68.

4. **Configure and perform code coverage measurement**

   Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

   - General information on trace configuration and recording is in "**“Best Practices for Trace Recording”**, page 64.

   - Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **“Trace Data Collection and Code Coverage Measurement”**, page 77.

5. **Export results**

   After the code coverage measurement is complete, export the results to a JSON file using the command **COVerage.EXPORT.JSONE**.

6. **Generate an HTML report**

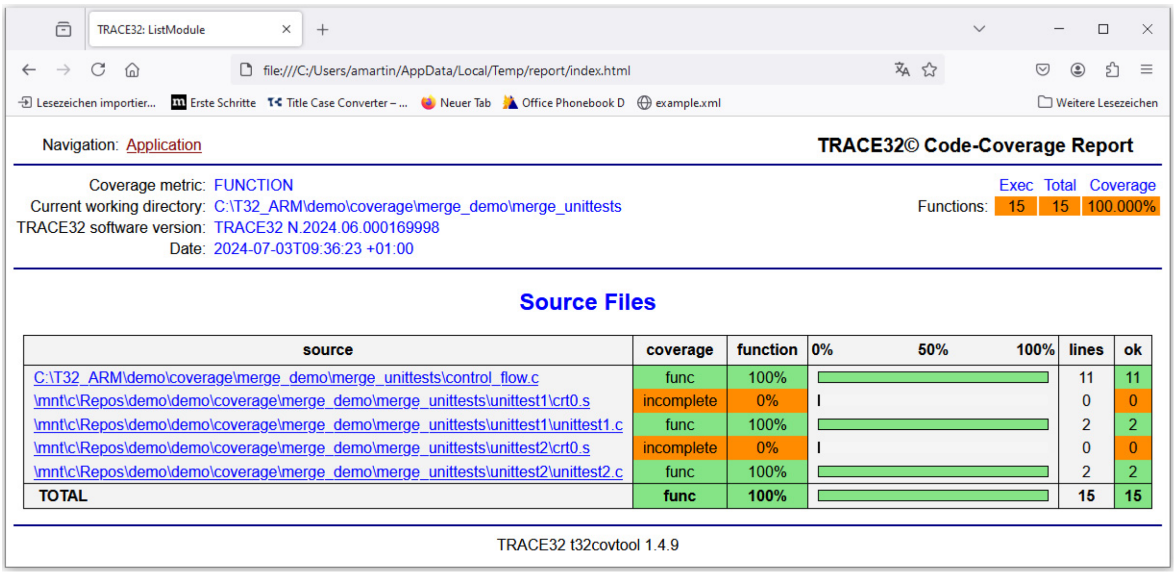
   Generate an HTML report from one or more JSON files as described in **“TRACE32 Merge and Report Tool”**, page 114.

7. **Evaluate the function coverage intuitively**

   Evaluate the function coverage intuitively via a web browser. TRACE32 uses the following two tags to mark the functions for function coverage:

   func: Function coverage achieved — at least one function's object code instructions has been executed.

   incomplete: Function coverage not achieved — none of the function's object code instructions has been executed.


TRACE32: ListModule
file:///C:/Users/amartin/AppData/Local/Temp/report/index.html
Lesezeichen importier... Erste Schritte Title Case Converter – ... Neuer Tab Office Phonebook D example.xml
Weitere Lesezeichen

Navigation: Application

TRACE32© Code-Coverage Report

Coverage metric: FUNCTION
Current working directory: C:\T32_ARM\demo\coverage\merge_demo\merge_unittests
TRACE32 software version: TRACE32 N.2024.06.000169998
Date: 2024-07-03T09:36:23 +01:00

	Exec	Total	Coverage
Functions:	15	15	100.000%

Source Files

source	coverage	function	0% 50% 100%	lines	ok
C:\T32_ARM\demo\coverage\merge_demo\merge_unittests\control_flow.c	func	100%		11	11
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\crt0.s	incomplete	0%		0	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\unittest1.c	func	100%		2	2
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\crt0.s	incomplete	0%		0	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\unittest2.c	func	100%		2	2
TOTAL	**func**	**100%**		**15**	**15**

TRACE32 t32covtool 1.4.9

Figure: Function coverage evaluation in a web browser.

To perform call coverage evaluation, follow these steps.

1. **Build the executable**

   Generate all files needed for call coverage, as detailed in chapter **"Build Process Call Coverage"**, page 46. Ensure adherence to the guidelines provided in **"Build Process Call Coverage"**, page 46.

2. **Choose a trace data collection variant**

   Choose the variant that best fits your test scenario:

   - Incremental code coverage measurement in Leash mode.

   - Incremental code coverage measurement in STREAM mode.

   - Continuous code coverage measurement RTS.

   - Continuous code coverage measurement SPY.

   Refer to **"Trace Data Collection Overview"**, page 60 for assistance in decision-making.

3. **Load necessary files**

   Load the files relevant for call coverage into TRACE32. See **"Preparation for Call Coverage"**, page 69.

4. **Configure and perform code coverage measurement**

   Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

   - General information on trace configuration and recording is in "**"Best Practices for Trace Recording"**, page 64.

   - Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **"Trace Data Collection and Code Coverage Measurement"**, page 77.

5. **Export results**

   After the code coverage measurement is complete, export the results to a JSON file using the command **COVerage.EXPORT.JSONE**.

6. **Generate an HTML report**

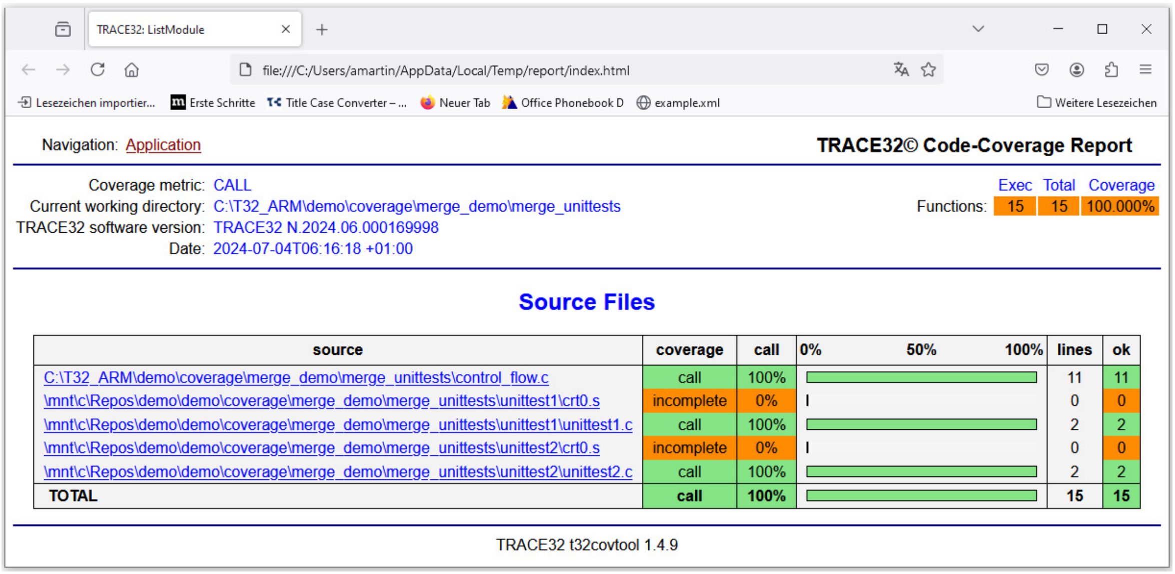
   Generate an HTML report from one or more JSON files as described in **"TRACE32 Merge and Report Tool"**, page 114.

7. Evaluate the call coverage intuitively

Evaluate the call coverage intuitively via a web browser. TRACE32 uses the following two tags to mark the functions for call coverage:

call: Call coverage achieved — all unconditional branches that represent a function call have been executed at least once. If a function does not include an unconditional branch that represent a function call, the function is tagged with call if at least one corresponding object code instruction generated for the function has been executed.

incomplete: Call coverage not achieved — at least one unconditional branch that represent a function call has not been executed. Or no object code instruction generated for the function has been executed for all call-less functions.



Navigation: Application

TRACE32© Code-Coverage Report

Coverage metric: CALL
Current working directory: C:\T32_ARM\demo\coverage\merge_demo\merge_unittests
TRACE32 software version: TRACE32 N.2024.06.000169998
Date: 2024-07-04T06:16:18 +01:00

	Exec	Total	Coverage
Functions:	15	15	100.000%

Source Files

source	coverage	call	0% 50% 100%	lines	ok
C:\T32_ARM\demo\coverage\merge_demo\merge_unittests\control_flow.c	call	100%	🟩	11	11
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\crt0.s	incomplete	0%	I	0	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest1\unittest1.c	call	100%	🟩	2	2
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\crt0.s	incomplete	0%	I	0	0
\mnt\c\Repos\demo\demo\coverage\merge_demo\merge_unittests\unittest2\unittest2.c	call	100%	🟩	2	2
TOTAL	**call**	**100%**	🟩	**15**	**15**

TRACE32 t32covtool 1.4.9

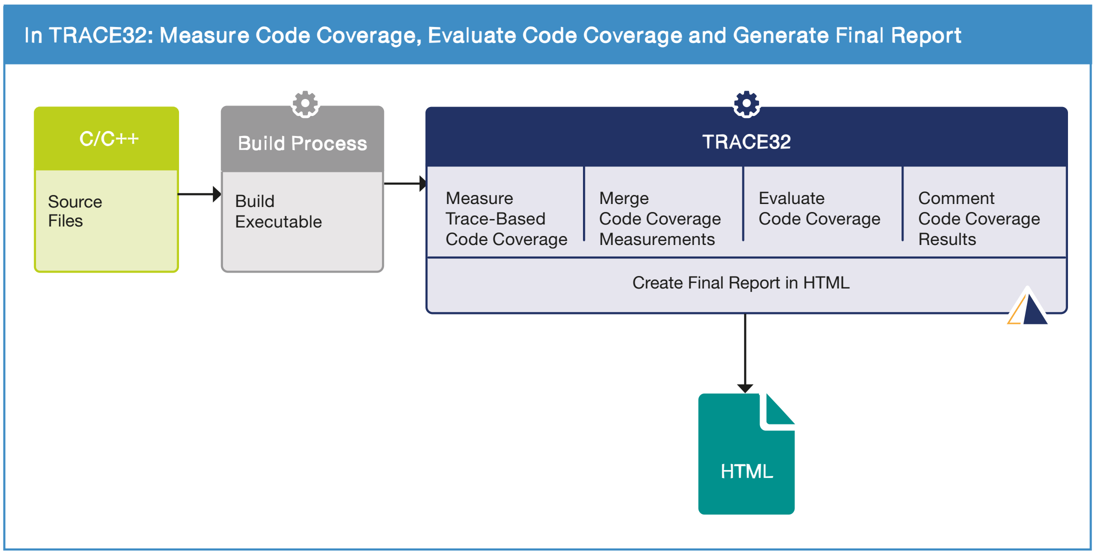
Figure: Call coverage evaluation in a web browser.

Workflows for Address-Based Metrics

This chapter addresses object code coverage and object code based (ocb) decision coverage.

General Procedure

For the address-based code coverage metrics, all measurement and evaluation steps were conducted in TRACE32. Finally, an HTML report can be generated for documentation purposes.



In TRACE32: Measure Code Coverage, Evaluate Code Coverage and Generate Final Report
C/C++
Source Files
Build Process
Build Executable
TRACE32
Measure Trace-Based Code Coverage
Merge Code Coverage Measurements
Evaluate Code Coverage
Comment Code Coverage Results
Create Final Report in HTML
HTML

Object Code Coverage Workflow

To perform a object code coverage evaluation, follow these steps.

1. **Build the executable**

   Ensure to follow the guidelines in **"Build Process Call Coverage"**, page 46.

2. **Choose a trace data collection variant**

   Select the variant that best fits your test scenario:

   - Incremental code coverage measurement in Leash mode.

   - Incremental code coverage measurement in STREAM mode.

   - Continuous code coverage measurement RTS.

   - Continuous code coverage measurement SPY.

   Refer to **"Trace Data Collection Overview"**, page 60 for assistance in decision-making.

3. **Load necessary files**

   Load the files relevant for object code coverage into TRACE32. See **"Preparation for Statement, Function, Object Code, ocb Decision Coverage"**, page 68.

4. **Configure and perform code coverage measurement**

   Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

   - General information on trace configuration and recording is in **"Best Practices for Trace Recording"**, page 64.

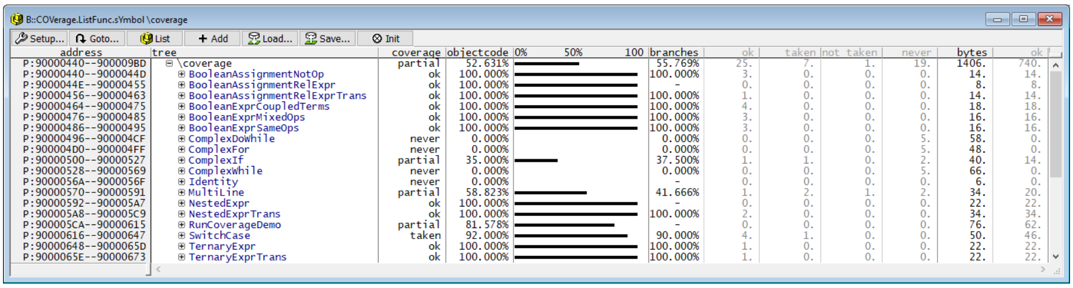
   - Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **"Trace Data Collection and Code Coverage Measurement"**, page 77.

5. **Assemble the results of various code coverage measurements**

   Ensure you only assemble test runs carried out with the identical executable(s). Instructions for this process can be found in **"Appendix B: Assemble Multiple Test Runs at Address Level"**, page 119.

6. **Evaluate object code coverage.**

   Evaluation details can be found at **"Object Code Coverage Evaluation in TRACE32"**, page 101.



```
address                  tree                                  coverage  objectcode  branches        ok   taken  not taken  never   bytes     ok
P:90000440--900009BD     ⊟ \coverage                            partial     52.631%    55.769%       25.     7.        1.    19.   1406.   740.
P:90000440--9000044D       ⊞ BooleanAssignmentNotOp                  ok    100.000%   100.000%        3.     0.        0.     0.     14.    14.
P:9000044E--90000455       ⊞ BooleanAssignmentRelExpr                ok    100.000%          -        0.     0.        0.     0.      8.     8.
P:90000456--90000463       ⊞ BooleanAssignmentRelExprTrans           ok    100.000%   100.000%        1.     0.        0.     0.     14.    14.
P:90000464--90000475       ⊞ BooleanExprCoupledTerms                 ok    100.000%   100.000%        4.     0.        0.     0.     18.    18.
P:90000476--90000485       ⊞ BooleanExprMixedOps                     ok    100.000%   100.000%        3.     0.        0.     0.     16.    16.
P:90000486--90000495       ⊞ BooleanExprSameOps                      ok    100.000%   100.000%        3.     0.        0.     0.     16.    16.
P:90000496--900004CF       ⊞ ComplexDoWhile                       never      0.000%     0.000%        0.     0.        0.     5.     58.     0.
P:900004D0--900004FF       ⊞ ComplexFor                           never      0.000%     0.000%        0.     0.        0.     5.     48.     0.
P:90000500--90000527       ⊞ ComplexIf                          partial     35.000%    37.500%        1.     1.        0.     2.     40.    14.
P:90000528--90000569       ⊞ ComplexWhile                         never      0.000%     0.000%        0.     0.        0.     5.     66.     0.
P:9000056A--9000056F       ⊞ Identity                             never      0.000%          -        0.     0.        0.     0.      6.     0.
P:90000570--90000591       ⊞ MultiLine                          partial     58.823%    41.666%        1.     2.        1.     2.     34.    20.
P:90000592--900005A7       ⊞ NestedExpr                              ok    100.000%          -        0.     0.        0.     0.     22.    22.
P:900005A8--900005C9       ⊞ NestedExprTrans                         ok    100.000%   100.000%        2.     0.        0.     0.     34.    34.
P:900005CA--90000615       ⊞ RunCoverageDemo                    partial     81.578%          -        0.     0.        0.     0.     76.    62.
P:90000616--90000647       ⊞ SwitchCase                           taken     92.000%    90.000%        4.     1.        0.     0.     50.    46.
P:90000648--9000065D       ⊞ TernaryExpr                             ok    100.000%   100.000%        1.     0.        0.     0.     22.    22.
P:9000065E--90000673       ⊞ TernaryExprTrans                        ok    100.000%   100.000%        1.     0.        0.     0.     22.    22.
```

Figure: Object code coverage evaluation in TRACE32.

7. **Comment uncovered code**

   Add comments to the uncovered code ranges, see **“Comment Your Results”**, page 112.

8. **Generate final HTML report**

   Generate your final code coverage report as described **“Appendix A: TRACE32 Coverage Report Utility”**, page 117.

Object Code Based (ocb) Decision Coverage Workflow

The code coverage metric ocb decision coverage is old fashioned and no longer really needed. However, it can be helpful for special problems. If such a situation arises, our support team will inform you.

To perform a ocb decision coverage evaluation, follow these steps.

1. **Build the executable**

   Ensure to follow the guidelines in **"Build Process Call Coverage"**, page 46.

   It is recommended to disable most if not all optimizations to avoid false-positive or false-negative results. Please also check **"Appendix F: Coding Guidelines"**, page 134.

2. **Choose a trace data collection variant**

   Select the variant that best fits your test scenario:

   - Incremental code coverage measurement in Leash mode.

   - Incremental code coverage measurement in STREAM mode.

   - Continuous code coverage measurement RTS.

   - Continuous code coverage measurement SPY.

   Refer to **"Trace Data Collection Overview"**, page 60 for assistance in decision-making.

3. **Load necessary files**

   Load the files relevant for object code coverage into TRACE32. See **"Preparation for Statement, Function, Object Code, ocb Decision Coverage"**, page 68.

4. **Configure and perform code coverage measurement**

   Configure the trace for the trace data collection variant chosen in step 2 and perform the code coverage measurement.

   - General information on trace configuration and recording is in **"Best Practices for Trace Recording"**, page 64.

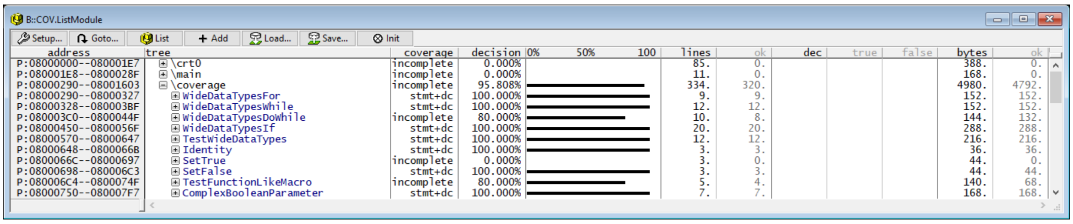
   - Specific guidelines for individual trace data collection variants and the actual code coverage measurement steps can be found in **"Trace Data Collection and Code Coverage Measurement"**, page 77.

5. **Assemble the results of various code coverage measurements**

   Ensure you only assemble test runs carried out with the identical executable(s). Instructions for this process can be found in **"Appendix B: Assemble Multiple Test Runs at Address Level"**, page 119.

6. Evaluate object code coverage

Evaluation details can be found at **“Object Code Based (ocb) Decision Coverage Evaluation”**, page 106.



address	tree	coverage	decision	0% 50% 100	lines	ok	dec	true	false	bytes	ok
P:08000000--080001E7	\crt0	incomplete	0.000%		85.	0.				388.	0.
P:080001E8--0800028F	\main	incomplete	0.000%		11.	0.				168.	0.
P:08000290--08001603	\coverage	incomplete	95.808%		334.	320.				4980.	4792.
P:08000290--08000327	WideDataTypesFor	stmt+dc	100.000%		9.	9.				152.	152.
P:08000328--080003BF	WideDataTypesWhile	stmt+dc	100.000%		12.	12.				152.	152.
P:080003C0--0800044F	WideDataTypesDoWhile	incomplete	80.000%		10.	8.				144.	132.
P:08000450--0800056F	WideDataTypesIf	stmt+dc	100.000%		20.	20.				288.	288.
P:08000570--08000647	TestWideDataTypes	stmt+dc	100.000%		12.	12.				216.	216.
P:08000648--0800066B	Identity	stmt+dc	100.000%		3.	3.				36.	36.
P:0800066C--08000697	SetTrue	incomplete	0.000%		3.	0.				44.	0.
P:08000698--080006C3	SetFalse	stmt+dc	100.000%		3.	3.				44.	44.
P:080006C4--0800074F	TestFunctionLikeMacro	incomplete	80.000%		5.	4.				140.	68.
P:08000750--080007F7	ComplexBooleanParameter	stmt+dc	100.000%		7.	7.				168.	168.

Figure: ocb decision coverage evaluation in TRACE32.

7. Comment uncovered code

Add comments to the uncovered code ranges, see **“Comment Your Results”**, page 112.

8. Generate final HTML report

Generate your final code coverage report as described **“Appendix A: TRACE32 Coverage Report Utility”**, page 117.

Build Process

Introductory Notes

General Recommendations for the Build Toolchain

The recommendations outlined here apply to all code coverage metrics. TRACE32 code coverage performs optimally at low compiler optimization levels, enhancing the mapping between object and source code. Code coverage analysis relies on object code captured as program flow trace, and accurate mapping is more effective with lower optimization levels. Consequently, TRACE32's trace-based code coverage cannot be conducted on production code.

NOTE:	It is recommended to configure the toolchain so that code optimizations are disabled and no jump tables are used. The following list shows recommended compiler configurations for selected toolchains: • GNU Compiler Collection (GCC) or Clang: **`-O0 -fno-jump-tables`** • TASKING VX-Toolset: **`-O0 --switch=linear`** • Wind River Diab Compiler: **`-Xoptimized-debug-off -Xdebug -source-line-barriers-on -Xswitch-table-off`**

Build Process Requirements for All Code Coverage Metrics at a Glance

In addition to the general recommendations for the build toolchain, further adjustments may be needed for individual code coverage metrics. The following changes could be required:

- Special compiler configuration

  Special compiler configurations may be required to enhance the mapping between the object code and the source code.

- Generation of .eca files

  The .eca files supply TRACE32 with essential information needed to map the program flow's object code to the source code level, information that is not included in the compiler-generated debug information. Lauterbach provides the command line tool t32cast for this purpose.

- Source code instrumentation

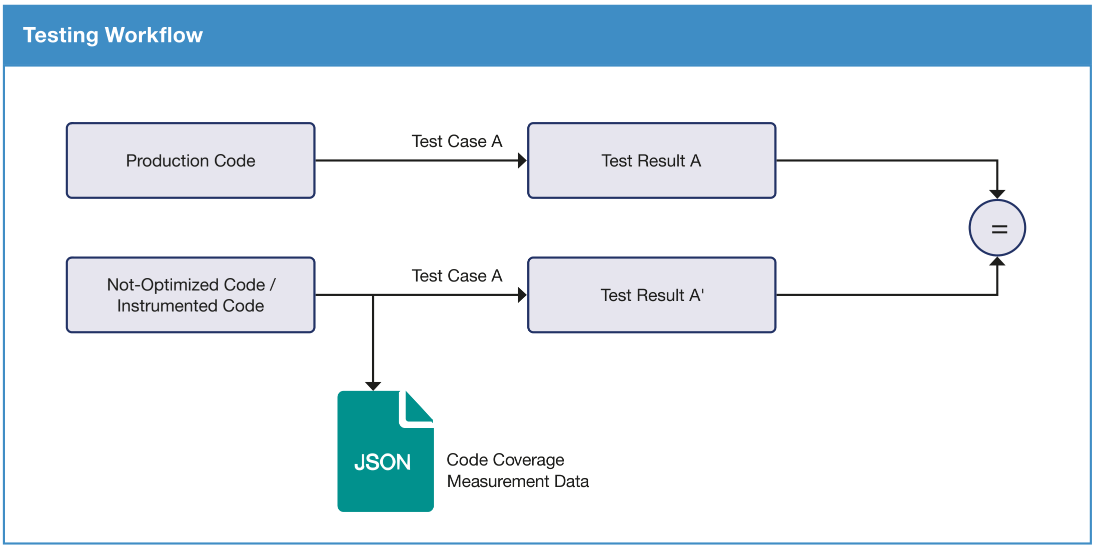
  Source code instrumentation may be required if gaps in code coverage persist after mapping the program flows's object code to the source code level.

Low compiler optimization levels are a well-known reason why TRACE32's trace-based code coverage cannot be performed on production code. Additionally, some code coverage metrics necessitate specific compiler configurations, and in some cases, code instrumentation. Therefore, there are several other factors that restrict the use of production code for TRACE32's trace-based code coverage.The table below offers an overview.

	Special Compiler Configuration	**.eca Files**	**Instrumentation**
Statement	—	—	—
Condition	—	yes, to provide condition details	likely
Decision	—	yes, to provide decision details	likely
MC/DC	—	yes, to provide condition/decision details	likely
Function	disable function inlining	—	—
Call	—	yes, to provide function call details	—
Object Code	—	—	—
ocb Decision (deprecated)	disable most optimizations	—	—

Verification of Alignment with Production Code

For safety-related projects, it is essential that the code used for coverage testing mirrors the production code exactly. Thus, both code variants should be tested side by side throughout the entire test lifecycle. The recommended testing workflow for such projects is illustrated in the figure below.



Build Process Call Coverage

TRACE32 requires the following inputs for call coverage measurement in addition to the C/C++ source files:

- A folder with the .eca files
- A non-instrumented executable

ECA files

To measure call coverage, TRACE32 needs to know the locations of function calls. This information is not contained in the debug information generated by the compiler. Therefore, Lauterbach provides a Clang-based command line tool called t32cast. This tool analyzes the C/C++ sources and generates an extended code analysis file (.eca) for each source file, containing the required location information. To generate these files, use the following command:

```
t32cast eca -o foo.c.eca foo.c
```

More details can be found in **"Command Line Parameters of t32cast"** in Application Note for t32cast, page 10 (app_t32cast.pdf).

It is recommended to integrate t32cast into your build process so that the ECA files are generated alongside the executable.

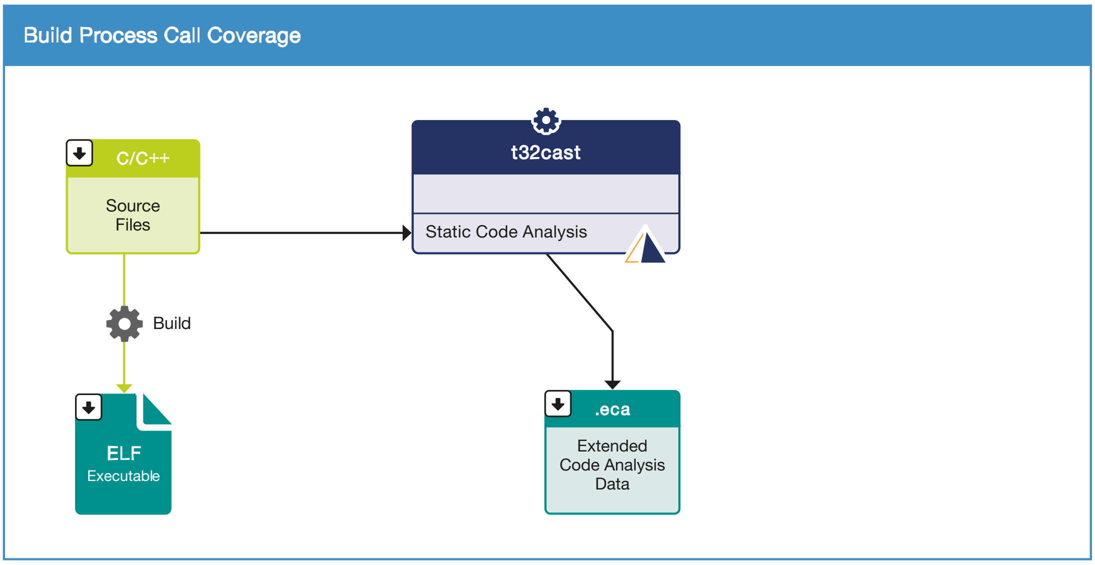


Figure: Build process for call coverage; all input/outputs of the build process that need to be loaded to TRACE32 for call coverage measurement are marked in this figure with an arrow pointing downwards.

Build Process MC/DC, Condition and Decision Coverage

If you have already chosen a code coverage mode, you can proceed directly to the relevant chapter.

- **"Build Process for Code Coverage Mode Targeted Instrumentation/No Instrumentation"**, page 52.
- **"Build Process Code Coverage Mode Breakpoint Assisted"**, page 57.
- **"Build Process Code Coverage Mode Full Instrumentation"**, page 58.

Decision Making Process

As detailed in **"Multiple Code Coverage Modes"**, page 16, several code coverage modes are available for measuring MC/DC, condition, and decision coverage. Before adapting the build process for TRACE32 code coverage measurement, you must choose the appropriate code coverage mode. Additionally, consider whether you can use a TRACE32 Instruction Set Simulator instead of a TRACE32 Debugger with the target during the build process.

Decide on the Appropriate Code Coverage Modes

The objective of this step is to choose the correct mode from the four TRACE32 code coverage modes, based on the number of observability gaps. To determine this number, follow these steps:

Note that you only need a TRACE32 debugger connected to the target hardware to detect observability gaps — trace recording is not required. The TRACE32 debugger is aware of the trace protocol properties based on the core configuration in the debugger.

1. **Build the executable**

   Please refer to **"General Recommendations for the Build Toolchain"**, page 43.

2. **Generate ECA files**

   Use t32cast to generate the ECA files for all C/C++ files. The .eca files contain the conditions/decision details necessary for detecting observability gaps. To create an ECA file with t32cast, use the command::

   ```
   t32cast eca -o foo.c.eca foo.c
   ```

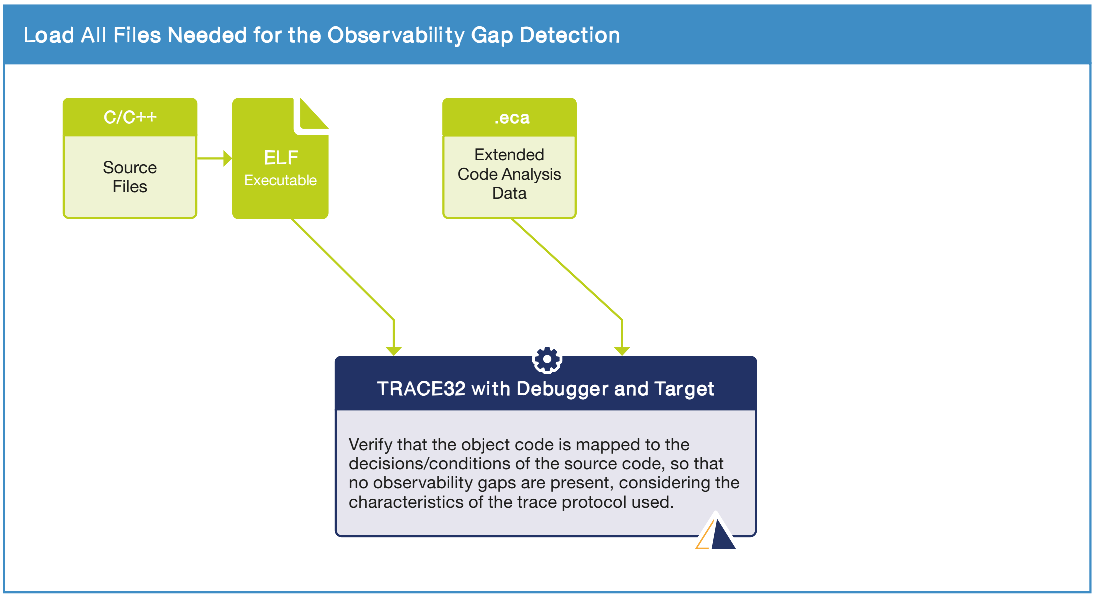
   More details can be found in **"Command Line Parameters of t32cast"** in Application Note for t32cast, page 10 (app_t32cast.pdf).

3. Load files into TRACE32

Load all files needed for observability gap detection into TRACE32. The following files must be loaded:

- Executable, which includes the paths to the source files

- Generated .eca files


Load All Files Needed for the Observability Gap Detection
C/C++
Source Files
ELF
Executable
.eca
Extended Code Analysis Data
TRACE32 with Debugger and Target
Verify that the object code is mapped to the decisions/conditions of the source code, so that no observability gaps are present, considering the characteristics of the trace protocol used.

The following commands can be used for this purpose.

```
; basic debugger setup for the target

; load the elf executable
; the elf file includes the paths to the source files
Data.LOAD.Elf "my_app.elf"

; load the .eca files
sYmbol.ECA.LOADALL /SkipErrors
```

4. Perform observability gaps detection

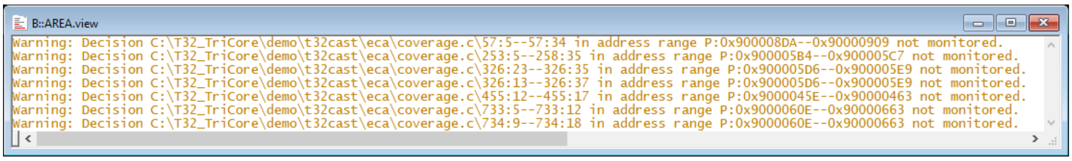
Set up and execute the mapping between decisions/conditions in the source code and the object code.

```
; clear message AREA
AREA.CLEAR

; Configure TRACE32 to account for the trace protocol in the
; mapping
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; perform mapping
sYmbol.ECA.BINary.PROCESS
; TRACE32 generates warnings when gaps in the mapping are detected
```

5. Inspect Observability Gaps

There are two ways to inspect the observabiltiy gaps:

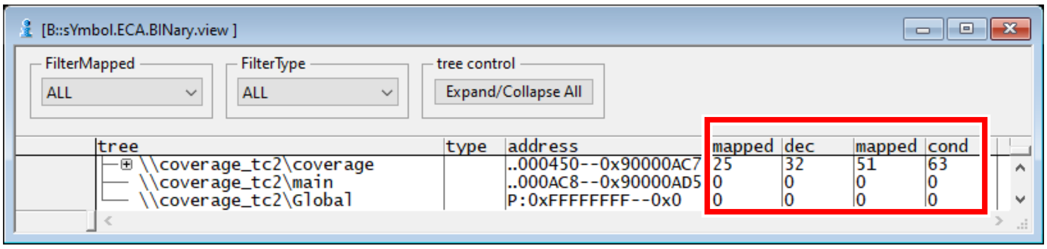
```
; inspect warnings in message AREA
AREA.view
```



```
B::AREA.view
Warning: Decision C:\T32_TriCore\demo\t32cast\eca\coverage.c\57:5--57:34 in address range P:0x900008DA--0x90000909 not monitored.
Warning: Decision C:\T32_TriCore\demo\t32cast\eca\coverage.c\253:5--258:35 in address range P:0x900005B4--0x900005C7 not monitored.
Warning: Decision C:\T32_TriCore\demo\t32cast\eca\coverage.c\326:23--326:35 in address range P:0x900005D6--0x900005E9 not monitored.
Warning: Decision C:\T32_TriCore\demo\t32cast\eca\coverage.c\326:13--326:37 in address range P:0x900005D6--0x900005E9 not monitored.
Warning: Decision C:\T32_TriCore\demo\t32cast\eca\coverage.c\455:12--455:17 in address range P:0x9000045E--0x90000463 not monitored.
Warning: Decision C:\T32_TriCore\demo\t32cast\eca\coverage.c\733:5--733:12 in address range P:0x9000060E--0x90000663 not monitored.
Warning: Decision C:\T32_TriCore\demo\t32cast\eca\coverage.c\734:9--734:18 in address range P:0x9000060E--0x90000663 not monitored.
```

Figure: A warning is displayed in the message area for each condition/decision where no mapping can be established between the source code and the object code.

```
; display decision/condition mapping overview
sYmbol.ECA.BINary.view
```



tree	type	address	mapped	dec	mapped	cond
\\coverage_tc2\coverage		..000450--0x90000AC7	25	32	51	63
\\coverage_tc2\main		..000AC8--0x90000AD5	0	0	0	0
\\coverage_tc2\Global		P:0xFFFFFFFF--0x0	0	0	0	0

Figure: The **mapped/dec** columns indicate the number of decisions in the software module (dec) and how many were successfully mapped. The **mapped/cond** columns show the number of conditions (cond) in the software module and how many were successfully mapped.

The following function returns the number of detected observability gaps as a decimal.

sYmbol.ECA.BINary.GAPNUMBER()

The result can indicate no, few, or many observability gaps. Note that using fewer optimization switches should result in fewer observability gaps. Based on the result, you need to choose the appropriate code coverage mode. The different code coverage modes are explained in **"Multiple Code Coverage Modes"**, page 16.

Decide on the Use of TRACE32 Instruction Set Simulator

A TRACE32 debugger can be used in conjunction with the target hardware to detect observability gaps, no trace recording is needed. In some cases, a TRACE32 Instruction Simulator (ISS) can also be sufficient. The benefit of using the TRACE32 ISS is that it eliminates the need for a debugger/target configuration during the build process. The TRACE32 ISS does not require a license for use in this context.

Unlike the TRACE32 debugger, the TRACE32 ISS does not automatically know the trace protocol properties after core configuration. Before using the ISS, ensure it identifies the same observability gaps as the debugger/target configuration.

Use the following command to export the observability gaps detected with a debugger/target configuration to a JSON file:

```
; export observabiltiy gaps from target test to JSON file
sYmbol.ECA.BINary.EXPORT.GAPS gaps_from_target_test.json
```

Next, perform the same test as described in **"Decide on the Appropriate Code Coverage Modes"**, page 47 with a TRACE32 Instruction Set Simulator and export the detected observability gaps to a JSON file as well:

```
; export observabiltiy gaps from ISS test to JSON file
sYmbol.ECA.BINary.EXPORT.GAPS gaps_from_iss_test.json
```

If both JSON files are identical, a TRACE32 Instruction Set Simulator can be used for the build process.

Here is some background information: The TRACE32 ISS provides program flow information through a bus trace, which differs from the flow trace protocol of the target hardware. While both trace types can be used to check whether conditional branches were evaluated as true or false, their properties may vary for conditional instructions.The table below provides an overview of key architectures, answering the following questions:

- **ISA:** Does the ISA of the core under debug include conditional instructions?
- **Trace Target:** Does the trace protocol of the core under debug generate information on the execution of conditional instructions?
- **Trace ISS:** Does the bus trace of the TRACE32 ISS provide information on the execution of conditional instructions?
- **Build with ISS:** TRACE32 ISS suitable for build process. When the **Trace Target** and the **Trace ISS** share the same properties, the TRACE32 ISS can be used during the build process to detect the observability gaps.

	ISA	Trace Target	Trace ISS	Build with ISS
Cortex-A **Cortex-R** **Cortex-M**	Yes	Yes for ETMv3, ETMv4.1 and higher No for PTM and ETMv4.0[1]	Yes	**Yes**
C6000	No	Lauterbach does not provide an Instruction Set Simulator for the C6000 core architecture.		No
C7000	No	Lauterbach does not provide an Instruction Set Simulator for the C7000 core architecture.		No
PowerArchitecture	Yes	Yes[2]	No	No
RH850	Yes	No	No	**Yes**
RISC-V	No	No	No[3]	**Yes**
AURIX™ TriCore™	Yes	No	No	**Yes**
Xtensa	Yes	Lauterbach does not provide an Instruction Set Simulator for the Xtensa core architecture.		No

[1] If the TRACE32 Instruction Set Simulator provides details on the execution of conditional instructions, but the program flow trace of the real target does not provide such information, you can disable the conditional instruction information in the TRACE32 ISS bus trace using the **SIM.ConditionTraceInfo OFF** command.

[2]If **NEXUS.HTM** is OFF, the program flow trace will not include any information on the execution of conditional instructions.

3) The TRACE32 Instruction Set Simulator for RISC-V supports only standard and ratified ISAs; custom ISAs are not supported.

Build Process for Code Coverage Mode Targeted Instrumentation/No Instrumentation

In addition to the C/C++ source files, TRACE32 requires the following inputs for code coverage measurement in code coverage mode target instrumentation/no instrumentation:

- A folder with the .eca files
- A non-instrumented executable, in the case that no observabiltiy gaps were detected
- An instrumented executable, in the case that observabiltiy gaps were detected

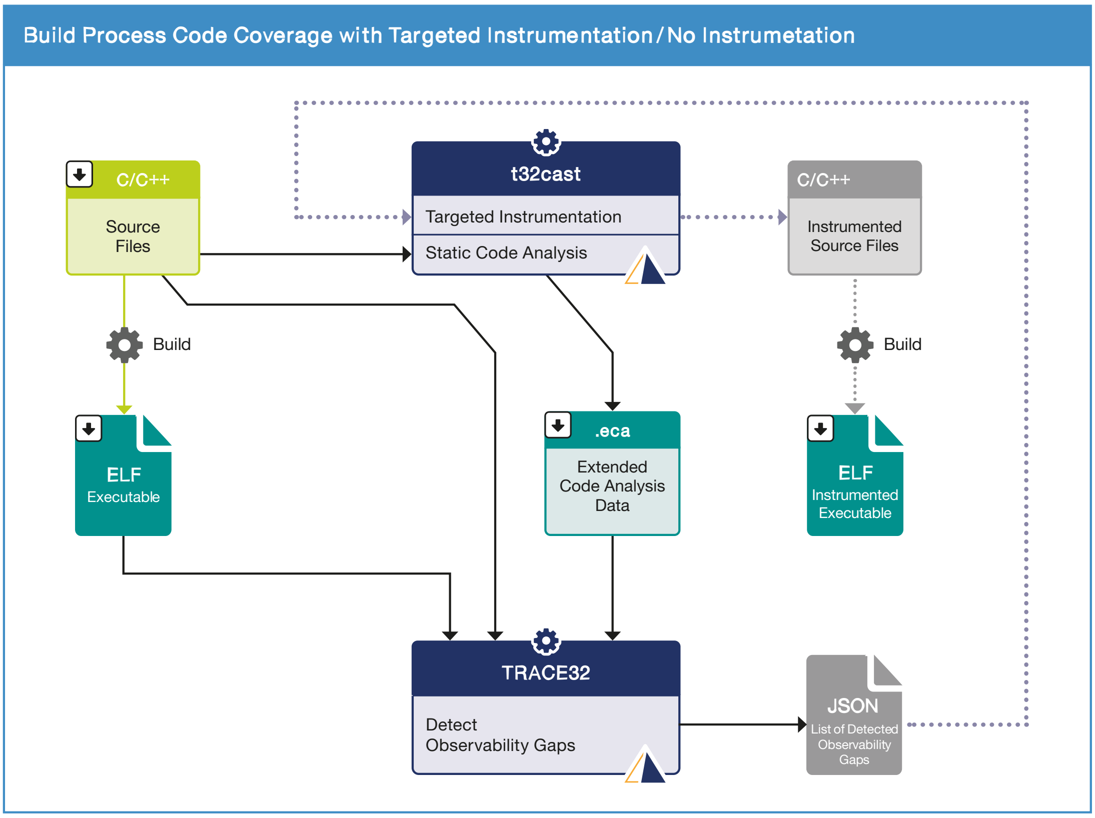

Build Process Code Coverage with Targeted Instrumentation / No Instrumetation
C/C++
Source Files
t32cast
Targeted Instrumentation
Static Code Analysis
C/C++
Instrumented Source Files
Build
Build
ELF
Executable
.eca
Extended Code Analysis Data
ELF
Instrumented Executable
TRACE32
Detect Observability Gaps
JSON
List of Detected Observability Gaps

Figure: All inputs/outputs of the build process that may need to be loaded into TRACE32 for coverage measurement in code coverage mode target instrumentation/no instrumentation are indicated in this figure by a downward-pointing arrow.

You need to extend your build process as follows:

1. **Add t32cast to generate the ECA files for all C/C++ files.**

   To create an ECA file with t32cast, please use the command:

   ```
   t32cast eca --export-cfg -o foo.c.eca foo.c
   ```

   More details can be found in **“Command Line Parameters of t32cast”** in Application Note for t32cast, page 10 (app_t32cast.pdf).

2. **Add TRACE32 to perform the observability gap check.**

   TRACE32 can be called from the Make file with a script that performs the check automatically. Please refer to **“Command Line Arguments for Starting TRACE32”** in TRACE32 Installation Guide, page 53 (installation.pdf) for details. This could look like the following:

   ```
   t32ecagaps.json: $(NAME).elf $(ECA)
   $(T32GRP)\t32marm.exe -c ../common/trace32.cfg -s ../common/export_gaps.cmm  $(NAME).elf
   ```

   You should have checked in step **“Decide on the Use of TRACE32 Instruction Set Simulator”**, page 50, if you can use a TRACE32 Instruction Set Simulator instead of a debugger/target configuration.

The script that runs in TRACE32 must include the following steps.

```
LOCAL &instrumented_tree
&instrumented_tree="./instrumented_sources"

; basic debugger setup for the target or basic ISS setup
…

; load the elf executable
; the elf file includes the paths to the source files
Data.LOAD.Elf "my_app.elf"

; load the .eca files
sYmbol.ECA.LOADALL /SkipErrors

; delete file og_detected.txt, if existing
IF FILE.EXIST(&instrumented_tree/og_detected.txt)
(
  RM &instrumented_tree/og_detected.txt
)

; Configure TRACE32 to account for the trace protocol used in the
; mapping
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; perform mapping
sYmbol.ECA.BINary.PROCESS

; Create a file named og_detected.txt to serve as a flag indicating
; that an instrumented executable needs to be generated
IF sYmbol.ECA.BINary.GAPNUMBER()>0.
(
     OPEN #1 &instrumented_tree/og_detected.txt /Create
     WRITE #1 "Observability gaps have been detected,"
     WRITE #1 "necessitating the generation of an instrumented"
     WRITE #1 "executable."
     CLOSE #1
)
sYmbol.ECA.BINary.EXPORT.AdJoinGAPS
```

3.A If the file `og_detected.txt` **does not exist, the non instrumented executable** `my_app.elf` **can be used for code coverage measurement in no instrumentation mode.**

3.B If the file `og_detected.txt` does exist, use t32cast to perform targeted instrumentation.

The result of this step should be a structure of directories (**instrumented_tree** in the figure below).

- For each source file that contains observabiltiy gaps, there is an instrumented version of this file in the **instrumented_tree** directory (hatched rectangles for instrumented source files in the figure below).

- For each source file that does not contain observabiltiy gaps, there is a copy of the original source in the **instrumented_tree** directory (white rectangles for not-instrumented source files in the figure below).

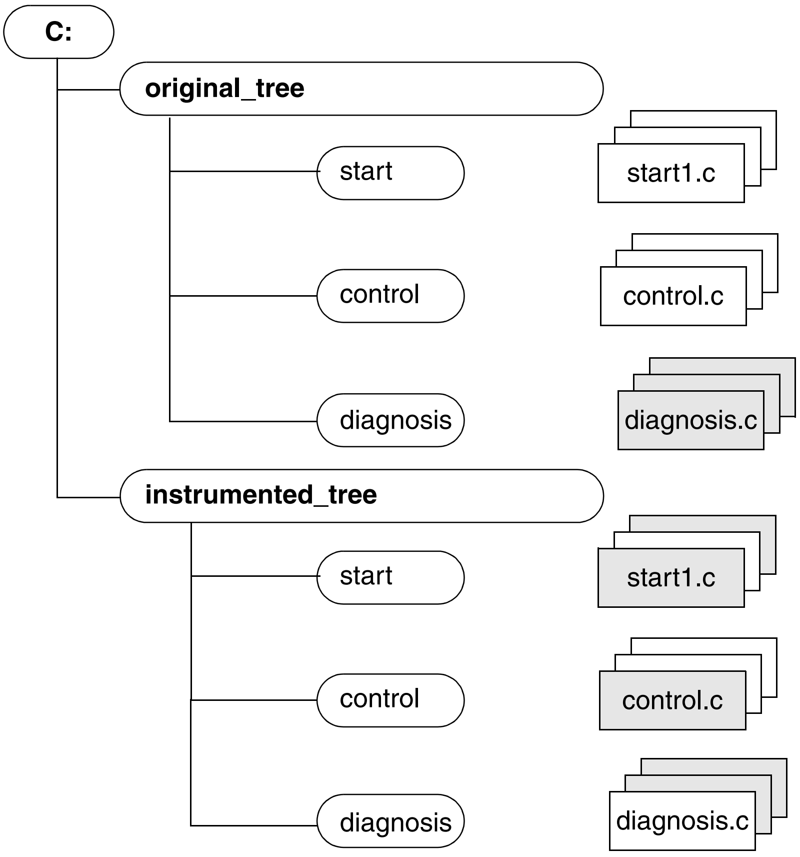


Figure: The instrumentation does not add any extra lines of source code. By preserving the structure of the **original_tree** directory in the **instrumented_tree** directory, TRACE32 can be configured to use the original, non-instrumented sources during testing.

To perform the code instrumentation task with t32cast, please use the following commands:

```
; create additional C source files with definitions of the
; instrumentation hooks
t32cast instrument --mode=mcdc --gen-instr-source-files
--probe-dir=instrumented_tree
; the files t32pp.c and t32pp.h created this way have to be compiled
; together with the instrumented source files


; process all source files

; use JSON files with observabiltiy gaps as input for targeted
; instrumentation and instrument all decisions for which
; a observabiltiy gap was detected

; source files without observabiltiy gaps should be simply
; copied to instrumented_tree
t32cast instrument --mode=mcdc
--filter=instrumented_tree\foo.c.json
-o instrumented_tree\foo.c original_tree\foo.c
```

Whereby the switch `mode=mcdc` must also be used for condition and decision coverage.

4.B Build the instrumented executable.

Build Process Code Coverage Mode Breakpoint Assisted

In addition to the C/C++ source files, TRACE32 requires the following inputs for code coverage measurement in breakpoint assisted code coverage mode:

- A folder with the .eca files
- A non-instrumented executable

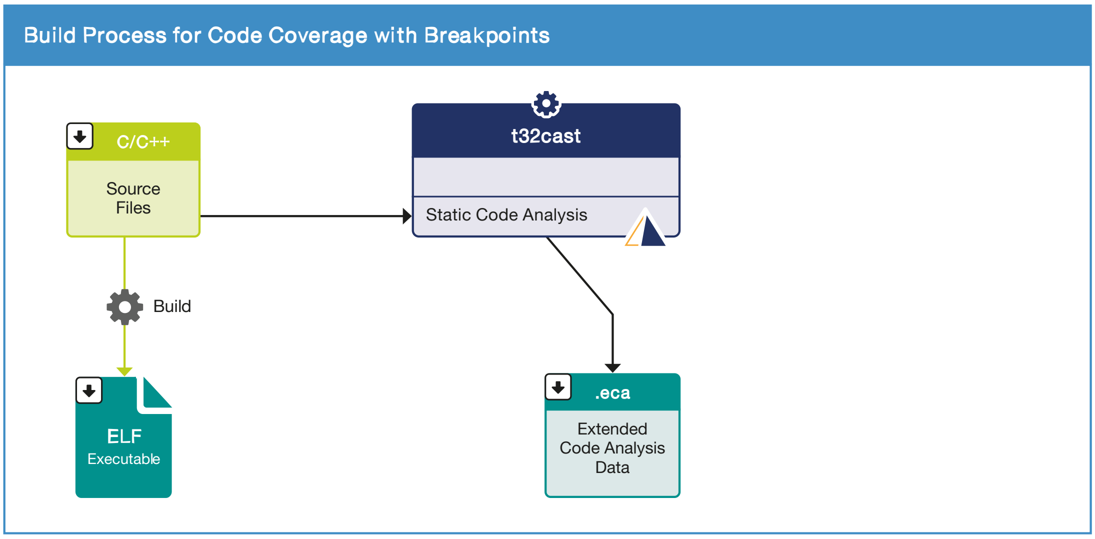

Build Process for Code Coverage with Breakpoints
C/C++
Source Files
Build
ELF
Executable
t32cast
Static Code Analysis
.eca
Extended Code Analysis Data

Figure: All input/outputs of the build process that are required for coverage measurement in breakpoint assisted code coverage mode are marked with an arrow pointing downwards.

The build process must be extended so that t32cast creates an ECA file for each source code file that is compiled. Please use the command:

```
t32cast eca -o foo.c.eca foo.c
```

More details can be found in **"Application Note for t32cast"** (app_t32cast.pdf).

Build Process Code Coverage Mode Full Instrumentation

TRACE32 requires the following inputs for code coverage with full instrumentation in addition to the C/C++ source files:

- A folder with the .eca files
- An instrumented executable

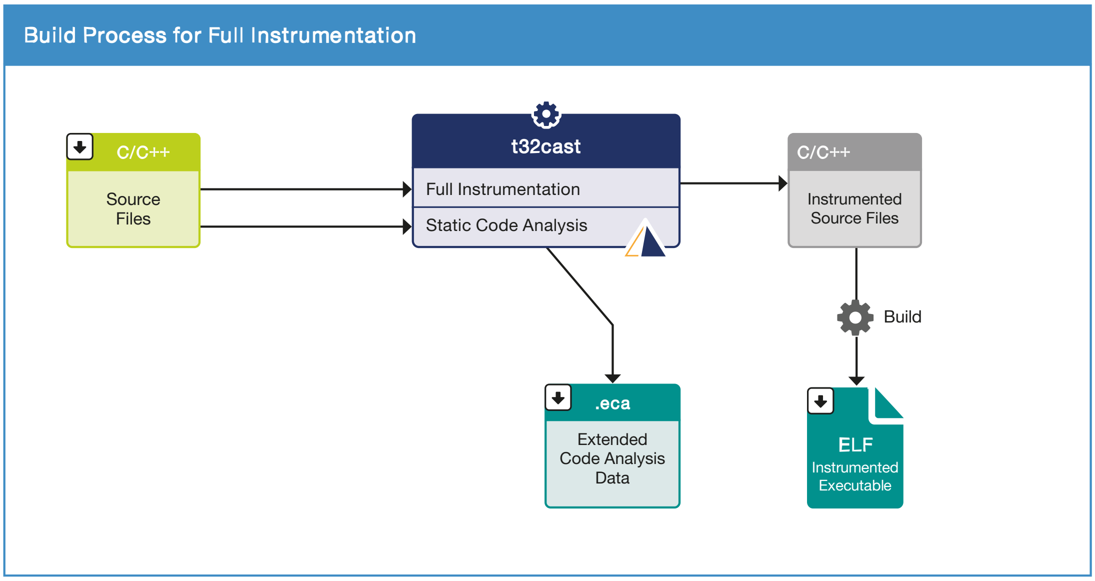


Figure: All input/outputs of the build process that are required for coverage measurement in code coverage mode full instrumentation are marked with an arrow pointing downwards.

The build process must be extended so that t32cast creates an ECA file for each source code file that is compiled. Please use the command:

```
    t32cast eca -o foo.c.eca foo.c
```

More details can be found in **“Application Note for t32cast”** (app_t32cast.pdf).

In addition, all C/C++ source files must be instrumented with t32cast, resulting in a directory structure containing all the instrumented source files. The instrumentation does not add any extra lines of source code. By preserving the structure of the **original_tree** directory in the **instrumented_tree** directory, TRACE32 can be configured to use the original, non-instrumented sources during testing.

```
; create additional C source files with definitions of the
instrumentation hooks
t32cast instrument --mode=mcdc --gen-instr-source-files
--probe-dir=<instr_dir>
; the files t32pp.c and t32pp.h created this way have to be compiled
; together with the source files

; instrument all decisions/conditions in all source files
t32cast instrument --mode=mcdc -o <instr_dir\file> <org_dir\file>
```

Note that the `--mode=mcdc` switch must be used also for condition and decision coverage.

Finally, an instrumented executable must be generated.

SMP Multicore Systems

If code coverage is performed on an **SMP system**, it is typically sufficient to prove that the object or source code line was executed by one of the cores. For this reason the core number of the trace records is ignored, when the trace information is transferred to the code coverage system.

iAMP Multicore Systems

The machine ID is used to load the executable and debug symbols onto a debug cluster in an **iA MP system**. For example:

```
; application_subsystem1.elf is loaded to the symbol space of machine 1
Data.LOAD.Elf application_subsystem1.elf 0x1:::0 /NoClear /NoCODE
```

This machine ID is required to link the object code to the source code. Therefore, the machine ID is stored along with the object code address in the internal TRACE32 code coverage system. Since the JSON file for code coverage metrics — such as statement coverage, decision coverage, condition coverage, modified condition/decision coverage (MC/DC), and both call and function coverage — contains only the source code, it does not include the machine ID.

TRACE32 Tool Configurations

The following TRACE32 tools are suitable for code coverage:

- **TRACE32 Debugger and Off-Chip Trace**

  A **PowerTrace** module is required for trace recording. **Trace.METHOD Analyzer** is automatically selected as soon as TRACE32 detects a PowerTrace module in its hardware configuration.

  Some processors, like most Cortex-M processors, can export program flow via a 4-bit trace interface. In such scenarios, a TRACE32 CombiProbe or a MikroTrace can also serve the purpose. **Trace.METHOD CAnalyzer** is automatically selected as soon as TRACE32 detects a TRACE32 CombiProbe or a MikroTrace module in its hardware configuration.

- **TRACE32 Debugger and On-Chip Trace**

  The **Trace.METHOD Onchip** command configure TRACE32 for onchip tracing. Onchip tracing is also possible via XCP.

- The **Trace.METHOD** HAnalyzer command configures TRACE32 for USB tracing. Since this trace memory is located on the host computer, you must define its size in advance using the **HAnalyzer.SIZE** command.

- **TRACE32 Instruction Set Simulator**

The TRACE32 Instruction Set Simulator simulates the instruction set, but does not model timing characteristics and peripherals. However, the simulator provides a bus trace so that code coverage is easy to perform.

There is also the option of performing the code coverage analysis with a TRACE32 Instruction Set Simulator (**Trace.METHOD Analyzer**).

- **TRACE32 Advanced Register Trace (ART)**

If Lauterbach does not offer an Instruction Set Simulator for the core architecture you are using, you can also use the TRACE32 Advanced Register Trace (Trace.METHOD ART). This is a single-step trace, which makes program execution very slow. This method is therefore only suitable for unit testing.

- **TRACE32 Debugger for virtual targets with trace support**

TRACE32 Debuggers for virtual targets should, because of their limitations, only be used for code coverage if needed. For details refer to **“Code Coverage with Virtual Targets”**, page 96.

A TRACE32 debug and trace tool is of course the best choice, as it allows testing in the target environment and thus integrates hardware and software. But for test phases that do not have these requirements, a TRACE32 Instruction Set Simulator can be a good choice. It has a number of advantages: it allows early testing when the target hardware is not yet available, scales well and delivers results quickly.

Choose the Appropriate Trace Data Collection Variant

The following overview is intended to help new users to make a decision for the appropriate trace data collection method. It is deliberately simplified and complex details are avoided.

If you are using a TRACE32 Advanced Register Trace (**Trace.METHOD ART**), please refer to **"ART Mode Code Coverage"**, page 98.

Trace Data Collection Variant	**Incremental in Leash Mode (fallback)**	**Incremental in STREAM Mode**	**Continuous SPY**	**Continuous RTS**
Code Coverage Measurement	Trace data is recorded, and the program is stopped once the trace memory is full. The recorded data is then processed for code coverage. Multiple recording steps are necessary to gather sufficient measurement data. The size of the trace memory limits the amount of data that can be recorded in a single recording step.	Trace data is recorded and streamed to the host. The program can be stopped once sufficient data has been collected. The recorded data is then processed for code coverage.	The trace data is recorded and streamed to the host. Streaming is periodically suspended to process the data for code coverage.	Trace data is simultaneously recorded, streamed to the host, and processed for code coverage.
Supported Traces	- TRACE32 Instruction Set - Simulator - Onchip trace - Trace of virtual targets - PowerTrace - µTrace Cortex-M - µTrace RISC-V 32-Bit - CombiProbe for Cortex-M - CombiProbe for RISC-V 32-Bit	- PowerTrace - µTrace Cortex-M - µTrace RISC-V 32-Bit - CombiProbe for Cortex-M - CombiProbe for RISC-V 32-Bit		
Supported Trace Protocols	all	all	all	- EMTv3, PTM, ETMv4 for Arm/Cortex - MCDS for Infineon AURIX - Nexus for MPC5xxx/STM SPC5xx - Nexus for PPC QorIQ
Supported Coverage Metrics	all	all	all	all
Restrictions	none	Not suitable for high-bandwidth trace ports.	Not suitable for high-bandwidth trace ports. Only partially suitable for rich OSes.	

Best Practices for Trace Recording

Reduce the Amount of Trace Data

It is recommended to reduce the amount of trace data to the required minimum to make best use of the available trace memory. If trace information is exported off-chip via a dedicated trace port this reduction can also help to avoid an overload of the trace port.

It is recommended to configure the onchip trace logic:

- to generate only trace information for the program flow.
- to generate additionally trace information for the task switches if a rich OS such as Linux is used.
- to not generate **chip timestamps** if supported by the trace protocol.

Details of how to do this can be found in the manuals:

- Arm: **“Training Arm CoreSight ETM Tracing”** (training_arm_etm.pdf), **“Training Cortex-M Tracing”** (training_cortexm_etm.pdf)
- MPC5xxx/SPC5xxx, QorIQ and RH850: **“Training MPC5xxx/SPC5xx Nexus Tracing”** (training_nexus_mpc5500.pdf)
- TriCore: **“Training AURIX Tracing”** (training_aurix_trace.pdf)
- For other processor architectures, please refer to the corresponding **“Processor Architecture Manuals”**.

For target systems using a rich OS such as Linux a method of determining task switches must also be included in the trace data. More information can be found here:

  - **“Training Linux Debugging”** (training_rtos_linux.pdf).
  - For other operating systems, please refer to the corresponding **“OS Awareness Manuals”** (rtos_*<os>*.pdf).

Ensure a Fault-Free Trace Recording

Before you start with code coverage, it is recommended to check if the trace recording is working properly. Here is a simple script:

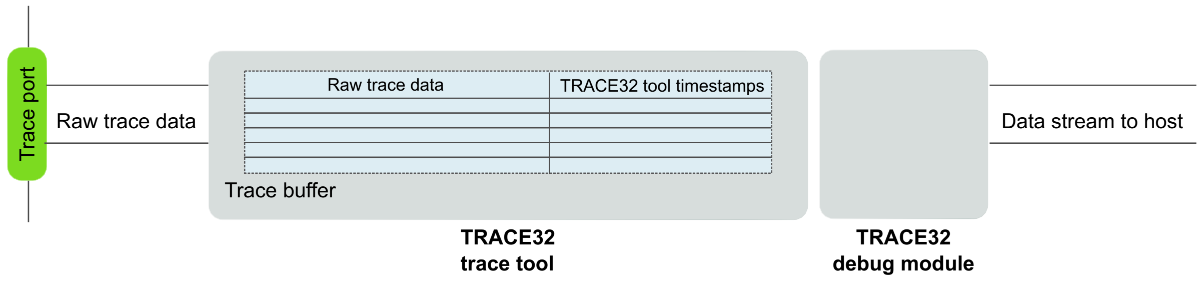
```
Go
Break
SILENT.Trace.Find FLOWERROR /ALL
IF FOUND.COUNT()!=0.
(
  PRIVATE &msg
  &msg="FLOWERRORS were found in the analyzed trace recording."
  &msg="&msg It is recommended to check"
  &msg="&msg if the trace recording works properly."
  ECHO FOUND.COUNT() "&msg"
)
ELSE
(
  ECHO "The analyzed trace recording does not contain FLOWERRORS."
)
ENDDO
```

The code coverage analysis can tolerate individual **FLOWERRORs**. However, it is recommended to ensure that the number of FLOWERRORs is as small as possible.

The code coverage analysis can tolerate gaps in the trace caused by **TARGET FIFO OVERFLOWs** but this will result in gaps in the coverage data.

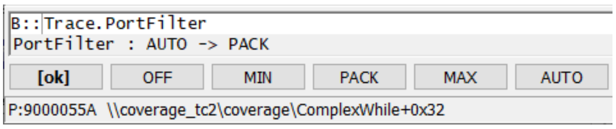
Disable Timestamps for Trace Streaming

All general rules applying to trace streaming are described under **Trace.Mode STREAM**.



Since the timestamps that TRACE32 assigns for the trace records have no significance for code coverage, they do not have to be streamed to the host computer. This considerably reduces the data rate. Please use the command **Trace.PortFilter MAX** for this purpose.

The current **PortFilter** setting is displayed in the TRACE32 state line when you enter the command **Trace.PortFilter** followed by a space.



```
B::Trace.PortFilter
PortFilter : AUTO -> PACK
```

```
P:9000055A  \\coverage_tc2\coverage\ComplexWhile+0x32
```

Steps in Preparation for Trace Data Collection

Notes on the Individual Test Variants

This chapter describes which files need to be loaded into TRACE32 for trace data recording and code coverage analysis. In fact, some files are only required for the code coverage analysis. First, general notes on the individual test variants:

Incremental code coverage (one test run with repeated cycles)

With incremental code coverage, the following two steps must be repeated until the test is complete.

1. Run program execution and record program flow to trace memory.
2. Upload trace contents to the host and perform code coverage analysis in TRACE32 PowerView GUI.

For this test scenario, we recommend loading all files in advance.

Incremental code coverage (two separate test runs)

In this test variant, the recording of the trace data and the code coverage analysis are mostly carried out by two different teams.

1. The trace team is exclusively responsible for trace recording. Each individual trace recording is saved in a file (command **Trace.SAVE**). The trace files are then passed on to the code coverage team for analysis.

   This means that the trace team does not have to load any files that are only required for code coverage. Files that are only required for the code coverage analysis are therefore marked with *(code coverage only)* in this chapter.

2. The code coverage team is exclusively responsible for the code coverage analysis. Each individual trace file is loaded (command **Trace.LOAD**), the code coverage analysis is performed and the result is added incrementally to the preceding analysis results.

   The code coverage team must always load all files.

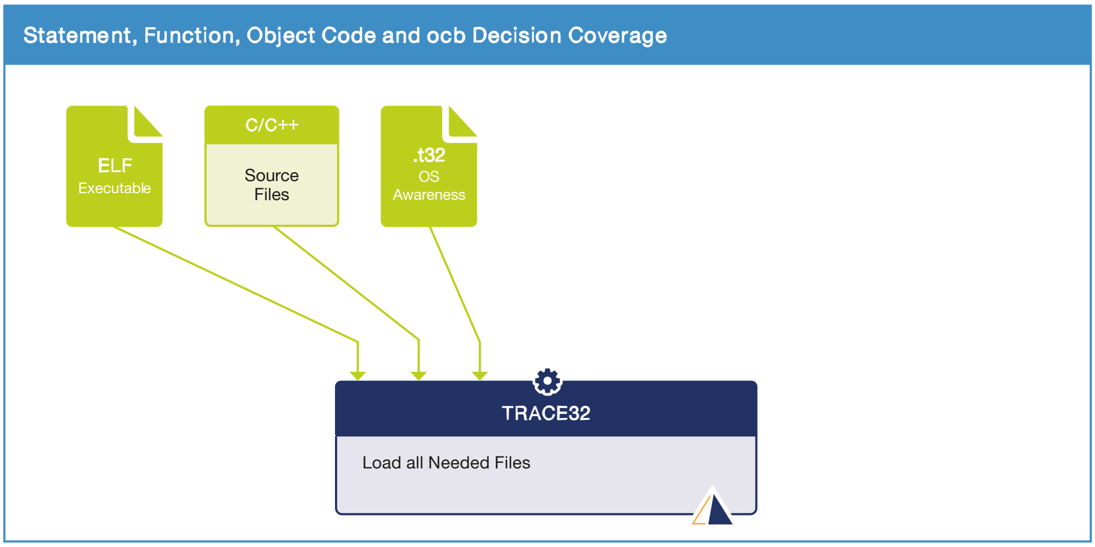
Live code coverage (RTS, SPY)

With live code coverage, everything is done at the simultaneous. Run program execution and record program flow, stream trace data to host and perform code coverage analysis in TRACE32 PowerView GUI.

For this test variant, all files must be loaded in advance. Since everything has to be performed quickly here, the executable must be mirrored in the **TRACE32 Virtual Memory**. (The code is usually read from the target memory to perform the decoding of the trace data. But this procedure is too slow for live code coverage.)

Preparation for Statement, Function, Object Code, ocb Decision Coverage

All trace data collection variants can be used here.



The following files need to be loaded into TRACE32:

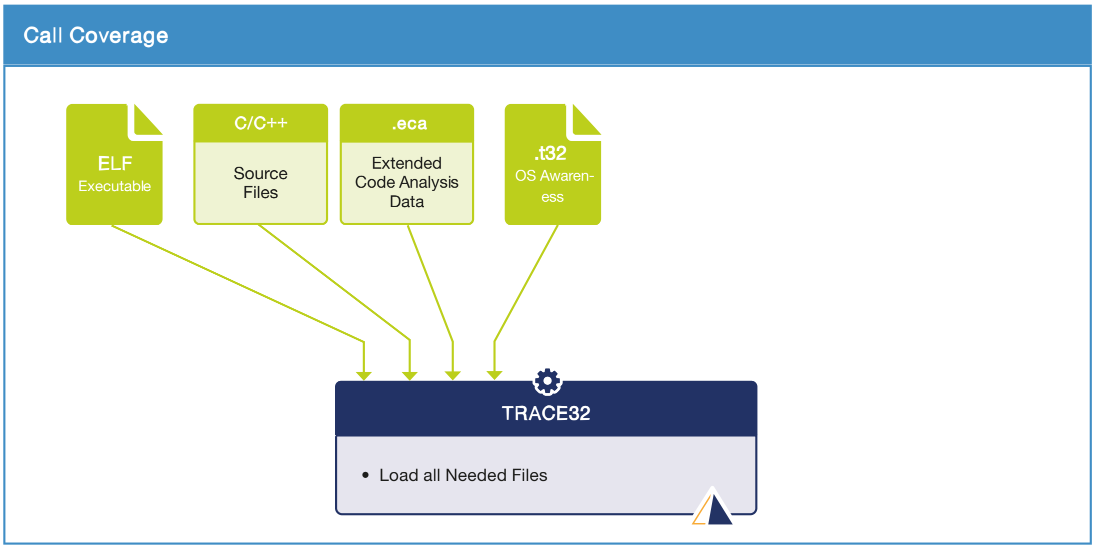
- Executable, which includes paths to all source files
- TRACE32 OS Awareness, if an operating system is used by the target application

The following commands can be used for this purpose:

```
; basic debug and trace setup

; load the elf executable
; the elf file includes the paths to the source files
Data.LOAD.Elf "my_app.elf"

; mirror the executable to the TRACE32 Virtual Memory
; live code coverage (RTS, SPY) only
Data.LOAD.Elf "my_app.elf" /VM

; load the OS Awareness
TASK.CONFIG myos.t32

; detect memory address ranges at the end of functions that were
; inserted due to memory alignment and removes them from the function
; address range
sYmbol.CLEANUP.AlignmentPaddings
```

Preparation for Call Coverage

All trace data collection variants can be used here.



The following files need to be loaded into TRACE32:

- Executable, which includes paths to all source files
- Generated .eca files *(code coverage only)*
- TRACE32 OS Awareness, if an operating system is used by the target application

The following commands can be used for this purpose:

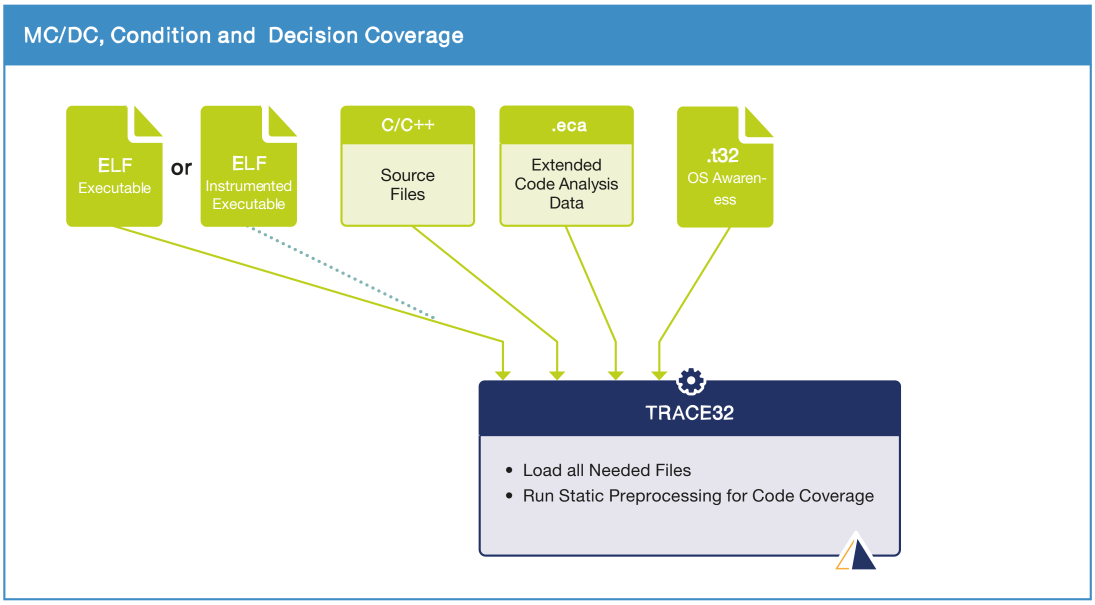
```
; basic debug and trace setup

; load the elf executable
; the elf file includes the paths to the source files
Data.LOAD.Elf "my_app.elf"

; mirror the executable to the TRACE32 Virtual Memory
; live code coverage (RTS, SPY) only
Data.LOAD.Elf "my_app.elf" /VM

; load the .eca files
sYmbol.ECA.LOADALL /SkipErrors

; load the OS Awareness
TASK.CONFIG myos.t32

; detects memory address ranges at the end of functions that were
; inserted due to memory alignment and removes them from the function
; address ranges.
sYmbol.CLEANUP.AlignmentPaddings
```

Preparation for MC/DC, Condition and Decision Coverage

The preparation is different for the individual code coverage modes:

- **“Preparation for Targeted Instrumentation/No Instrumentation”**, page 70.
- **“Preparation for Code Coverage with Breakpoints”**, page 72.
- **“Preparation for Full Instrumentation”**, page 75

Preparation for Targeted Instrumentation/No Instrumentation


MC/DC, Condition and Decision Coverage
ELF Executable
or
ELF Instrumented Executable
C/C++ Source Files
.eca Extended Code Analysis Data
.t32 OS Awareness
TRACE32
Load all Needed Files
Run Static Preprocessing for Code Coverage

The following files need to be loaded into TRACE32:

- Not-instrumented executable or the instrumented executable. Each executable includes the paths to all source files.

NOTE:	Please note that TRACE32 performs the code coverage analysis for the instrumented executable with the original, non-instrumented source code files. For this reason, the paths to the source code files included in the instrumented executable file must always be adapted accordingly. The **sYmbol.SourcePATH** command group offers various ways of doing this. An introduction to this topic can be found in **“Option and Commands to Get the Correct Paths for the HLL Source Files”** in Training Source Level Debugging, page 9 (training_source_level_debugging.pdf)

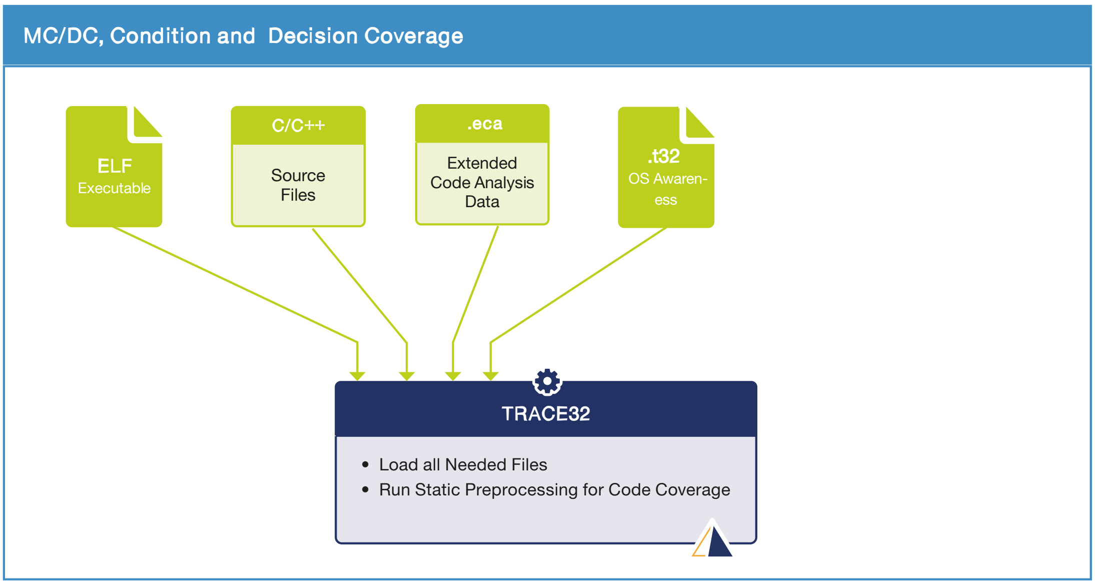
- Generated .eca files *(code coverage only)*
- TRACE32 OS Awareness, if an operating system is used by the target application.

After loading all the necessary files, static preprocessing must be performed to prepare the MC/DC, condition or decision coverage analysis *(code coverage only)*.

The following framework can be used for this purpose:

```
; basic debug and trace setup

; load appropriate executable

; adjust the links to source files in "my_app_targeted.elf" so that
; they refer to the non-instrumented source files
IF FILE.EXIST(gaps.json)
(
   Data.LOAD.Elf "my_app_targeted.elf"
   sYmbol.SourcePATH.Translate "c:/my_app/instrumented" "c:/my_app/source"
    PRINT "Executable with targeted instrumentation loaded."
)
ELSE
(
   Data.LOAD.Elf "my_app.elf"
   PRINT "Not-instrumented executable loaded."
)

; load the .eca files
sYmbol.ECA.LOADALL /SkipErrors

; load the OS Awareness
TASK.CONFIG myos.t32

; detects memory address ranges at the end of functions that were
; inserted due to memory alignment and removes them from the function
; address ranges
sYmbol.CLEANUP.AlignmentPaddings

; Configuration of static preprocessing in preparation for
, code coverage analysis

; consider conditional opcodes in the object code
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; consider source code instrumentation probes in "my_app_targeted.elf"
IF &instrumented
(
   sYmbol.ECA.BINary.ControlFlowMode.INSTR ON
)

; perform the static analysis for MC/DC, condition and decision coverage
sYmbol.ECA.BINary.PROCESS

IF sYmbol.ECA.BINary.GAPNUMBER()>0.
(
   PRINT sYmbol.ECA.BINary.GAPNUMBER() " observability gaps detected. \
   Please check the remaining observability gaps."
)
```

Preparation for Code Coverage with Breakpoints


MC/DC, Condition and Decision Coverage
ELF
Executable
C/C++
Source Files
.eca
Extended Code Analysis Data
.t32
OS Awaren-ess
TRACE32
Load all Needed Files
Run Static Preprocessing for Code Coverage

The following files need to be loaded into TRACE32:

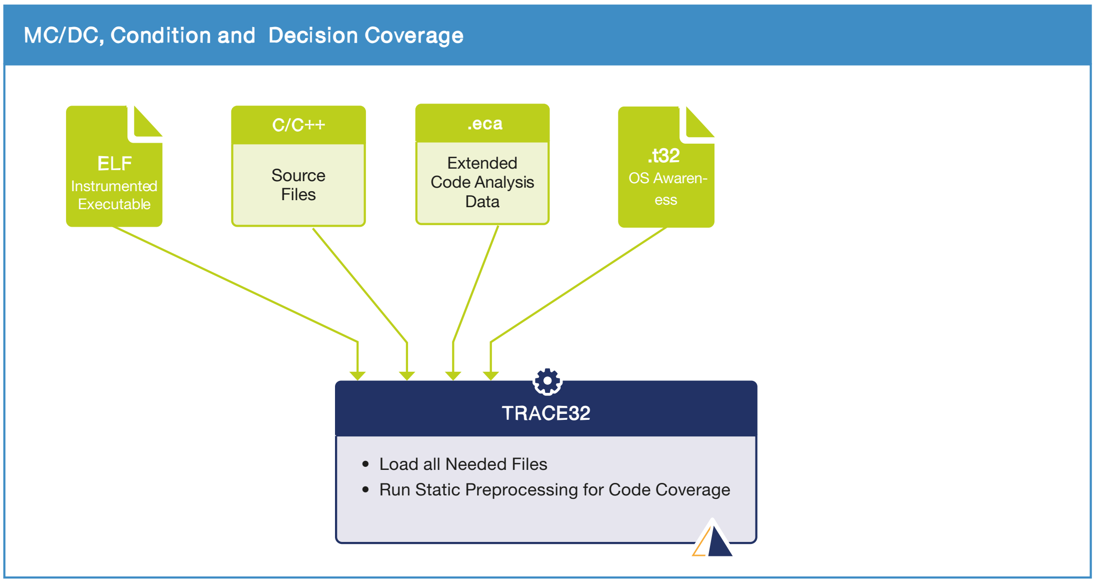
- Not-instrumented executable, which includes the links to all source files.
- Generated .eca files.
- TRACE32 OS Awareness, if an operating system is used by the target application.

After loading all the necessary files, static preprocessing must be performed to prepare the MC/DC, condition or decision coverage analysis.

The following framework can be used for this purpose:

```
; basic debug and trace setup

; load executable
Data.LOAD.Elf "my_app.elf"

; load the .eca files
sYmbol.ECA.LOADALL /SkipErrors

; load the OS Awareness
TASK.CONFIG myos.t32

; detects memory address ranges at the end of functions that were
; inserted due to memory alignment and removes them from the function
; address ranges
sYmbol.CLEANUP.AlignmentPaddings

; Configuration of static preprocessing in preparation for
, code coverage analysis

; consider conditional opcodes in the object code
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; perform the static analysis for MC/DC, condition and decision coverage
sYmbol.ECA.BINary.PROCESS
```

Now trace data collection for code coverage can be started. For details refer to **“Trace Data Collection and Code Coverage Measurement”, page 77**.


MC/DC, Condition and Decision Coverage
ELF
Instrumented Executable
C/C++
Source Files
.eca
Extended Code Analysis Data
.t32
OS Awareness
TRACE32
Load all Needed Files
Run Static Preprocessing for Code Coverage

The following files need to be loaded into TRACE32:

- Instrumented executable

NOTE:	Please note that TRACE32 performs the code coverage analysis for the instrumented executable with the original, non-instrumented source code files. For this reason, the paths to the source code files included in the instrumented executable file must always be adapted accordingly. The **sYmbol.SourcePATH** command group offers various ways of doing this. An introduction to this topic can be found in **"Option and Commands to Get the Correct Paths for the HLL Source Files"** in Training Source Level Debugging, page 9 (training_source_level_debugging.pdf)

- Generated .eca files *(code coverage only)*
- TRACE32 OS Awareness, if an operating system is used by the target application.

After loading all the necessary files, static preprocessing must be performed to prepare the MC/DC, condition or decision coverage analysis *(code coverage only).*

The following framework can be used for this purpose:

```
; basic debug and trace setup

; load executable
Data.LOAD.Elf "my_app_full.elf"

; load the .eca files
sYmbol.ECA.LOADALL /SkipErrors

; adjust the paths to source files in "my_app_full.elf" so that
; they refer to the non-instrumented source files
sYmbol.SourcePATH.Translate "c:/my_app/instrumented" "c:/my_app/source"

; load the OS Awareness
TASK.CONFIG myos.t32

; detects memory address ranges at the end of functions that were
; inserted due to memory alignment and removes them from the function
; address ranges
sYmbol.CLEANUP.AlignmentPaddings

; Configuration of static preprocessing in preparation for
, code coverage analysis

; configure TRACE32 to consider trace event of conditional
; branches/instructions as source for monitoring
; decisions/conditions for code coverage
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; configure TRACE32 to consider trace source code instrumentation probes
; in "my_app_full.elf" as source for monitoring decisions/conditions for
; code coverage
sYmbol.ECA.BINary.ControlFlowMode.INSTR ON

; perform the static analysis for MC/DC, condition and decision coverage
sYmbol.ECA.BINary.PROCESS

IF sYmbol.ECA.BINary.GAPNUMBER()>0.
(
   PRINT sYmbol.ECA.BINary.GAPNUMBER() " observability gaps detected. \
   Please check the remaining observability gaps."
)
```

Trace Data Collection and Code Coverage Measurement

This chapter provides detailed instructions for performing the different variants of trace data collection.

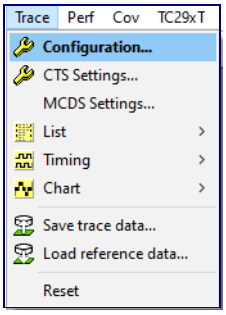
- **“Incremental Code Coverage”**, page 77.
- **“Incremental Code Coverage in STREAM Mode”**, page 80.
- **“RTS Mode Code Coverage”**, page 84.
- **“SPY Mode Code Coverage”**, page 90.

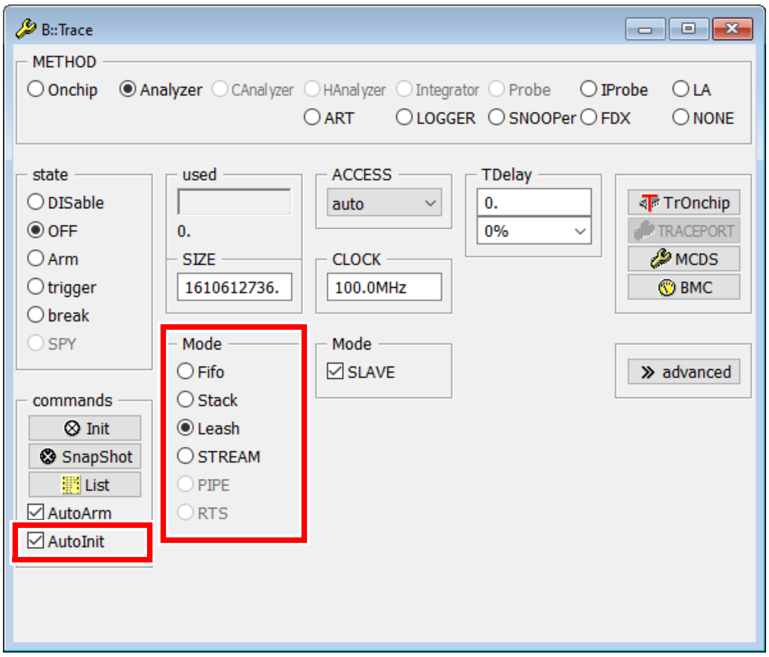
Incremental Code Coverage

Incremental coverage is supported by all processor architectures which provide information about program flow that is saved to trace buffer and all TRACE32 configurations. It also supports all code coverage metrics supported by TRACE32. **It is a reliable fallback methods that can be used in the vast majority of situations.**

Data Collection

1. Set the trace to Leash Mode either via the **Trace configuration** window or via the command **Trace.Mode Leash**. This ensures that the target will halt when the trace buffer becomes nearly full, preventing loss of data. Stack or Fifo mode can also be used if Leash Mode is not supported.
2. Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to ensure that the trace buffer is always cleared before the trace recording is started.





METHOD: Onchip [✓ Analyzer] CAnalyzer HAnalyzer Integrator Probe IProbe LA ART LOGGER SNOOPer FDX NONE

state: DISable [✓ OFF] Arm trigger break SPY

used: 0.

SIZE: 1610612736.

ACCESS: auto

CLOCK: 100.0MHz

TDelay: 0. 0%

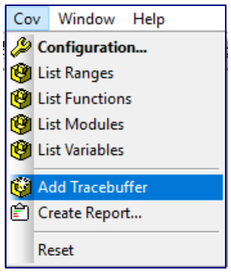
Mode: Fifo Stack [✓ Leash] STREAM PIPE RTS

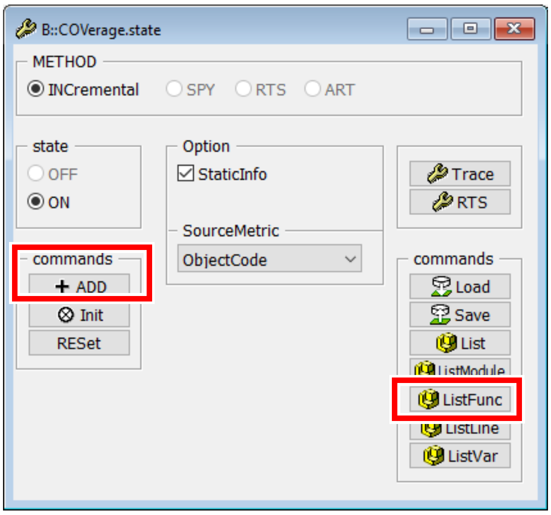
Mode: [✓ SLAVE]

commands: [✓ AutoArm] [✓ AutoInit]

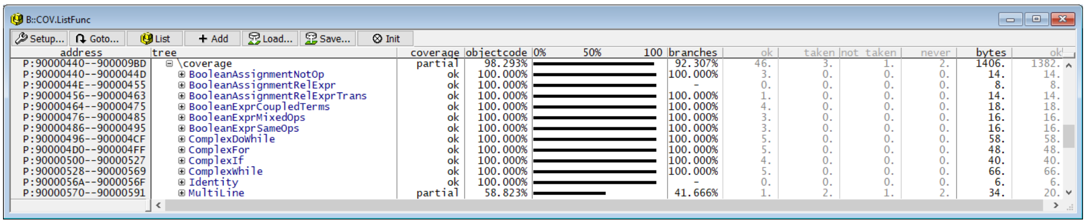
3. Start program execution and wait until it stops.

4. After program execution has stopped, the trace data can be added to the coverage system with the **COVerage.ADD** command or by using the **+ADD** button in the **COVerage Configuration** window, or by selecting '**Add Tracebuffer**' from the **Cov** menu (shown in the image below).





5. The code coverage measurement can be displayed by using the **ListFunc** button in the **COVerage Configuration** window.



```
        address        tree                                     coverage objectcode 0%     50%     100 branches       ok    taken not taken   never    bytes      ok
 P:90000440--900009BD  ⊟ \coverage                               partial    98.293%                     92.307%      46.      3.       1.      2.    1406.   1382.
 P:90000440--9000044D    ⊞ BooleanAssignmentNotOp                     ok   100.000%                    100.000%       3.      0.       0.      0.      14.     14.
 P:9000044E--90000455    ⊞ BooleanAssignmentRelExpr                   ok   100.000%                          -        0.      0.       0.      0.       8.      8.
 P:90000456--90000463    ⊞ BooleanAssignmentRelExprTrans              ok   100.000%                    100.000%       1.      0.       0.      0.      14.     14.
 P:90000464--90000475    ⊞ BooleanExprCoupledTerms                    ok   100.000%                    100.000%       4.      0.       0.      0.      18.     18.
 P:90000476--90000485    ⊞ BooleanExprMixedOps                        ok   100.000%                    100.000%       3.      0.       0.      0.      16.     16.
 P:90000486--90000495    ⊞ BooleanExprSameOps                         ok   100.000%                    100.000%       3.      0.       0.      0.      16.     16.
 P:90000496--900004CF    ⊞ ComplexDoWhile                             ok   100.000%                    100.000%       5.      0.       0.      0.      58.     58.
 P:900004D0--900004FF    ⊞ ComplexFor                                 ok   100.000%                    100.000%       5.      0.       0.      0.      48.     48.
 P:90000500--90000527    ⊞ ComplexIf                                  ok   100.000%                    100.000%       4.      0.       0.      0.      40.     40.
 P:90000528--90000569    ⊞ ComplexWhile                               ok   100.000%                    100.000%       5.      0.       0.      0.      66.     66.
 P:9000056A--9000056F    ⊞ Identity                                   ok   100.000%                          -        0.      0.       0.      0.       6.      6.
 P:90000570--90000591    ⊞ MultiLine                             partial    58.823%                     41.666%       1.      2.       1.      2.      34.     20.
```

Details on the code coverage analysis itself are provided in the chapter **"Code Coverage Evaluation in TRACE32"**, page 101.

6. If more trace data is required, repeat step 3 and 4 until the desired level of coverage is obtained.

If the data recording and the code coverage analysis are executed by different teams, it is possible to save the collected trace data and process it at a later point in time. Please refer to the commands **Trace.SAVE** and **Trace.LOAD**.

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **"TRACE32 Merge and Report Tool"**, page 114.

Example Script

The entire process can be automated by creating a PRACTICE script. It is assumed that the preconditions listed in **“Best Practices for Trace Recording”**, page 64 are satisfied before running the script. In the example script default settings are commented out.

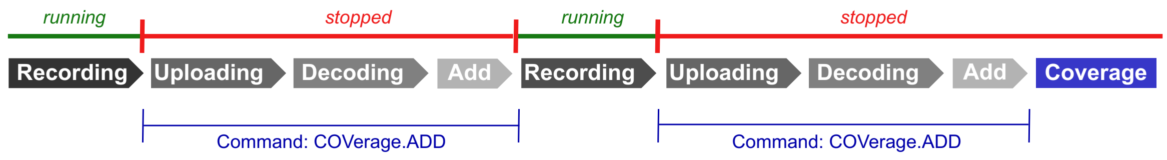
```
…
// Trace.METHOD as automatically selected by TRACE32
Trace.Mode Leash
// Trace.AutoArm ON
Trace.AutoInit ON
COVerage.RESet
// COVerage.METHOD INCremental
RePeaT 10.
(
    Go.direct
    WAIT !STATE.RUN()
    COVerage.ADD
)
COVerage.ListFunc

// export test result for later reuse
COVerage.EXPORT.JSONE coverage_data1
```

Summary

A characteristic feature of incremental code coverage is that the individual steps are executed one by one. Trace information is recorded while the program is running. After the program has been stopped, the command **COVerage.ADD** ensures that:

- the raw trace data is **uploaded** to the host computer
- the raw trace data is **decoded** to reconstruct the complete program flow
- the program flow is finally **added** to the code coverage system

This workflow is summarized in the diagram below.



Details about the code coverage analysis itself are provided in the chapter **“Code Coverage Evaluation in TRACE32”**, page 101.

Incremental Code Coverage in STREAM Mode

If a TRACE32 trace hardware tool such as PowerTrace is used it is possible to stream the trace data to a file on the host file system. Information about the general conditions for trace streaming can be found in the command description of the **Trace.Mode STREAM** command.

If the trace data is streamed to the host computer, longer recording times can be achieved. Incremental code coverage in STREAM mode supports all code coverage metrics supported by TRACE32.

In case of large amounts of trace data, processing may take a long time. TRACE32 provides two alternative methods to avoid this situation.

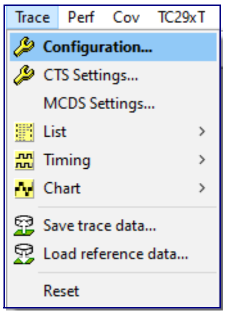
The first method is RTS, which is supported for all major architectures. RTS means that trace data is processed while being recorded and the code coverage results are displayed dynamically. Please see **“RTS Mode Code Coverage”**, page 84 for additional information.

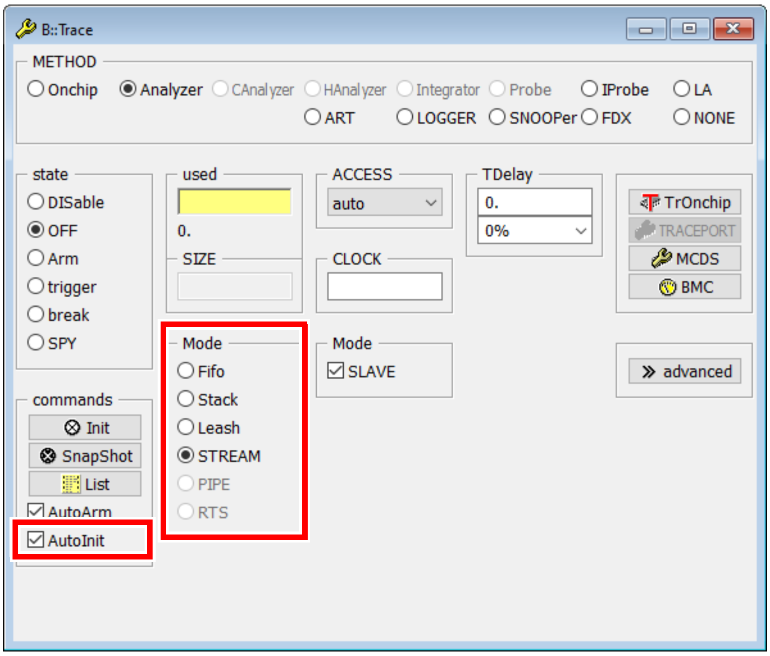
If RTS is not supported for your core architectures, then SPY Mode Code Coverage can be an alternative. Please see **“SPY Mode Code Coverage”**, page 90 for more details.

Data Collection

1. Set the trace to STREAM Mode either via the **Trace Configuration** window or via the **Trace.Mode STREAM** command.

2. Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to ensure that the trace buffer is always cleared before the trace recording is started.




B::Trace

METHOD: Onchip [✓ Analyzer] CAnalyzer HAnalyzer Integrator Probe IProbe LA ART LOGGER SNOOPer FDX NONE

state: DISable [✓ OFF] Arm trigger break SPY

used: 0.

ACCESS: auto

TDelay: 0. 0%

Mode: Fifo Stack Leash [✓ STREAM] PIPE RTS

Mode: ✓ SLAVE

✓ AutoArm

✓ AutoInit

3. TRACE32 by default opens a streaming file in the directory for temporary files (**OS.PresentTemporaryDirectory()**).

   The streaming file can be optionally set using the command **Trace.STREAMFILE**. It is recommended to use the fastest drive available on the host, ideally not the boot drive.

   ```
   Trace.STREAMFILE "d:\temp\mystream.t32"
   ```

4. The maximum size allowed for a streaming file can be optionally set with the help of the **Trace.STREAMFileLimit** command.

   ```
   ; limit the size of the streaming file to 5 GBytes
   Trace.STREAMFileLimit 5000000000.
   ```

   Please be aware, that the trace recording is stopped, when the size limit for the streaming file is reached.

5. Since code coverage does not need any timestamp information, please use the command **Trace.PortFilter MAX** to instruct TRACE32 to stream only the raw trace data. Further background information can be found in the chapter **“Disable Timestamps for Trace Streaming”**, page 65.

6. Start the program execution.

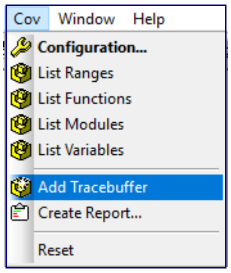
7. The program execution on the target must be stopped in order to perform the code coverage analysis.

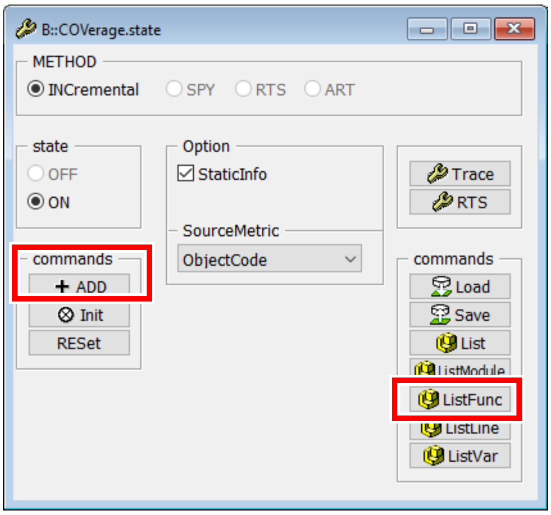
   - The user may manually stop the program execution.

   - A breakpoint may be used to stop the program execution.

   - With the help of a script, the program execution may be stopped after a specific period of time.

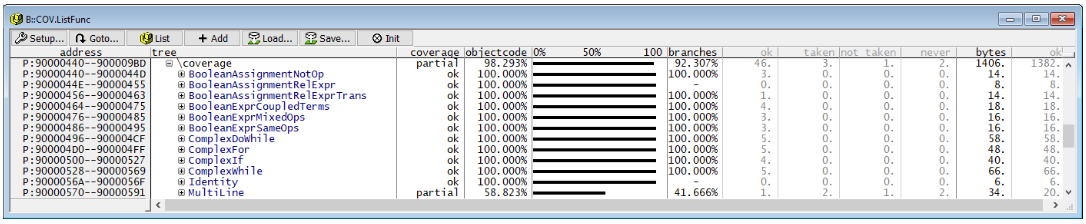
8. After the program execution has stopped, the trace data can be added to the coverage system with the **COVerage.ADD** command or by using the **+ADD** button in the **COVerage Configuration** window, or by selecting ‘**Add Tracebuffer**’ from the **Coverage** menu (shown in the image below).





METHOD: (● INCremental)
state: (● ON)
Option: ☑ StaticInfo
SourceMetric: ObjectCode

9. Intermediate results can be displayed by using the **ListFunc** button in the **COVerage Configuration** window.



address	tree	coverage	objectcode	0% 50% 100	branches	ok	taken	not taken	never	bytes	ok
P:90000440--900009BD	⊟ \coverage	partial	98.293%		92.307%	46.	3.	1.	2.	1406.	1382.
P:90000440--9000044D	⊞ BooleanAssignmentNotOp	ok	100.000%		100.000%	3.	0.	0.	0.	14.	14.
P:9000044E--90000455	⊞ BooleanAssignmentRelExpr	ok	100.000%		-	0.	0.	0.	0.	8.	8.
P:90000456--90000463	⊞ BooleanAssignmentRelExprTrans	ok	100.000%		100.000%	1.	0.	0.	0.	14.	14.
P:90000464--90000475	⊞ BooleanExprCoupledTerms	ok	100.000%		100.000%	4.	0.	0.	0.	18.	18.
P:90000476--90000485	⊞ BooleanExprMixedOps	ok	100.000%		100.000%	3.	0.	0.	0.	16.	16.
P:90000486--90000495	⊞ BooleanExprSameOps	ok	100.000%		100.000%	3.	0.	0.	0.	16.	16.
P:90000496--900004CF	⊞ ComplexDoWhile	ok	100.000%		100.000%	5.	0.	0.	0.	58.	58.
P:900004D0--900004FF	⊞ ComplexFor	ok	100.000%		100.000%	5.	0.	0.	0.	48.	48.
P:90000500--90000527	⊞ ComplexIf	ok	100.000%		100.000%	4.	0.	0.	0.	40.	40.
P:90000528--90000569	⊞ ComplexWhile	ok	100.000%		100.000%	5.	0.	0.	0.	66.	66.
P:9000056A--9000056F	⊞ Identity	ok	100.000%		-	0.	0.	0.	0.	6.	6.
P:90000570--90000591	⊞ MultiLine	partial	58.823%		41.666%	1.	2.	1.	2.	34.	20.

Details on the code coverage analysis itself are provided in the chapter **“Code Coverage Evaluation in TRACE32”**, page 101.

10. Steps 6 and 8 can be repeated until the desired level of coverage is obtained.

If the data is recorded at a test site and there is no time for evaluation, it is possible to save the collected raw trace data and process it at a later point in time. Please refer to the commands **Trace.STREAMSAVE** and **Trace.STREAMLOAD**.

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **“TRACE32 Merge and Report Tool”**, page 114.

Example Script

In this example script default settings are commented out. It is assumed that the preconditions listed in **"Best Practices for Trace Recording"**, page 64 are satisfied before running the script.

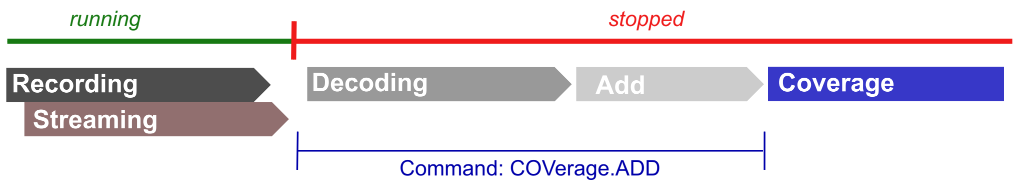
```
...
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer
// Trace.AutoArm ON
Trace.AutoInit ON

Trace.Mode STREAM
Trace.STREAMFile "D:\streamfile.t32"
Trace.STREAMFileLimit 5000000000.

Trace.PortFilter MAX

COVerage.RESet
// COVerage.METHOD INCremental

Go
WAIT 10.s
Break
COVerage.ADD
COVerage.ListFunc

// export test result for later reuse
COVerage.EXPORT.JSONE coverage_data1
```

Summary

The advantage of incremental code coverage with streaming is that larger amounts of trace data can be recorded in a single test run. However, before the recorded trace data can be processed, the program execution must be stopped. The command **COVerage.ADD** ensures that:

- the raw trace data is **decoded** to reconstruct the complete program flow
- the program flow is **added** to the code coverage system

This workflow is summarized in the diagram below.



Details about the code coverage analysis itself are provided in the chapter **"Code Coverage Evaluation in TRACE32"**, page 101.

RTS Mode Code Coverage

TRACE32 can process the trace data during recording. This operation mode of the trace is called RTS.

RTS is currently supported for the following processor architecture/trace protocols:

- Arm ETMv3, PTM and Arm ETMv4
- Nexus for MPC5xxx and QorIQ
- TriCore MCDS

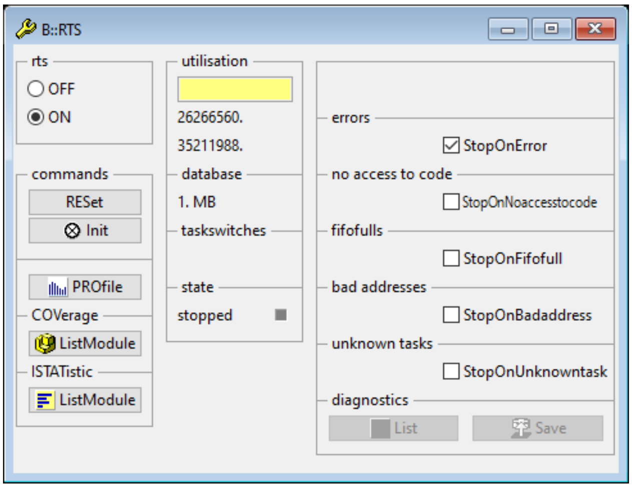
If RTS is not supported for your core architectures, then SPY mode code coverage could be an alternative. Please refer to **“SPY Mode Code Coverage”**, page 90.

RTS requires a TRACE32 trace hardware tool such as PowerTrace and streaming of the trace data to a file on the host file system has to work without issues. Information on the general conditions for trace streaming can be found in the command description of the **Trace.Mode STREAM** command.

RTS mode code coverage supports only the following code coverage metrics: statement coverage, function coverage, object code coverage and ocb decision coverage.

Data Collection

1. Switch the RTS system to ON in the **RTS.state** window or with the help of the **RTS.ON** command.



rts: OFF [✓ ON]

utilisation: 26266560. 35211988.

database: 1. MB

taskswitches

state: stopped

errors: [✓ StopOnError]

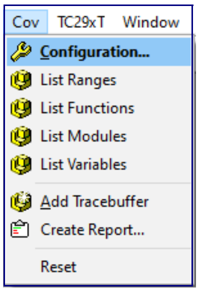
no access to code: StopOnNoaccesstocode

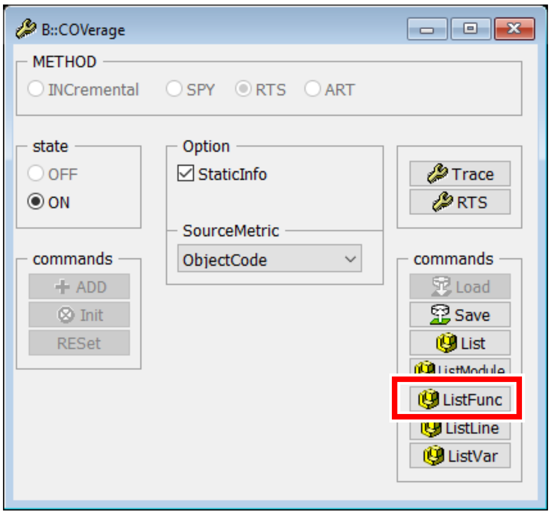
fifofulls: StopOnFifofull

bad addresses: StopOnBadaddress

unknown tasks: StopOnUnknowntask

2. Open a **COVerage.ListFunc** window by using the **ListFunc** button in the **COVerage Configuration** window or by using the command **COVerage.ListFunc**. Please be aware that trace data recorded in RTS mode are only processed by TRACE32 as long as one window in TRACE32 displays code coverage information.





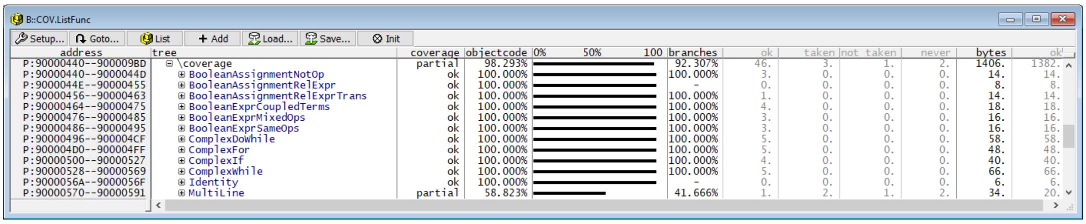
METHOD: ( ) INCremental ( ) SPY (●) RTS ( ) ART

state: ( ) OFF (●) ON

Option: ☑ StaticInfo

SourceMetric: ObjectCode

3. Start the program and observe the measured code coverage.



```
         address        tree                                 coverage objectcode 0%     50%     100 branches      ok    taken  not taken    never    bytes      ok
P:90000440--900009BD  ⊟ \coverage                            partial    98.293%                      92.307%     46.       3.        1.       2.    1406.    1382.
P:90000440--9000044D    ⊞ BooleanAssignmentNotOp                  ok   100.000%                     100.000%      3.       0.        0.       0.      14.      14.
P:9000044E--90000455    ⊞ BooleanAssignmentRelExpr                ok   100.000%                            -      0.       0.        0.       0.       8.       8.
P:90000456--90000463    ⊞ BooleanAssignmentRelExprTrans           ok   100.000%                     100.000%      1.       0.        0.       0.      14.      14.
P:90000464--90000475    ⊞ BooleanExprCoupledTerms                 ok   100.000%                     100.000%      4.       0.        0.       0.      18.      18.
P:90000476--90000485    ⊞ BooleanExprMixedOps                     ok   100.000%                     100.000%      3.       0.        0.       0.      16.      16.
P:90000486--90000495    ⊞ BooleanExprSameOps                      ok   100.000%                     100.000%      3.       0.        0.       0.      16.      16.
P:90000496--900004CF    ⊞ ComplexDoWhile                          ok   100.000%                     100.000%      5.       0.        0.       0.      58.      58.
P:900004D0--900004FF    ⊞ ComplexFor                              ok   100.000%                     100.000%      5.       0.        0.       0.      48.      48.
P:90000500--90000527    ⊞ ComplexIf                               ok   100.000%                     100.000%      4.       0.        0.       0.      40.      40.
P:90000528--90000569    ⊞ ComplexWhile                            ok   100.000%                     100.000%      5.       0.        0.       0.      66.      66.
P:9000056A--9000056F    ⊞ Identity                                ok   100.000%                            -      0.       0.        0.       0.       6.       6.
P:90000570--90000591    ⊞ MultiLine                          partial    58.823%                      41.666%      1.       2.        1.       2.      34.      20.
```

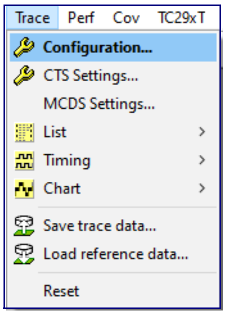
Details on the code coverage analysis itself are provided in the chapter **"Code Coverage Evaluation in TRACE32"**, page 101.

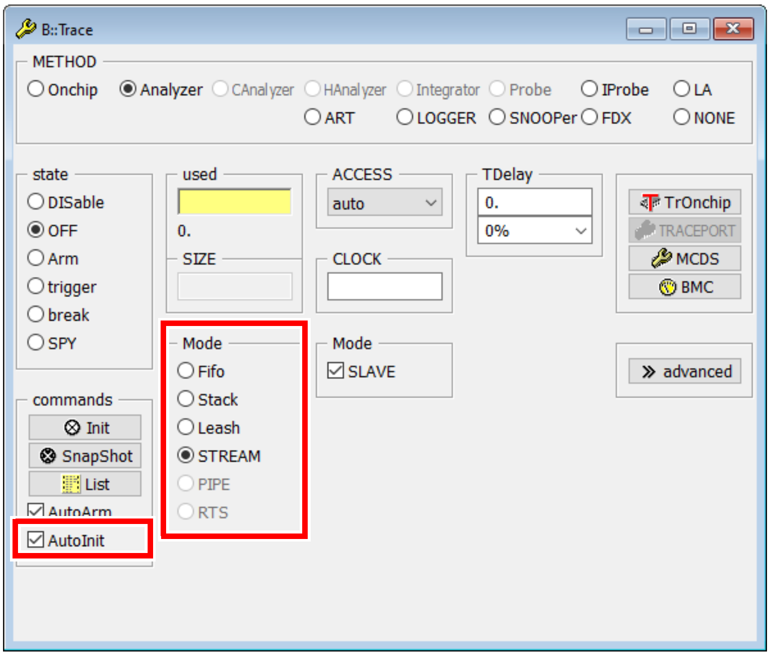
4. Stop the program exucution when your tests are completed.

RTS discards the trace data after it is processed by default. If you want to keep the trace data for additional verification tasks perform these configuration steps before setting up RTS mode code coverage as described above.

1. Set the trace to **STREAM** mode either via the **Trace Configuration** window or the **Trace.Mode STREAM** command.

2. Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to ensure that the trace buffer is always cleared before the trace recording is started.





3. TRACE32 by default opens a streaming file in the directory for temporary files (**OS.PresentTemporaryDirectory()**).

   The streaming file can be optionally set by using the command **Trace.STREAMFILE**. It is recommended to use the fastest drive available on the host, ideally not the boot drive.

```
Trace.STREAMFILE "d:\temp\mystream.t32"
```

4. The maximum size allowed for a streaming file can be optionally set with the help of the command **Trace.STREAMFileLimit**.

```
; limit the size of the streaming file to 5 GBytes
Trace.STREAMFileLimit 5000000000.
```

   Please be aware, that the trace recording is stopped, when the size limit for the streaming file is reached.

5. Since code coverage does not need any timestamp information, please use the command **Trace.PortFilter MAX** to instruct TRACE32 to stream only the raw trace data. Further background information can be found in the chapter **"Disable Timestamps for Trace Streaming"**, page 65.

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **“TRACE32 Merge and Report Tool”**, page 114.

Example Scripts

This example script discards the trace data after it is processed; default settings are commented out. It is assumed that the preconditions listed in **“Best Practices for Trace Recording”**, page 64 are satisfied before running the script.

```
…
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer

; Set breakpoint to end of test run
Break.Set vTestComplete

COVerage.RESet
RTS.ON
COVerage.ListFunc

Go
WAIT !STATE.RUN()
…

// export test result for later reuse
COVerage.EXPORT.JSONE coverage_data1
```

This example script saves the trace data to a streaming file; default settings are commented out.

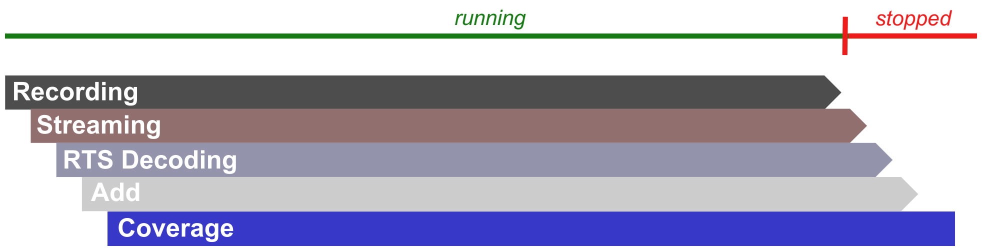
```
…
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer
// Trace.AutoArm ON
Trace.AutoInit ON

Trace.Mode STREAM
Trace.STREAMFile "D:\streamfile.t32"
Trace.STREAMFileLimit 5000000000.

Trace.PortFilter MAX

; Set breakpoint to end of test run
Break.Set vTestComplete

COVerage.RESet
RTS.ON
COVerage.ListFunc

Go
WAIT !STATE.RUN()
Trace.List

// export test result for later reuse
COVerage.EXPORT.JSONE coverage_data1
```

Summary

The big advantage of RTS mode code coverage is that all necessary steps run in parallel. Large amounts of trace data can be processed quickly. Code coverage measurement becomes available immediately.

The following steps are performed concurrently with trace data collection:

- The raw trace data are **streamed** to the host computer, optionally it can be saved to the streaming file
- The raw trace data are **decoded** to reconstruct the program flow
- The program flow is **added** to the code coverage system
- The code **coverage** results are updated



Details about the code coverage analysis itself are provided in the chapter **"Code Coverage Evaluation in TRACE32"**, page 101.

SPY Mode Code Coverage

TRACE32 supports processing of trace data while being recorded for all architectures:

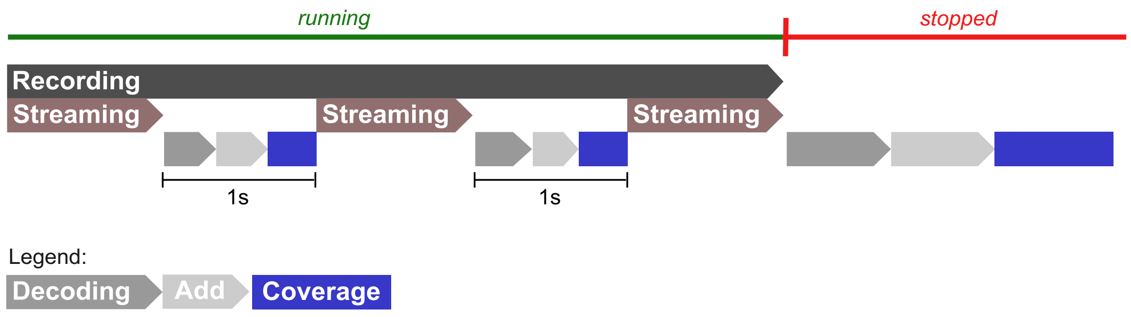
- TRACE32 trace hardware tool such as PowerTrace is required
- Streaming of the trace data to a file on the host file system is working without issues

  Information about the general conditions for trace streaming can be found in the description of the command **Trace.Mode STREAM**.

SPY mode code coverage achieves lower processing speeds than RTS mode code coverage, but supports all code coverage metrics supported by TRACE32.

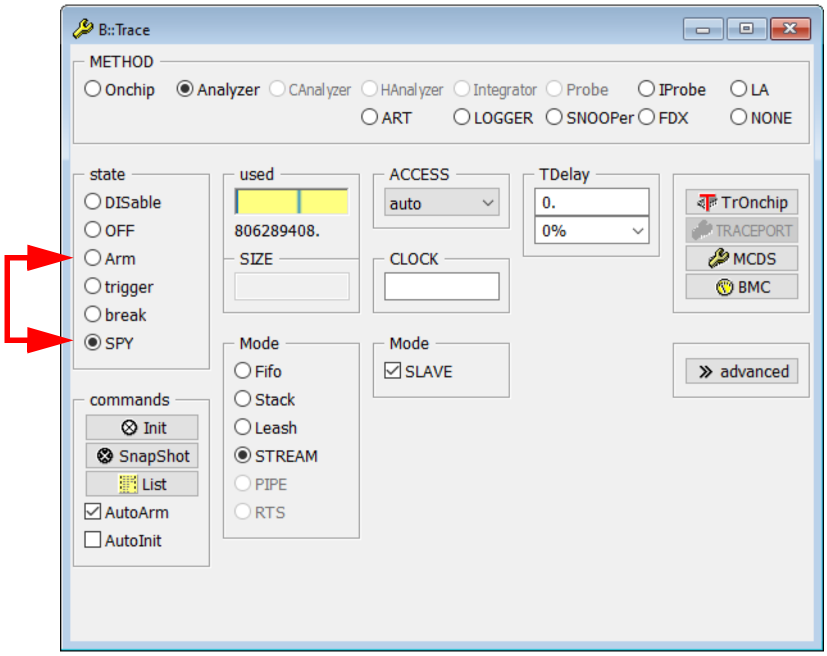
Operation States

For SPY mode code coverage, trace streaming is periodically suspended in order to decode the raw trace data and to process it for code coverage. Please be aware that TRACE32 does not suspend trace streaming if the trace memory of the TRACE32 trace tool, that operates as a large FIFO, is filled more the 50%.



TRACE32 indicates the current trace state by changing between Arm and SPY.

- **Arm:** Trace data is being recorded and streamed to the streaming file on the host computer.
- **SPY:** Trace data is being recorded and the content of the streaming file is processed for code coverage.



B::Trace

METHOD: ○ Onchip [● Analyzer] ○ CAnalyzer ○ HAnalyzer ○ Integrator ○ Probe ○ IProbe ○ LA ○ ART ○ LOGGER ○ SNOOPer ○ FDX ○ NONE

state: ○ DISable ○ OFF ○ Arm ○ trigger ○ break [● SPY]

used: 806289408.

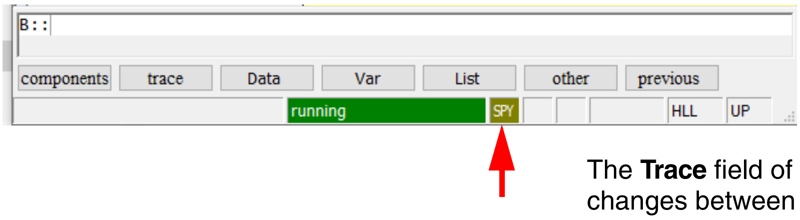
ACCESS: auto

TDelay: 0. 0%

Mode: ○ Fifo ○ Stack ○ Leash [● STREAM] ○ PIPE ○ RTS

Mode: [✓ SLAVE]

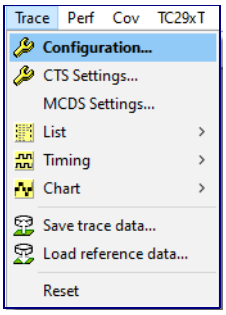
commands: [✓ AutoArm] ☐ AutoInit

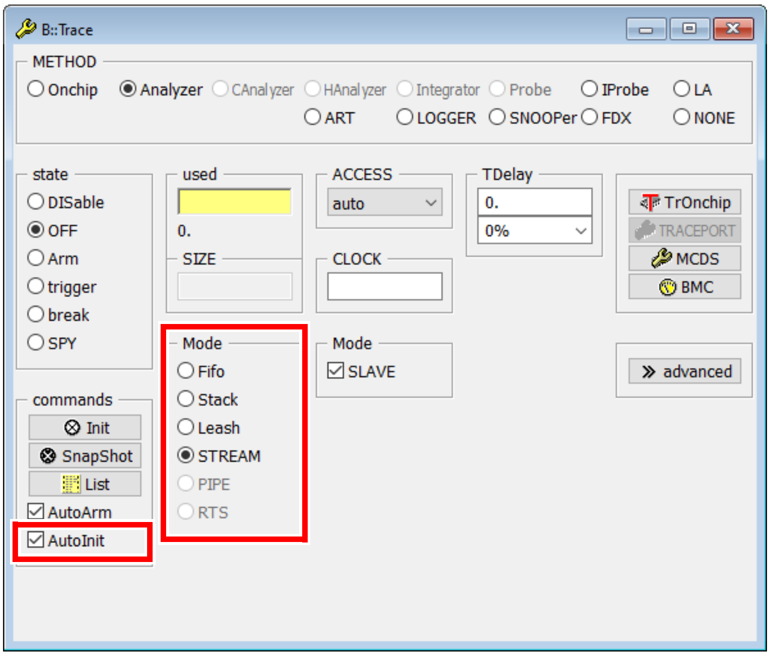


The **Trace** field of the TRACE32 state line changes between Arm and SPY

1. Set the trace mode to **STREAM** either via the **Trace configuration** window or via the **Trace.Mode STREAM** command.

2. Enable the **AutoInit** checkbox or use the command **Trace. ON** to ensure that the trace buffer is always cleared before the trace recording is started.





METHOD: Analyzer
state: OFF
used: 0.
ACCESS: auto
TDelay: 0. / 0%
Mode: STREAM
SLAVE: ✓
AutoArm: ✓
AutoInit: ✓

3. TRACE32 by default opens a streaming file in the directory for temporary files (**OS.PresentTemporaryDirectory()**).

   The streaming file can be optionally set using the command **Trace.STREAMFILE**. It is recommended to use the fastest drive available on the host, ideally not the boot drive.

```
Trace.STREAMFILE "d:\temp\mystream.t32"
```

4. The maximum size allowed for a streaming file can be optionally set with the help of the command **Trace.STREAMFileLimit**.

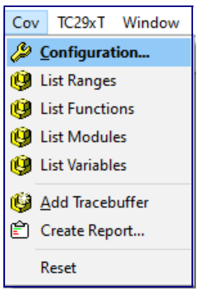
```
; limit the size of the streaming file to 5 GBytes
Trace.STREAMFileLimit 5000000000.
```

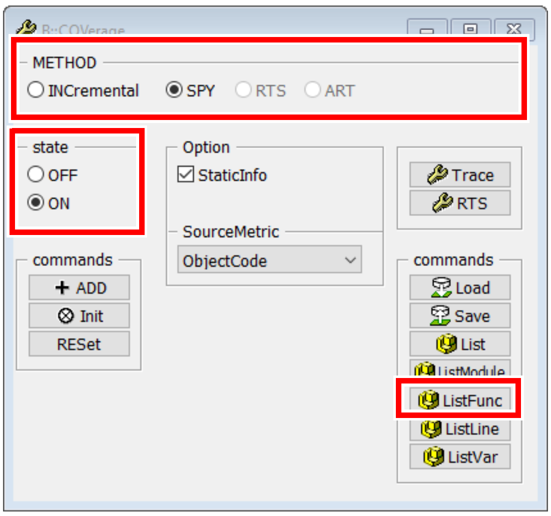
   Please be aware, that the trace recording is stopped, when the size limit for the streaming file is reached.

5. Since code coverage does not need any timestamp information, please use the command **Trace.PortFilter MAX** to instruct TRACE32 to stream only the raw trace data. Further background information can be found in the chapter **“Disable Timestamps for Trace Streaming”**, page 65.

6. Set the coverage method to SPY by using the command **COVerage.METHOD SPY** or by selecting **SPY** in the **COVerage configuration** window.

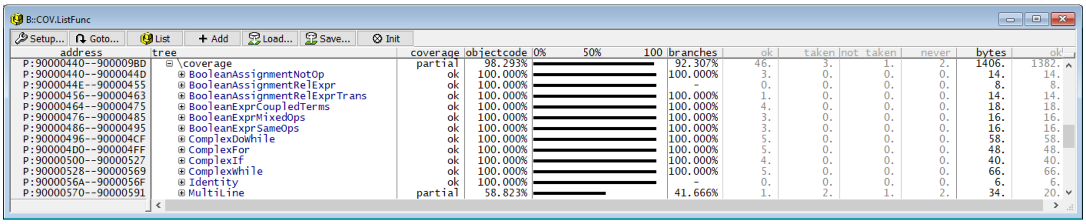
7. Enable **SPY** mode code coverage by the command **COVerage.ON** or by selecting the **ON** radio button in the state field.





8. Open a **COVerage.ListFunc** window by using the **ListFunc** button in the **COVerage configuration** window or by using the command **COVerage.ListFunc**. Please be aware that trace data recorded in SPY mode code coverage is only periodically processed by TRACE32, if at least one window in TRACE32 displays code coverage information.

9. Start the program and observe directly the results of the code coverage.



B::COV.ListFunc

address	tree	coverage	objectcode	0% 50% 100	branches	ok	taken	not taken	never	bytes	ok
P:90000440--900009BD	\coverage	partial	98.293%		92.307%	46.	3.	1.	2.	1406.	1382.
P:90000440--9000044D	BooleanAssignmentNotOp	ok	100.000%		100.000%	3.	0.	0.	0.	14.	14.
P:9000044E--90000455	BooleanAssignmentRelExpr	ok	100.000%		-	0.	0.	0.	0.	8.	8.
P:90000456--90000463	BooleanAssignmentRelExprTrans	ok	100.000%		100.000%	1.	0.	0.	0.	14.	14.
P:90000464--90000475	BooleanExprCoupledTerms	ok	100.000%		100.000%	4.	0.	0.	0.	18.	18.
P:90000476--90000485	BooleanExprMixedOps	ok	100.000%		100.000%	3.	0.	0.	0.	16.	16.
P:90000486--90000495	BooleanExprSameOps	ok	100.000%		100.000%	3.	0.	0.	0.	16.	16.
P:90000496--900004CF	ComplexDoWhile	ok	100.000%		100.000%	5.	0.	0.	0.	58.	58.
P:900004D0--900004FF	ComplexFor	ok	100.000%		100.000%	5.	0.	0.	0.	48.	48.
P:90000500--90000527	ComplexIf	ok	100.000%		100.000%	4.	0.	0.	0.	40.	40.
P:90000528--90000569	ComplexWhile	ok	100.000%		100.000%	5.	0.	0.	0.	66.	66.
P:9000056A--9000056F	Identity	ok	100.000%		-	0.	0.	0.	0.	6.	6.
P:90000570--90000591	MultiLine	partial	58.823%		41.666%	1.	2.	1.	2.	34.	20.

Details on the code coverage analysis itself are provided in the chapter **"Code Coverage Evaluation in TRACE32"**, page 101.

10. Stop the program execution when your tests have completed.

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **"TRACE32 Merge and Report Tool"**, page 114.

Example Script

In the script the default settings are commented out. It is assumed that the preconditions listed in **"Best Practices for Trace Recording"**, page 64 are satisfied before running the script.

```
…
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer
// Trace.AutoArm ON
Trace.AutoInit ON

Trace.Mode STREAM
Trace.STREAMFile "D:\streamfile.t32"
Trace.STREAMFileLimit 5000000000.

Trace.PortFilter MAX

; Set breakpoint to end of test run
Break.Set vTestComplete

COVerage.RESet
COVerage.METHOD SPY
COVerage.ON
COVerage.ListFunc

Go
WAIT !STATE.RUN()
Trace.List
…
// export test result for later reuse
COVerage.EXPORT.JSONE coverage_data1
```

Summary

SPY Mode Code Coverage can process trace data concurrently while recording. However, it does not achieve the same processing speeds as RTS mode code coverage.

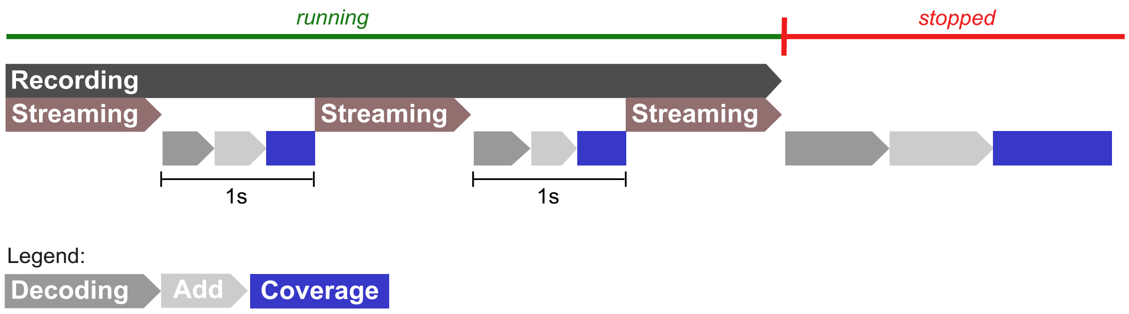
The following steps are involved:

- Trace information is **recorded** continuously.
- The raw trace data is **streamed** to a file on the host computer, but the streaming is periodically suspended:

  - to **decode** the raw trace data to reconstruct the program flow

  - to **add** the program flow to the code coverage system

  - to update code **coverage** results



Details about the code coverage analysis itself are provided in the chapter **"Code Coverage Evaluation in TRACE32"**, page 101.

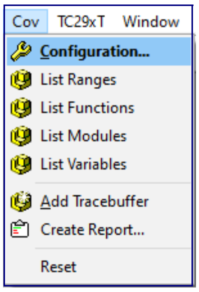
Code Coverage with Virtual Targets

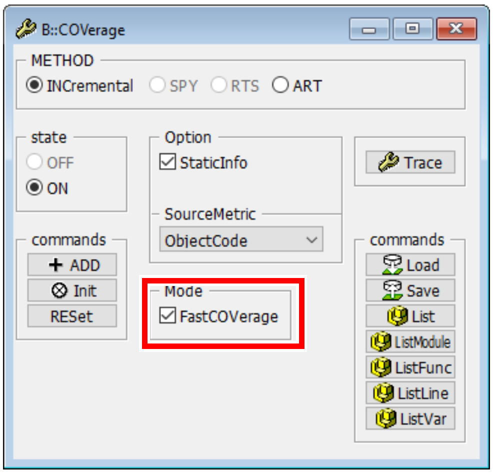
Tracing the program execution on a virtual target slows down its performance. To minimize this impact, Lauterbach works closely together with manufacturers such as Synopsys. The basic idea is that some parts of the code coverage processing are offloaded to the virtual target. This information is uploaded to the TRACE32 code coverage system with the command **COVerage.ADD** after the program execution has been stopped. The **MCD interface** comes with built-in support for this.

To use this feature the following conditions must be met:

- **`PBI=MCD`** must be specified in the TRACE32 configuration file, usually **`~~/config.t32`**.
- The Virtual Target must support program address tagging.

  COVerage.Mode FastCOVerage ON must be set. If the Virtual Target does not support program address tagging, TRACE32 will display the error message "function not implemented".





METHOD: [✓ INCremental] SPY RTS ART
state: OFF [✓ ON]
Option: [✓ StaticInfo]
SourceMetric: ObjectCode
Mode: [✓ FastCOVerage]

The program addressed tagged in the virtual target can be used for:

- Object code coverage (see **"Object Code Coverage Evaluation in TRACE32"**, page 101)
- Decision coverage (ocb) (see **"Object Code Based (ocb) Decision Coverage Evaluation"**, page 106)

An example script might look like this:

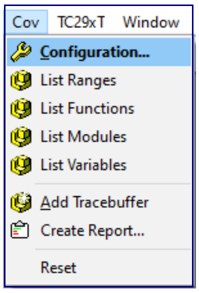
```
COVerage.RESet
COVerage.METHOD INCremental
COVerage.Mode FastCOVerage ON

Go

; Use a breakpoint or time-out to control length of runtime

Break

COVerage.Add

COVerage.ListFunc
```

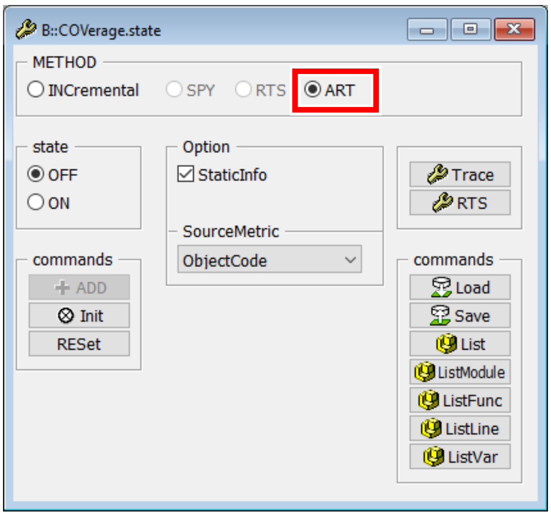
Details about the code coverage analysis itself are provided in the chapter **“Code Coverage Evaluation in TRACE32”**, page 101.

ART Mode Code Coverage

ART is an acronym for Advanced Register Trace. The **ART** trace operates by single stepping on assembler level. After each step, the contents of the CPU registers are uploaded to TRACE32 and stored in a similar fashion as a program flow trace.

This pseudo-trace data can be used for code coverage. This is not supported for all processor architectures. The **Coverage.METHOD ART** can only be selected if supported. Please be aware that ART has a significant impact on the real-time performance of the target. Each step takes 5 to 10 ms.




B::COVerage.state

METHOD: ○ INCremental ○ SPY ○ RTS [✓ ART]

state: [✓ OFF] ○ ON

Option: [✓ StaticInfo]

SourceMetric: ObjectCode

Trace data recorded with ART can be used for:

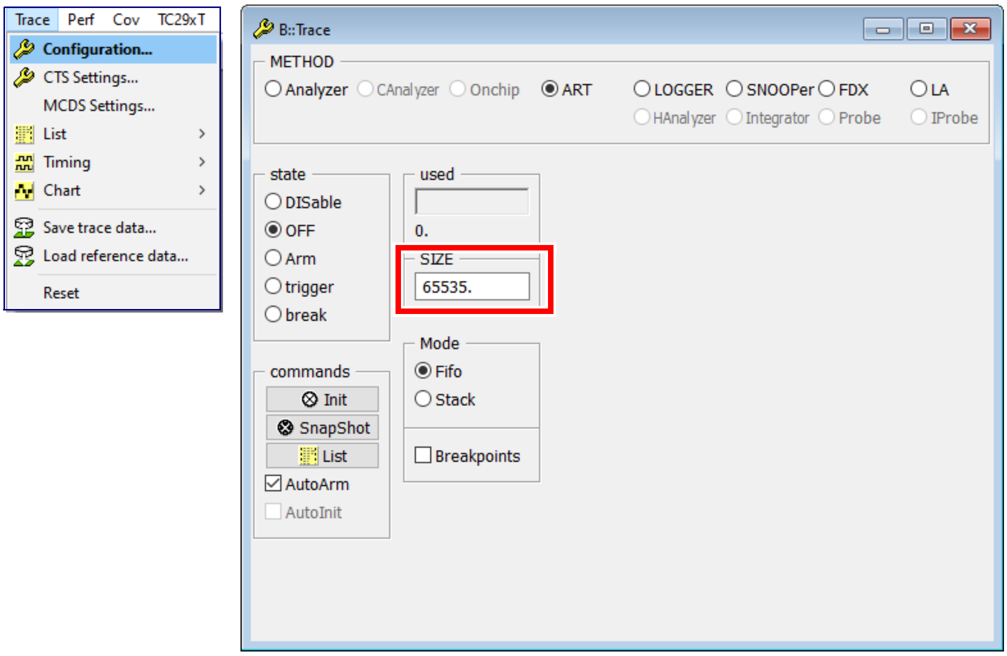
- Object code coverage (see **"Object Code Coverage Evaluation in TRACE32"**, page 101)
- Decision coverage (ocb) (see **"Object Code Based (ocb) Decision Coverage Evaluation"**, page 106)

Where possible, it is recommended to use the TRACE32 Instruction Set Simulator with **Trace.METHOD Analyzer** instead of ART. This has a better performance and supports all code coverage metrics.

The TRACE32 Instruction Set Simulator simulates the instruction set, but does not model timing characteristics and peripherals. However, the simulator provides a bus trace so that code coverage is easy to perform. For details on how to start the TRACE32 Instruction Set Simulator refer to **"Section PBI=<driver>"** in TRACE32 Installation Guide, page 48 (installation.pdf).

Before you start do not forget to switch debugging to mixed or assembler mode by using the **Mode.Asm** or **Mode.Mix** commands.

1. Select **Trace.METHOD ART** in the **Trace configuration** window.
2. Set the size of the ART buffer, using either the command **ART.SIZE** *<n>* or by entering the value in the **SIZE** field of the **Trace configuration** window.



METHOD: Analyzer CAnalyzer Onchip [✓ ART] LOGGER SNOOPer FDX LA HAnalyzer Integrator Probe IProbe

state: DISable [✓ OFF] Arm trigger break

used: 0.

SIZE: 65535.

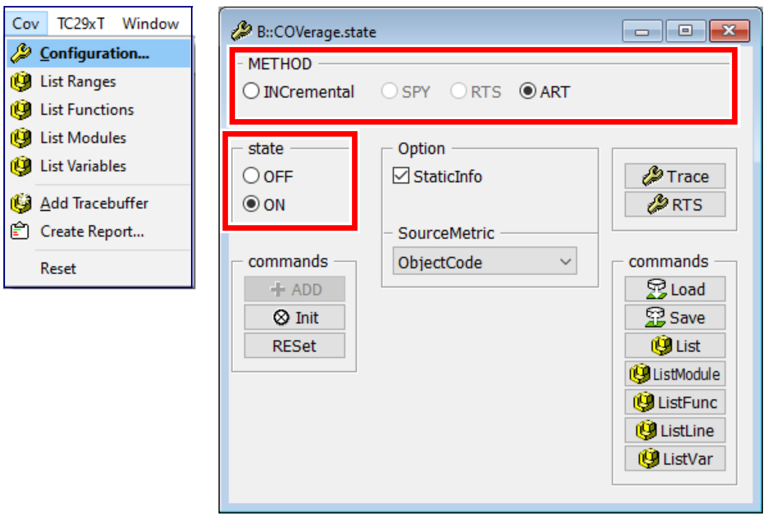
Mode: [✓ Fifo] Stack

☐ Breakpoints

☑ AutoArm

☐ AutoInit

3. Set **COVerage.METHOD ART** in the **COVerage configuration** window.
4. Enable ART code coverage with **COVerage.ON**.



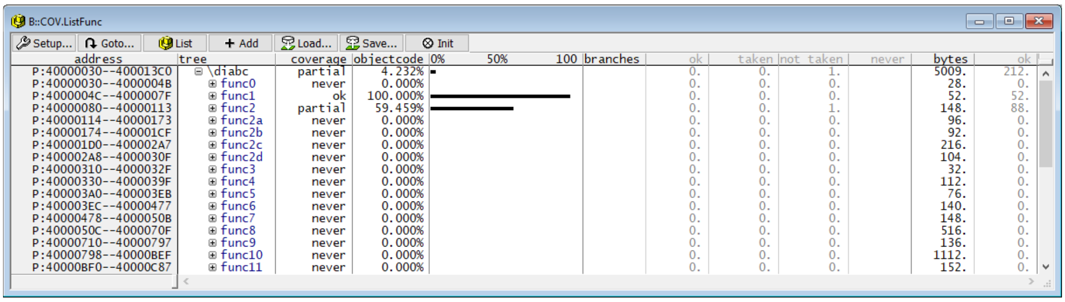
METHOD: INCremental SPY RTS [✓ ART]

state: OFF [✓ ON]

Option: ☑ StaticInfo

SourceMetric: ObjectCode

5. Open a **COVerage.ListFunc** window, single step the target and observe the result.



address	tree	coverage	objectcode	0% 50% 100	branches	ok	taken	not taken	never	bytes	ok
P:40000030--400013C0	\diabc	partial	4.232%			0.	0.	1.		5009.	212.
P:40000030--4000004B	func0	never	0.000%			0.	0.	0.		28.	0.
P:4000004C--4000007F	func1	ok	100.000%			0.	0.	0.		52.	52.
P:40000080--40000113	func2	partial	59.459%			0.	0.	1.		148.	88.
P:40000114--40000173	func2a	never	0.000%			0.	0.	0.		96.	0.
P:40000174--400001CF	func2b	never	0.000%			0.	0.	0.		92.	0.
P:400001D0--400002A7	func2c	never	0.000%			0.	0.	0.		216.	0.
P:400002A8--4000030F	func2d	never	0.000%			0.	0.	0.		104.	0.
P:40000310--4000032F	func3	never	0.000%			0.	0.	0.		32.	0.
P:40000330--4000039F	func4	never	0.000%			0.	0.	0.		112.	0.
P:400003A0--400003EB	func5	never	0.000%			0.	0.	0.		76.	0.
P:400003EC--40000477	func6	never	0.000%			0.	0.	0.		140.	0.
P:40000478--4000050B	func7	never	0.000%			0.	0.	0.		148.	0.
P:4000050C--4000070F	func8	never	0.000%			0.	0.	0.		516.	0.
P:40000710--40000797	func9	never	0.000%			0.	0.	0.		136.	0.
P:40000798--40000BEF	func10	never	0.000%			0.	0.	0.		1112.	0.
P:40000BF0--40000C87	func11	never	0.000%			0.	0.	0.		152.	0.

Details about the code coverage analysis itself are provided in the chapter **"Code Coverage Evaluation in TRACE32"**, page 101.

Example Script

A simple example is shown below.

```
Mode.Mixed

Trace.RESet
Trace.METHOD ART
Trace.SIZE 65535.        ; Set the size of the ART buffer

COVerage.RESet
COVerage.METHOD ART
COVerage.ON

Step 65534.            ; Single step on assembler level to capture data
COVerage.ListFunc      ; Open a Window to see results
```

Code Coverage Evaluation in TRACE32

Only two address-based code coverage metrics—Object Code Coverage and the outdated Object Code-Based (OCB) Decision Coverage — have to be evaluated in TRACE32. All other code coverage metrics are preferably evaluated in a web browser.

Object Code Coverage Evaluation in TRACE32

Object code coverage: Object code coverage ensures that each object code instruction was executed at least once and all conditional instructions (e.g. conditional branches) have evaluated to both true and false.

There are two tagging schemes:

- **ok | only exec | not exec | never**

  This is the tagging scheme for all trace protocols that provide details on the execution of conditional branches and conditional instructions. Refer also to the **COVerage.INFO** command. Currently, this tagging scheme is used only for Arm/Cortex cores with Arm-ETMv1 or Arm-ETMv3 protocols, as well as Arm-ETMv4 with **ETM.COND ON**.

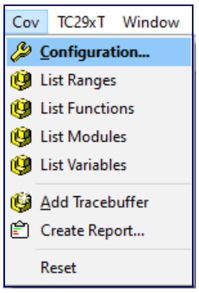
- **ok | taken | not taken | never**

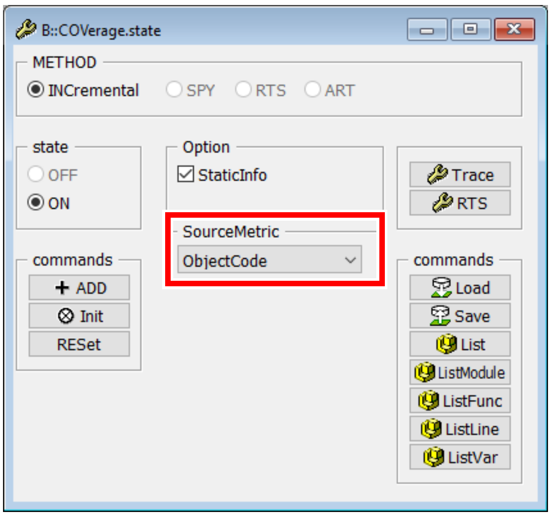
  This tagging scheme is used by all trace protocols that provide details solely on the execution of conditional branches. It is currently applied by most core architectures.

For details refer to **“Appendix G: Object Code Coverage Tags in Detail”**, page 137.

Evaluation

If you want to use the trace data stored in the code coverage system for object code coverage, select the SourceMetric **ObjectCode** in the **COVerage configuration window** or use the command **COVerage.Option SourceMetric** ObjectCode.





The following commands show a tabular analysis:

COVerage.ListModule

COVerage.ListFunc

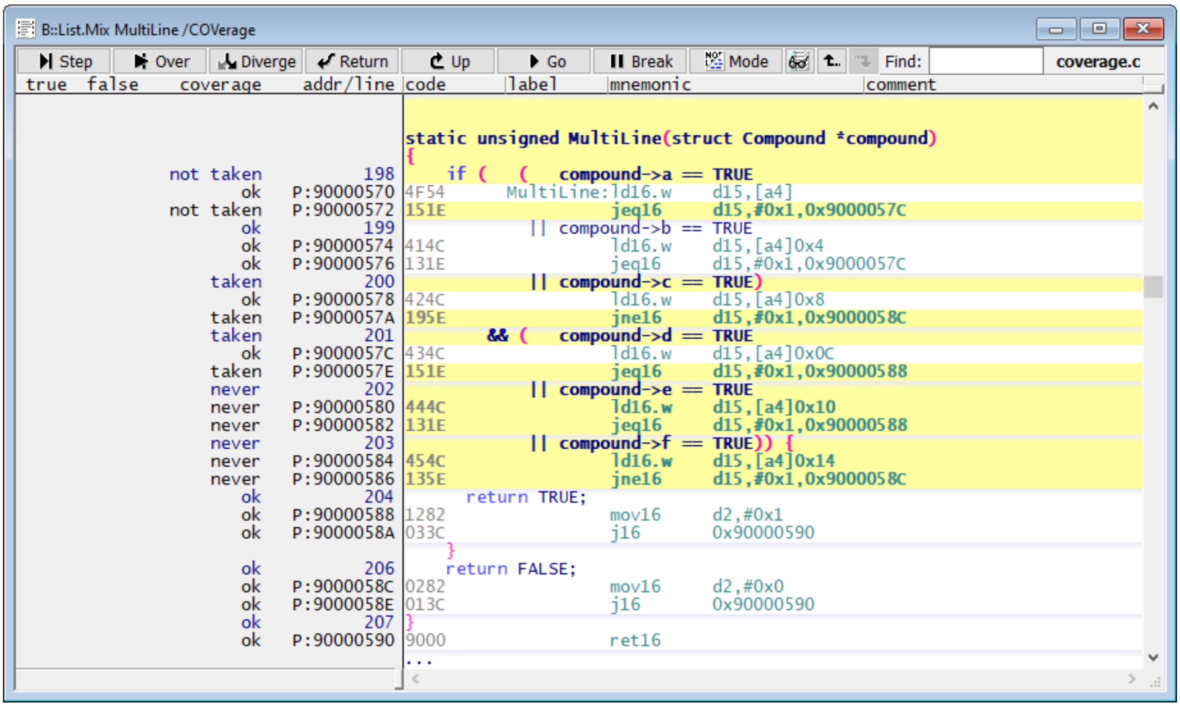
COVerage.ListLine

The following command shows the tagging on source and object code level.

List.Mix /COVerage

This TRACE32 command displays the object code tagging for the function *MultiLine*:

```
List.Mix MultiLine /COVerage
```



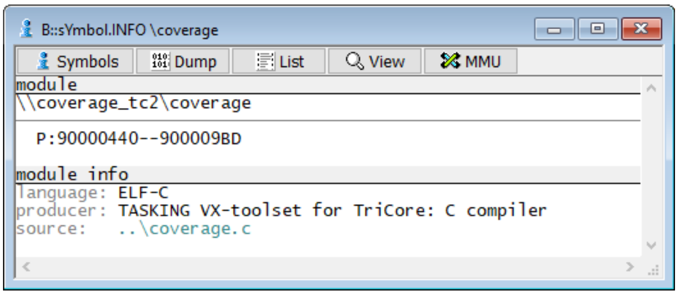
```
true  false     coverage     addr/line code    label     mnemonic          comment

                                               static unsigned MultiLine(struct Compound *compound)
                                               {
                     not taken         198        if (    (     compound->a == TRUE
                            ok  P:90000570 4F54       MultiLine:ld16.w     d15,[a4]
                     not taken  P:90000572 151E                 jeq16      d15,#0x1,0x9000057C
                            ok         199                   || compound->b == TRUE
                            ok  P:90000574 414C                 ld16.w     d15,[a4]0x4
                            ok  P:90000576 131E                 jeq16      d15,#0x1,0x9000057C
                         taken         200                   || compound->c == TRUE)
                            ok  P:90000578 424C                 ld16.w     d15,[a4]0x8
                         taken  P:9000057A 195E                 jne16      d15,#0x1,0x9000058C
                         taken         201               && (   compound->d == TRUE
                            ok  P:9000057C 434C                 ld16.w     d15,[a4]0x0C
                         taken  P:9000057E 151E                 jeq16      d15,#0x1,0x90000588
                         never         202                   || compound->e == TRUE
                         never  P:90000580 444C                 ld16.w     d15,[a4]0x10
                         never  P:90000582 131E                 jeq16      d15,#0x1,0x90000588
                         never         203                   || compound->f == TRUE)) {
                         never  P:90000584 454C                 ld16.w     d15,[a4]0x14
                         never  P:90000586 135E                 jne16      d15,#0x1,0x9000058C
                            ok         204            return TRUE;
                            ok  P:90000588 1282                 mov16      d2,#0x1
                            ok  P:9000058A 033C                 j16        0x90000590
                                                  }
                            ok         206          return FALSE;
                            ok  P:9000058C 0282                 mov16      d2,#0x0
                            ok  P:9000058E 013C                 j16        0x90000590
                            ok         207     }
                            ok  P:90000590 9000                 ret16
                                               ...
```

The screenshot on the previous page was taken with the Infineon TriCore™ debugger. Its instruction set contains no conditional instructions beyond conditional branches. Thus the object code is tagged as follows:

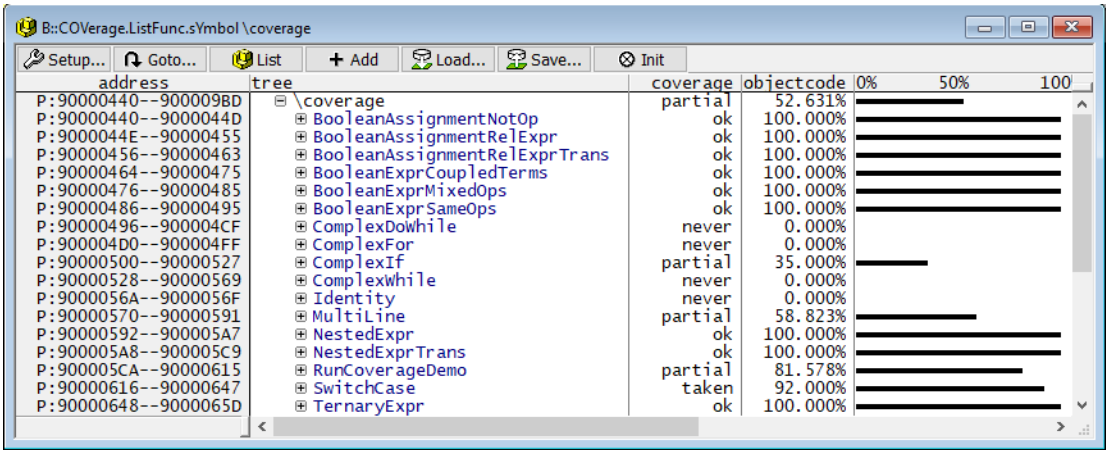
ok	The object code instruction is fully covered. If the object code is a conditional branch it is tagged with ok if the conditional branch has be at least once *taken* and *not taken*. All other object code instructions are tagged with ok if they have been executed at least once.
never	The object code instruction has never been executed.
taken	If the object code is a conditional branch it is tagged with *taken* if the conditional branch has be at least once *taken*, but never *not taken*.
not taken	If the object code is a conditional branch it is tagged with *not taken* if the conditional branch has be at least once *not taken*, but never *taken*.

This TRACE32 command displays a tabular analysis of all functions of the module "coverage". A module usually corresponds to a source code file.

```
COVerage.ListFunc.sYmbol \coverage
```

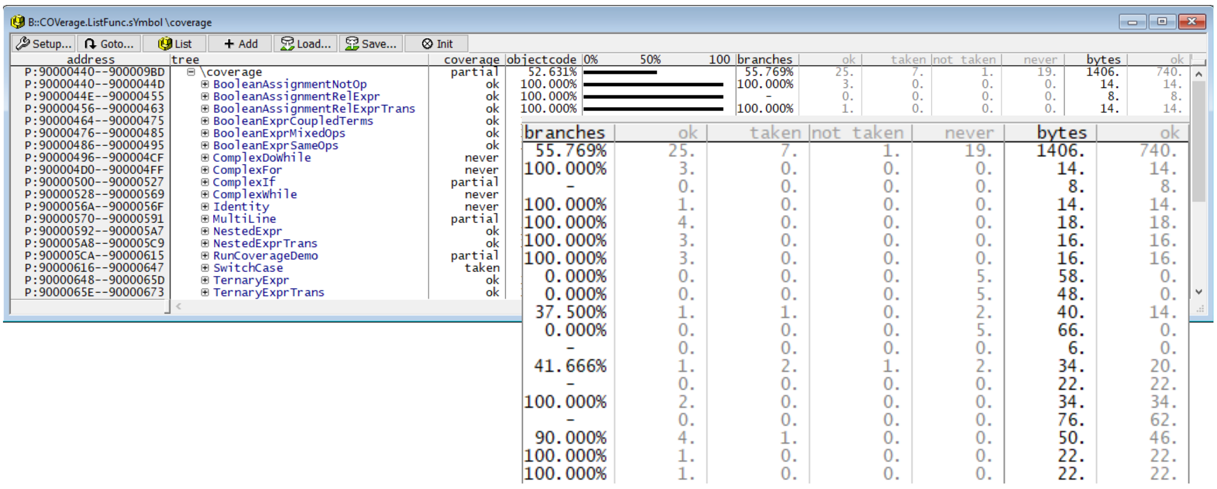


```
module
\\coverage_tc2\coverage

  P:90000440--900009BD

module info
language: ELF-C
producer: TASKING VX-toolset for TriCore: C compiler
source:   ..\coverage.c
```



```
       address          tree                                  coverage objectcode 0%      50%      100
  P:90000440--900009BD   ⊟ \coverage                           partial    52.631%
  P:90000440--9000044D     ⊞ BooleanAssignmentNotOp                 ok   100.000%
  P:9000044E--90000455     ⊞ BooleanAssignmentRelExpr               ok   100.000%
  P:90000456--90000463     ⊞ BooleanAssignmentRelExprTrans          ok   100.000%
  P:90000464--90000475     ⊞ BooleanExprCoupledTerms                ok   100.000%
  P:90000476--90000485     ⊞ BooleanExprMixedOps                    ok   100.000%
  P:90000486--90000495     ⊞ BooleanExprSameOps                     ok   100.000%
  P:90000496--900004CF     ⊞ ComplexDoWhile                      never     0.000%
  P:900004D0--900004FF     ⊞ ComplexFor                          never     0.000%
  P:90000500--90000527     ⊞ ComplexIf                         partial    35.000%
  P:90000528--90000569     ⊞ ComplexWhile                        never     0.000%
  P:9000056A--9000056F     ⊞ Identity                            never     0.000%
  P:90000570--90000591     ⊞ MultiLine                         partial    58.823%
  P:90000592--900005A7     ⊞ NestedExpr                             ok   100.000%
  P:900005A8--900005C9     ⊞ NestedExprTrans                        ok   100.000%
  P:900005CA--90000615     ⊞ RunCoverageDemo                   partial    81.578%
  P:90000616--90000647     ⊞ SwitchCase                          taken    92.000%
  P:90000648--9000065D     ⊞ TernaryExpr                            ok   100.000%
```

Further details are displayed if you open the window in its full size:



B::COVerage.ListFunc.sYmbol \coverage

address	tree	coverage	objectcode	branches	ok	taken	not taken	never	bytes	ok
P:90000440--900009BD	\coverage	partial	52.631%	55.769%	25.	7.	1.	19.	1406.	740.
P:90000440--9000044D	BooleanAssignmentNotOp	ok	100.000%	100.000%	3.	0.	0.	0.	14.	14.
P:9000044E--90000455	BooleanAssignmentRelExpr	ok	100.000%	-	0.	0.	0.	0.	8.	8.
P:90000456--90000463	BooleanAssignmentRelExprTrans	ok	100.000%	100.000%	1.	0.	0.	0.	14.	14.
P:90000464--90000475	BooleanExprCoupledTerms	ok		100.000%	4.	0.	0.	0.	18.	18.
P:90000476--90000485	BooleanExprMixedOps	ok		100.000%	3.	0.	0.	0.	16.	16.
P:90000486--90000495	BooleanExprSameOps	ok		100.000%	3.	0.	0.	0.	16.	16.
P:90000496--900004CF	ComplexDoWhile	never		0.000%	0.	0.	0.	5.	58.	0.
P:900004D0--900004FF	ComplexFor	never		0.000%	0.	0.	0.	5.	48.	0.
P:90000500--90000527	ComplexIf	partial		37.500%	1.	1.	0.	2.	40.	14.
P:90000528--90000569	ComplexWhile	never		0.000%	0.	0.	0.	5.	66.	0.
P:9000056A--9000056F	Identity	never		-	0.	0.	0.	0.	6.	0.
P:90000570--90000591	MultiLine	partial		41.666%	1.	2.	1.	2.	34.	20.
P:90000592--900005A7	NestedExpr	ok		-	0.	0.	0.	0.	22.	22.
P:900005A8--900005C9	NestedExprTrans	ok		100.000%	2.	0.	0.	0.	34.	34.
P:900005CA--90000615	RunCoverageDemo	partial		-	0.	0.	0.	0.	76.	62.
P:90000616--90000647	SwitchCase	taken		90.000%	4.	1.	0.	0.	50.	46.
P:90000648--9000065D	TernaryExpr	ok		100.000%	1.	0.	0.	0.	22.	22.
P:9000065E--90000673	TernaryExprTrans	ok		100.000%	1.	0.	0.	0.	22.	22.

Conditional branches	
branches	Percentage calculated according to the following formula: $$\frac{2 \times ok + taken + nottaken}{2 \times (ok + taken + nottaken + never)}$$
ok	Number of conditional branches that are both *taken* and *not taken*
taken	Number of conditional branches that are only *taken*
not taken	Number of conditional branches that are only *not taken*
never	Number of conditional branches that are neither *taken* nor *not taken*

Byte count	
bytes	Number of bytes
ok	Number of bytes that are already tagged as ok

```
// Demo script "~~/demo/t32cast/eca/measure_mcdc.cmm"

// Select code coverage metric object code
COVerage.Option SourceMetric ObjectCode

// List code coverage results at source and object code level
List.Mix MultiLine /COVerage

// List code coverage results at function level
COVerage.ListFunc.sYmbol \coverage
```

Object Code Based (ocb) Decision Coverage Evaluation

Lauterbach regards this code coverage metric as outdated. However, it can be helpful for special problems. If such a situation arises, our support team will inform you.

Evaluation Strategy

Decision coverage: Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once.

TRACE32 Interpretation: ocb decision coverage is achieved if full object code coverage is achieved.

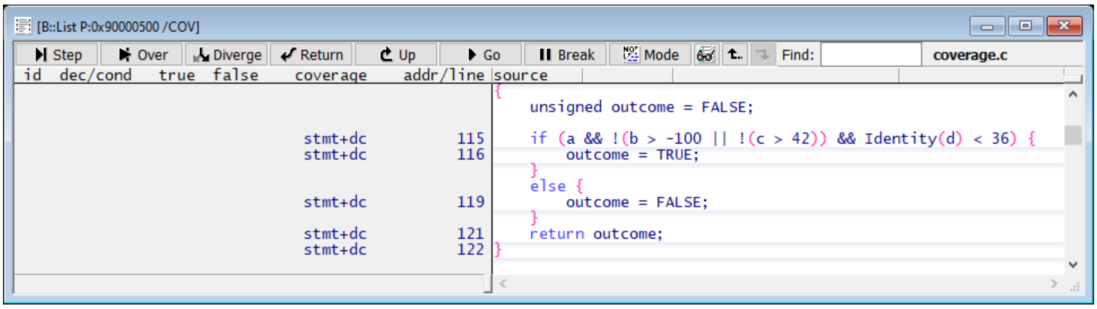
However, the following should be considered:

Unoptimized code can lead to false negative results. False negative means that decisions are tagged as incomplete although decision coverage has already been achieved. That means ocb decision coverage may need more test cases than the standard decision coverage

Optimized code can lead to false positive results if a condition is no longer represented by a conditional branch/instruction or the trace protocol provides no information about the state of conditional instructions. False positive means that decision coverage is indicated too early.

Since the source code is not analyzed for ocb decision coverage, TRACE32 does not know where decisions are located. Therefor source code lines are tagged as follows:

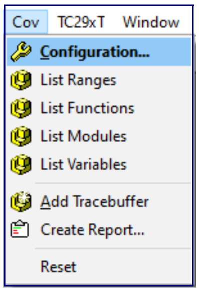
- **dc+stmt | incomplete**

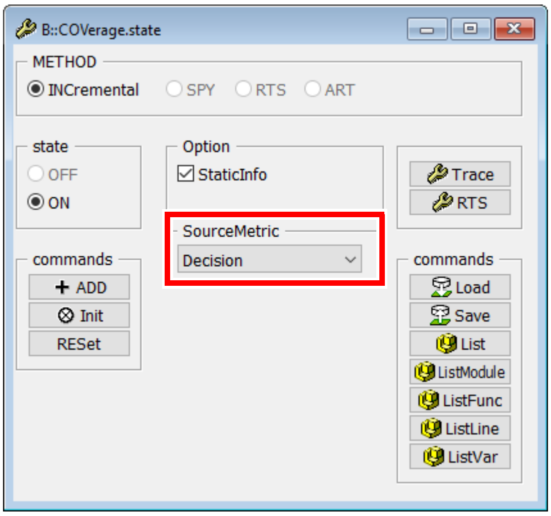


```
id  dec/cond   true  false    coverage     addr/line source
                                                     {
                                                         unsigned outcome = FALSE;

                                   stmt+dc         115   if (a && !(b > -100 || !(c > 42)) && Identity(d) < 36) {
                                   stmt+dc         116       outcome = TRUE;
                                                         }
                                                         else {
                                   stmt+dc         119       outcome = FALSE;
                                                         }
                                   stmt+dc         121   return outcome;
                                   stmt+dc         122 }
```

If you want to use the trace data stored in the code coverage system for ocb decision coverage, select the SourceMetric **Decision** in **COVerage state window** or use the command **COVerage.Option SourceMetric** Decision.

You cannot specifically select ocd decision coverage in TRACE32. TRACE32 automatically performs it if no .eca data was loaded during the code coverage measurement.





METHOD: [✓ INCremental] SPY RTS ART

state: OFF [✓ ON]

Option: [✓ StaticInfo]

SourceMetric: Decision

The following commands show a tabular analysis:

COVerage.ListModule

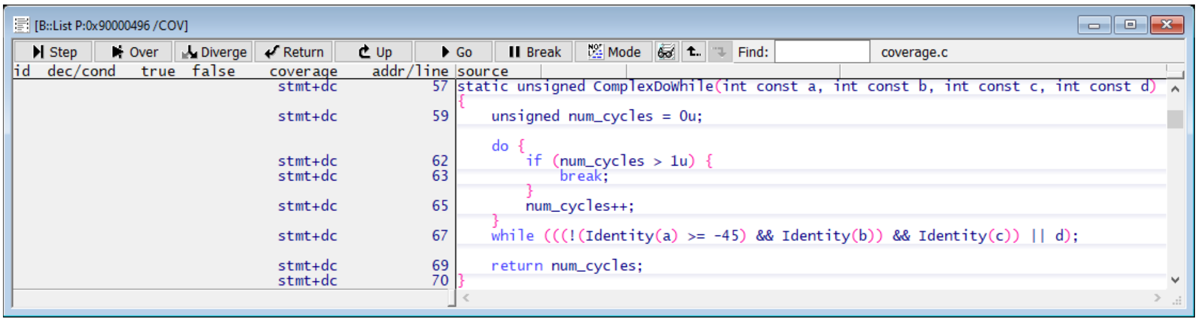
COVerage.ListFunc

The following command shows the tagging on source code level.

List.HII /COVerage

This TRACE32 command displays the ocb decision coverage tagging for the function *ComplexDoWhile*:

```
List.HLL ComplexDoWhile /COVerage
```



```
id  dec/cond   true  false      coverage      addr/line source
                                stmt+dc              57 static unsigned ComplexDoWhile(int const a, int const b, int const c, int const d)
                                                        {
                                stmt+dc              59     unsigned num_cycles = 0u;

                                                            do {
                                stmt+dc              62         if (num_cycles > 1u) {
                                stmt+dc              63             break;
                                                                }
                                stmt+dc              65         num_cycles++;
                                                            }
                                stmt+dc              67     while (((!(Identity(a) >= -45) && Identity(b)) && Identity(c)) || d);

                                stmt+dc              69     return num_cycles;
                                stmt+dc              70 }
```

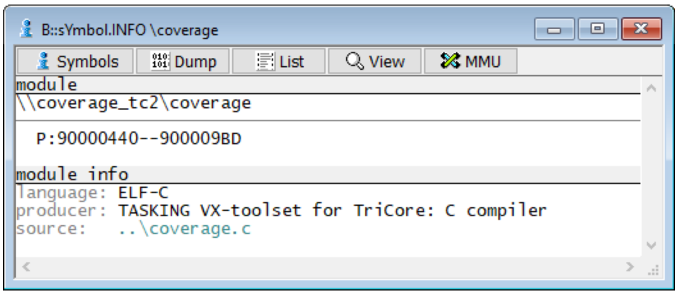
Source code lines are tagged as follows:

dc+stmt	The source code line achieved full object code coverage and thereby either decision or statement coverage.
incomplete	The source code line did not achieve full object code coverage and thereby no decision or statement coverage.

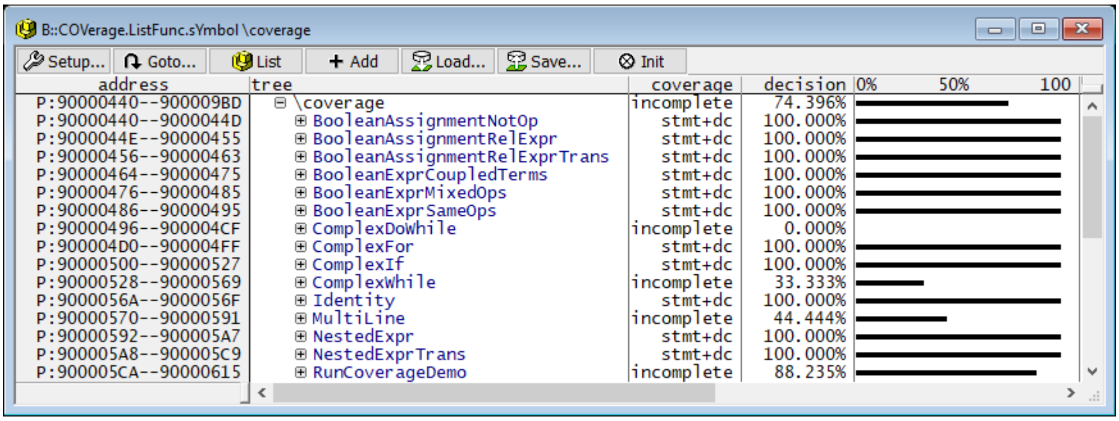
Object code instructions get object code tagging, if ocb decision coverage is performed.

This TRACE32 command displays a tabular analysis of all functions of the "coverage" module. A module usually corresponds to a source code file.

```
COVerage.ListFunc.sYmbol \coverage
```



```
module
\\coverage_tc2\coverage

  P:90000440--900009BD

module info
language:  ELF-C
producer:  TASKING VX-toolset for TriCore: C compiler
source:    ..\coverage.c
```

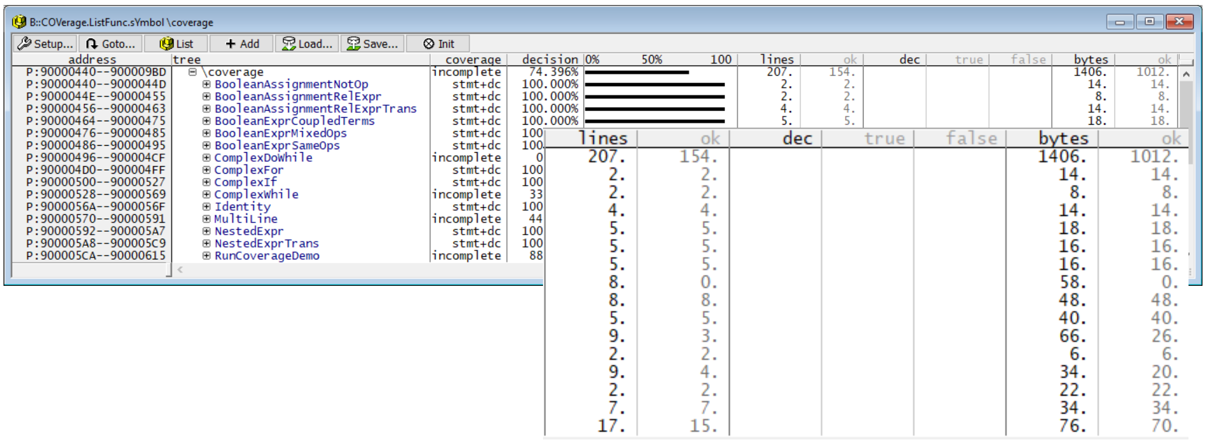


```
address               tree                              coverage    decision  0%     50%     100
P:90000440--900009BD  ⊟ \coverage                       incomplete   74.396%
P:90000440--9000044D    ⊞ BooleanAssignmentNotOp          stmt+dc   100.000%
P:9000044E--90000455    ⊞ BooleanAssignmentRelExpr        stmt+dc   100.000%
P:90000456--90000463    ⊞ BooleanAssignmentRelExprTrans   stmt+dc   100.000%
P:90000464--90000475    ⊞ BooleanExprCoupledTerms         stmt+dc   100.000%
P:90000476--90000485    ⊞ BooleanExprMixedOps             stmt+dc   100.000%
P:90000486--90000495    ⊞ BooleanExprSameOps              stmt+dc   100.000%
P:90000496--900004CF    ⊞ ComplexDoWhile               incomplete     0.000%
P:900004D0--900004FF    ⊞ ComplexFor                      stmt+dc   100.000%
P:90000500--90000527    ⊞ ComplexIf                       stmt+dc   100.000%
P:90000528--90000569    ⊞ ComplexWhile                 incomplete    33.333%
P:9000056A--9000056F    ⊞ Identity                        stmt+dc   100.000%
P:90000570--90000591    ⊞ MultiLine                    incomplete    44.444%
P:90000592--900005A7    ⊞ NestedExpr                      stmt+dc   100.000%
P:900005A8--900005C9    ⊞ NestedExprTrans                 stmt+dc   100.000%
P:900005CA--90000615    ⊞ RunCoverageDemo              incomplete    88.235%
```

Tags for Object Code Based (ocb) Decision Coverage

- **stmt+dc**: All source code lines of the function/module are tagged with stmt+dc.
- **incomplete**: At least one source code line of the function/module is tagged with incomplete.
-

Further details are displayed when you open the window in its full size:



```
       address       tree                                     coverage    decision 0%      50%      100   lines     ok   dec  true  false   bytes     ok
P:90000440--900009BD  ⊟ \coverage                            incomplete   74.396%                          207.   154.                      1406.   1012.
P:90000440--9000044D    ⊞ BooleanAssignmentNotOp               stmt+dc   100.000%                            2.     2.                        14.     14.
P:9000044E--90000455    ⊞ BooleanAssignmentRelExpr             stmt+dc   100.000%                            2.     2.                         8.      8.
P:90000456--90000463    ⊞ BooleanAssignmentRelExprTrans        stmt+dc   100.000%                            4.     4.                        14.     14.
P:90000464--90000475    ⊞ BooleanExprCoupledTerms              stmt+dc   100.000%                            5.     5.                        18.     18.
P:90000476--90000485    ⊞ BooleanExprMixedOps                  stmt+dc   100                                 5.     5.                        16.     16.
P:90000486--90000495    ⊞ BooleanExprSameOps                   stmt+dc   100                                 5.     5.                        16.     16.
P:90000496--900004CF    ⊞ ComplexDoWhile                     incomplete    0                                 8.     0.                        58.      0.
P:900004D0--900004FF    ⊞ ComplexFor                           stmt+dc   100                                 8.     8.                        48.     48.
P:90000500--90000527    ⊞ ComplexIf                            stmt+dc   100                                 5.     5.                        40.     40.
P:90000528--90000569    ⊞ ComplexWhile                       incomplete   33                                 9.     3.                        66.     26.
P:9000056A--9000056F    ⊞ Identity                             stmt+dc   100                                 2.     2.                         6.      6.
P:90000570--90000591    ⊞ MultiLine                          incomplete   44                                 9.     4.                        34.     20.
P:90000592--900005A7    ⊞ NestedExpr                           stmt+dc   100                                 2.     2.                        22.     22.
P:900005A8--900005C9    ⊞ NestedExprTrans                      stmt+dc   100                                 7.     7.                        34.     34.
P:900005CA--90000615    ⊞ RunCoverageDemo                    incomplete   88                                17.    15.                        76.     70.
```

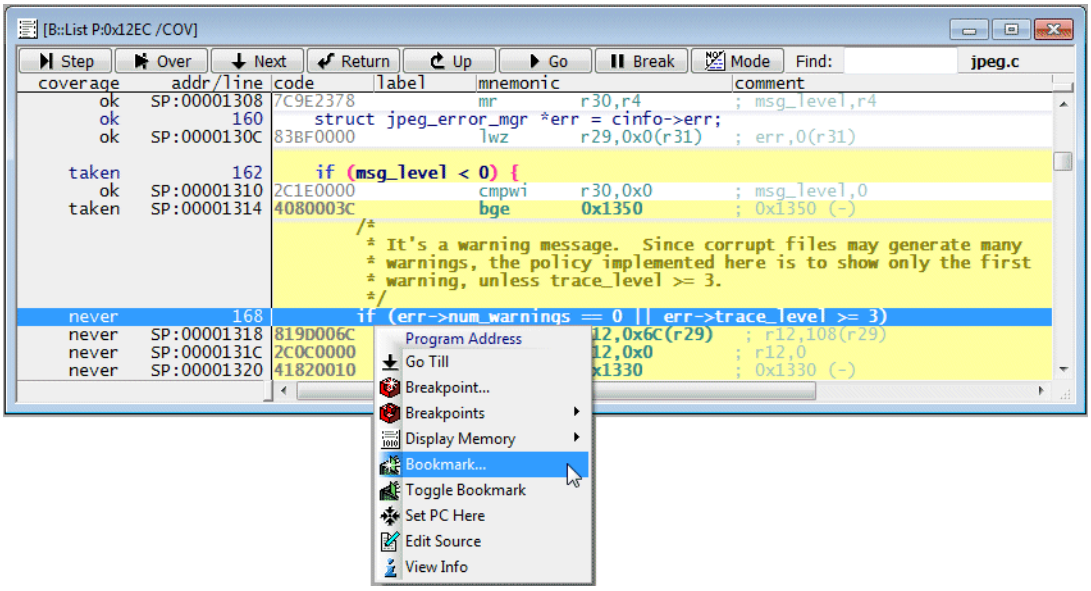
Line count	
lines	Number of source code lines within the function/module
ok	Number of source code lines tagged with stmt+dc

Byte count	
bytes	Number of bytes within the function/module
ok	Number of bytes tagged with stmt+dc

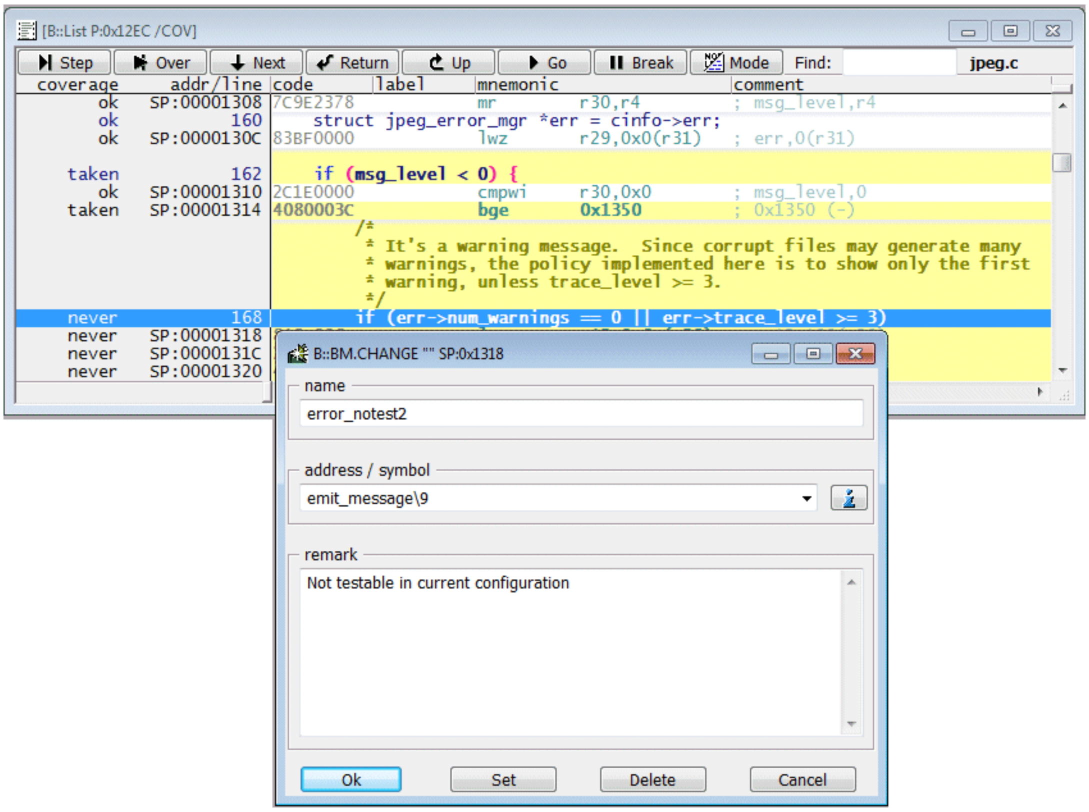
```
// Demo script "~~/demo/t32cast/eca/measure_mcdc.cmm"

// Select code coverage metric decision
COVerage.Option SourceMetric Decision

// List code coverage results at source code line level
List.Hll ComplexDoWhile /COVerage

// List code coverage results at function level
COVerage.ListFunc.sYmbol \coverage
```

Comment Your Results

Address-based bookmarks can be used to comment not covered code ranges, which are fine but not testable in the current system configuration.



```
coverage   addr/line code     label  mnemonic            comment
     ok  SP:00001308 7C9E2378        mr       r30,r4     ; msg_level,r4
     ok          160     struct jpeg_error_mgr *err = cinfo->err;
     ok  SP:0000130C 83BF0000        lwz      r29,0x0(r31)   ; err,0(r31)

  taken          162     if (msg_level < 0) {
     ok  SP:00001310 2C1E0000        cmpwi    r30,0x0    ; msg_level,0
  taken  SP:00001314 4080003C        bge      0x1350     ; 0x1350 (-)
                               /*
                                * It's a warning message.  Since corrupt files may generate many
                                * warnings, the policy implemented here is to show only the first
                                * warning, unless trace_level >= 3.
                                */
  never          168           if (err->num_warnings == 0 || err->trace_level >= 3)
  never  SP:00001318 819D006C                 12,0x6C(r29)   ; r12,108(r29)
  never  SP:0000131C 2C0C0000                 12,0x0     ; r12,0
  never  SP:00001320 41820010                 x1330      ; 0x1330 (-)
```



```
coverage   addr/line code     label  mnemonic            comment
     ok  SP:00001308 7C9E2378        mr       r30,r4     ; msg_level,r4
     ok          160     struct jpeg_error_mgr *err = cinfo->err;
     ok  SP:0000130C 83BF0000        lwz      r29,0x0(r31)   ; err,0(r31)

  taken          162     if (msg_level < 0) {
     ok  SP:00001310 2C1E0000        cmpwi    r30,0x0    ; msg_level,0
  taken  SP:00001314 4080003C        bge      0x1350     ; 0x1350 (-)
                               /*
                                * It's a warning message.  Since corrupt files may generate many
                                * warnings, the policy implemented here is to show only the first
                                * warning, unless trace_level >= 3.
                                */
  never          168           if (err->num_warnings == 0 || err->trace_level >= 3)
  never  SP:00001318
  never  SP:0000131C
  never  SP:00001320
```

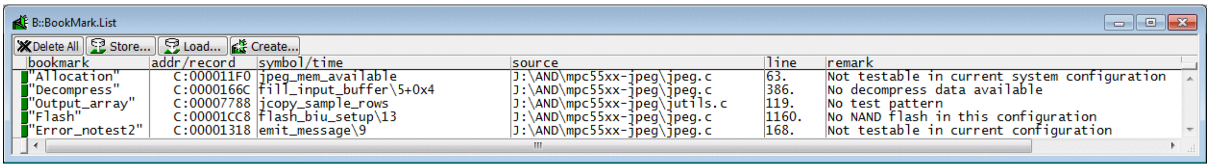
name: error_notest2

address / symbol: emit_message\9

remark: Not testable in current configuration

List all bookmarks:

```
BookMark.List
```



```
bookmark          addr/record  symbol/time              source                          line   remark
"Allocation"      C:000011F0   jpeg_mem_available       J:\AND\mpc55xx-jpeg\jpeg.c      63.    Not testable in current system configuration
"Decompress"      C:0000166C   fill_input_buffer\5+0x4  J:\AND\mpc55xx-jpeg\jpeg.c      386.   No decompress data available
"Output_array"    C:00007788   jcopy_sample_rows        J:\AND\mpc55xx-jpeg\jutils.c    119.   No test pattern
"Flash"           C:00001CC8   flash_biu_setup\13       J:\AND\mpc55xx-jpeg\jpeg.c      1160.  No NAND flash in this configuration
"Error_notest2"   C:00001318   emit_message\9           J:\AND\mpc55xx-jpeg\jpeg.c      168.   Not testable in current configuration
```

The current bookmarks can be saved to a file and reloaded later on.

```
STOre <file> BookMark
```

TRACE32 Merge and Report Tool

Typically, code coverage is not measured in a single test pass, but is approached gradually. This creates the need for:

- saving the results from single test passes.
- merging the saved results and/or to generate an overall report.

As already described, the **COVerage.EXPORT.JSONE** command allows you to export information on the functions and source code lines from the code coverage system to a JSON file. Lauterbach offers the command line tool **t32covtool** to merge the exported results and/or create an overall report. t32covtool runs on Windows and Linux.

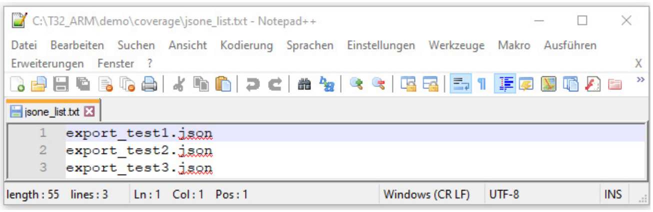
t32covtool can be used for the source metrics statement, full decision, condition coverage, MC/DC as well as call and function coverage.

It cannot process object code metrics and is therefore not suitable for object code and object code based decision coverage.

The command line tool t32covtool and its options.

```
t32covtool <options> <input>
```

-f **--force-overwrite**	Optional, overwrite output directory if existing.
-h **--help**	Print help.
-j **--output-json** *<file>*	Merge JSON files into a summary JSON file.

-l
--filelist *<file>*
	The *<input>* to t32covtool can be either a number of JSON files or a .txt file containing a list of JSON files (option --filelist). Using a .txt file is particularly recommended when there are many JSON files. In the .txt file, each JSON file should be listed on a separate line, as shown in the example.
	
	 C:\T32_ARM\demo\coverage\jsone_list.txt - Notepad++
	Datei Bearbeiten Suchen Ansicht Kodierung Sprachen Einstellungen Werkzeuge Makro Ausführen
	Erweiterungen Fenster ?
	jsone_list.txt
	length : 55 lines : 3 Ln : 1 Col : 1 Pos : 1 Windows (CR LF) UTF-8 INS
	`export_test1.json`
	`export_test2.json`
	`export_test3.json`

-m **--source-metric** *<metric>*	Choose source code metric for report. Supported metrics are: statement, decision, condition, mcds, call, function
-o *<dir>* **--outputdir** *<dir>*	Optional, set output directory.
-v **--version**	Print version.

Example 1

Generate an HTML report
- specify the source metric *decision*.
- specify *report_24* as output directory, advise t32covtool to overwrite the directory if it already exists.
- specify the input files.

```
t32covtool --source-metric decision
--outputdir report_24 --force-overwrite
export_unittest1.json export_unittest2.json export_unittest3.json
```

Example 2

Generate an HTML report
- specify the source metric *statement*.
- specify *report_24* as output directory, advise t32covtool to overwrite the directory if it already exists.
- specify *jsone_list.txt* that include list of input files.

```
t32covtool --source-metric  statement
--outputdir report_24 --force-overwrite
--filelist jsone_list.txt
```

Example 3

Generate an html report and a summary JSON file
- specify the source metric *decision*.
- specify *report_24* as output directory, advise t32covtool to overwrite the directory if it already exists.
- specify the files name for the accumulated JSON.
- specify *jsone_list.txt* that include list of input files.

```
t32covtool --source-metric decision
--outputdir report_24 --force-overwrite --output-json sum.json
--filelist jsone_list.txt
```

Example 4

Generate an accumulated JSON file.
- specify the files name for the accumulated JSON.
- specify *jsone_list.txt* that include list of input files.

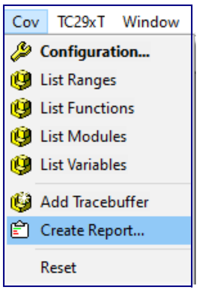
```
t32covtool --output-json sum.json
--filelist jsone_list.txt
```

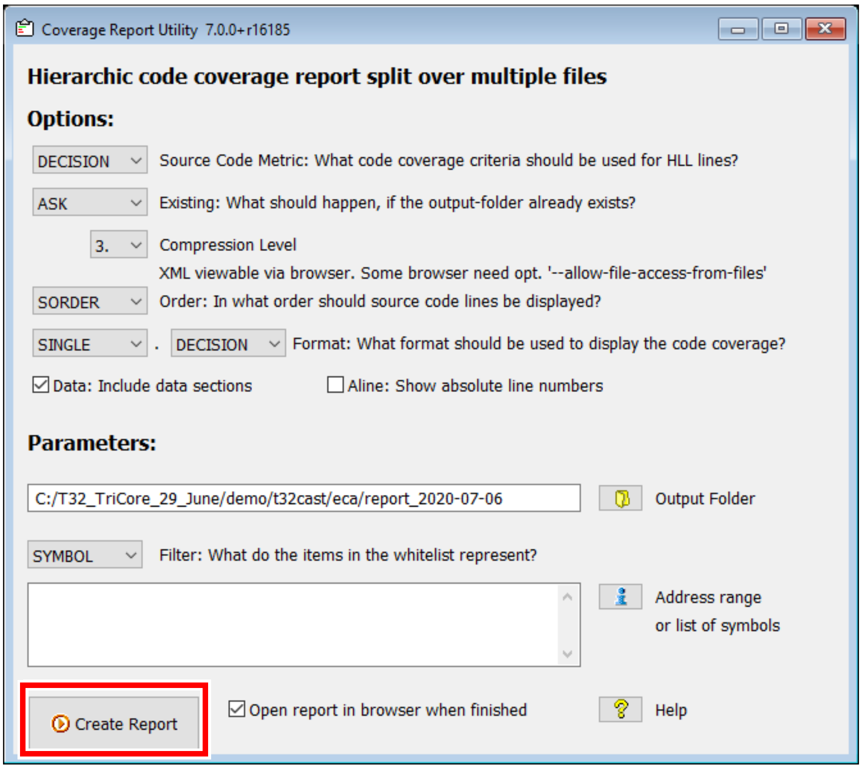
You can find a sample script for using the command line tool **t32covtool** at `~~/demo/coverage/merge_demo/merge_unittests/demo.cmm`.

Appendix A: TRACE32 Coverage Report Utility

After the code coverage measurement is completed, a code coverage report has to be generated in order to document the results. TRACE32 includes a Coverage Report Utility for this purpose.

Choose **Create Report...** in the **Cov** menu to open the **TRACE32 Coverage Report Utility**.





Coverage Report Utility 7.0.0+r16185

Hierarchic code coverage report split over multiple files

Options:

DECISION — Source Code Metric: What code coverage criteria should be used for HLL lines?

ASK — Existing: What should happen, if the output-folder already exists?

3. — Compression Level
XML viewable via browser. Some browser need opt. '--allow-file-access-from-files'

SORDER — Order: In what order should source code lines be displayed?

SINGLE . DECISION — Format: What format should be used to display the code coverage?

☑ Data: Include data sections ☐ Aline: Show absolute line numbers

Parameters:

C:/T32_TriCore_29_June/demo/t32cast/eca/report_2020-07-06 — Output Folder

SYMBOL — Filter: What do the items in the whitelist represent?

Address range or list of symbols

Create Report ☑ Open report in browser when finished Help

Push the **Create Report** button to generate a standard report.

The implementation of the dialog can be found in the following PRACTICE script:
"~~/demo/coverage/multi_file_report/create_report.cmm".

The comments in the script contain information against which browsers the script was tested and which additional setting might be necessary. It is recommended to read this in advance.

```
PEDIT ~~/demo/coverage/multi_file_report/create_report.cmm
```

If you start the script with parameters, the script is directly executed.

```
CD.DO ~~/demo/coverage/multi_file_report/create_report.cmm \
"manual" "SYMBOL" "\coverage" \
"METRIC=DECISION EXISTING=REPLACE COMPRESSION=2"
```

Note

For larger projects it is recommended to copy the object code into the **TRACE32 Virtual Memory**. This makes the creation of the report faster. Here a short script example.

```
Data.Load.elf my_project /VM          ; Load your code again, this time
                                      ; into the TRACE32 Virtual Memory.

Trace.ACCESS VM                       ; Advise TRACE32 to use the code
                                      ; loaded to the TRACE32 Virtual
                                      ; Memory for trace decoding

…                                     ; Create your report

Trace.ACCESS auto                     ; Reset the TRACE.ACCESS to its
                                      ; default
```

If you use dynamic memory management (MMU) with SYStem.Option MMUSPACES ON, the following command sequence is recommended:

```
TRANSlation.SHADOW ON                   ; Allow several address spaces
                                        ; in TRACE32 Virtual Memory

Data.LOAD.Elf my_project 0x2::0 /VM     ; Load your code again, e.g. to
                                        ; space ID 0x2, this time into
                                        ; the TRACE32 virtual memory

Trace.ACCESS VM                         ; Advise TRACE32 to use the code
                                        ; loaded to the TRACE32 Virtual
                                        ; Memory for trace decoding

…                                       ; Create your report

Trace.ACCESS auto                       ; Reset the TRACE.ACCESS to its
                                        ; default

TRANSlation.SHADOW OFF                  ; Reset TRANSlation.SHADOW to
                                        ; its default
```

Appendix B: Assemble Multiple Test Runs at Address Level

There are two ways to assemble multiple test runs.

- Save and reload the data content of the code coverage system
- Save and reload the complete trace information

NOTE:	Please make sure that you only assemble test runs that were carried out with the identical executable(s).

Save and Restore Code Coverage Measurement

COVerage.SAVE *<file>*	This command saves the following data in the specified *<file>*: object code coverage tagging based on addresses the MC/DC status of all conditions based on their addresses The default extension is .acd (Analyzer Coverage Data).

To assemble the results from several test runs, you can use:

- Your TRACE32 debug and trace tool connected to your target hardware.
- Alternatively you can use a TRACE32 Instruction Set Simulator (see **"Section PBI=<driver>"** in TRACE32 Installation Guide, page 48 (installation.pdf)).

Before you load an acd file into TRACE32 with the following command you need to make sure, that:

- the test executable has been loaded into memory
- the debug symbol information for the test executable has been loaded
- if needed for the selected code coverage metric, .eca files are loaded

COVerage.LOAD *<file>* **/Replace**	Load coverage data from *<file>* into the TRACE32 code coverage system. All existing coverage data is cleared.
COVerage.LOAD *<file>* **/Add**	Add coverage data from *<file>* to the TRACE32 code coverage system.

Example script

Save data content of the code coverage system:

```
COVerage.SAVE testrun1.acd

...

COVerage.SAVE testrun2.acd

...
```

Assemble coverage data from several test runs:

```
...                                         ; Basic setups

Data.LOAD.Elf jpeg.elf                      ; Load code into memory and
                                            ; debug info into TRACE32

// sYmbol.ECA.LOADALL /SkipErrors           ; Load .eca files if needed

COVerage.LOAD testrun1.acd /Replace

COVerage.LOAD testrun2.acd /Add

...

COVerage.Option SourceMetric Statement      ; Specify code coverage metric

...

COVerage.ListFunc                           ; Display code coverage for
                                            ; all functions
```

Save and Restore Trace Recording

Trace.SAVE *<file>*	Save trace buffer contents to *<file>*.

Saving the trace buffer contents enables you to re-examine your tests in detail any time.

To assemble the results from several test runs, you can use:

- Your TRACE32 debug and trace tool connected to your target hardware.
- Alternatively you can use a TRACE32 Instruction Set Simulator (see **"Section PBI=<driver>"** in TRACE32 Installation Guide, page 48 (installation.pdf)).

In either case you need to make sure, that the debug symbol information for the test executable has been loaded into TRACE32 PowerView.

Trace.LOAD *<file>*	Load trace information from *<file>* to TRACE32. The default extension is .ad (Analyzer Data).
COVerage.ADD	Add loaded trace information into the TRACE32 code coverage system.

Example script

Save trace buffer contents of several tests to files.

```
Trace.SAVE test1.ad

...

Trace.SAVE test2.ad

...
```

Reload saved trace buffer contents and add them to the code coverage system.

```
...                                       ; Basic setups

Data.LOAD.Elf jpeg.elf                    ; Load debug info into TRACE32

// sYmbol.ECA.LOADALL /SkipErrors         ; Load .eca files if needed

Trace.LOAD test1.ad                       ; Load trace information from
                                          ; file
```

```
COVerage.ADD                                ; add the trace information
                                            ; into code coverage system

Trace.LOAD test2.ad                         ; load trace information from
                                            ; next file

COVerage.ADD                                ; add the trace information
                                            ; into code coverage system

...

COVerage.Option SourceMetric Statement      ; specify code coverage metric

COVerage.ListFunc                           ; Display coverage for all
                                            ; functions

...

Trace.LOAD test2.ad                         ; load trace information from
Trace.List                                  ; file for detailed
                                            ; re-examination
```

Appendix C: Assembler-Only Functions and Code Coverage

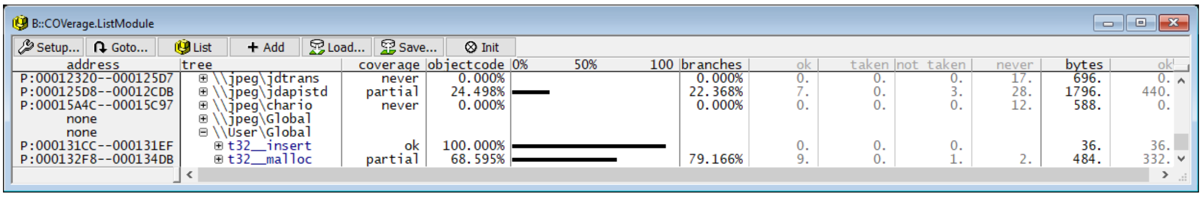
Object Code Coverage

Code that is not part of a source code function is discarded for the object code coverage. If you want to include this code you have to assign a function name to it:

sYmbol.INFO *<symbol>*	Display details about a debug symbol.
sYmbol.RANGE(*<symbol>)*	Returns the address range used by the specified symbol.
sYmbol.NEW.Function *<name> <addressrange>*	Create a function.

```
sYmbol.NEW.Function t32__malloc sYmbol.RANGE(__malloc)

sYmbol.NEW.Function t32__insert sYmbol.RANGE(__insert)
```

The manually created functions are assigned to the \\User\Global module.



address	tree	coverage	objectcode	0% 50% 100	branches	ok	taken	not taken	never	bytes	ok
P:00012320--000125D7	⊞ \\jpeg\jdtrans	never	0.000%		0.000%	0.	0.	0.	17.	696.	0.
P:000125D8--00012CDB	⊞ \\jpeg\jdapistd	partial	24.498%		22.368%	7.	0.	3.	28.	1796.	440.
P:00015A4C--00015C97	⊞ \\jpeg\chario	never	0.000%		0.000%	0.	0.	0.	12.	588.	0.
none	⊞ \\jpeg\Global										
none	⊟ \\User\Global										
P:000131CC--000131EF	⊞ t32__insert	ok	100.000%			0.	0.	0.		36.	36.
P:000132F8--000134DB	⊞ t32__malloc	partial	68.595%		79.166%	9.	0.	1.	2.	484.	332.

The object code lines of the assembler functions are marked with the same tags as the object code lines of source code functions.

Source Code Metrics

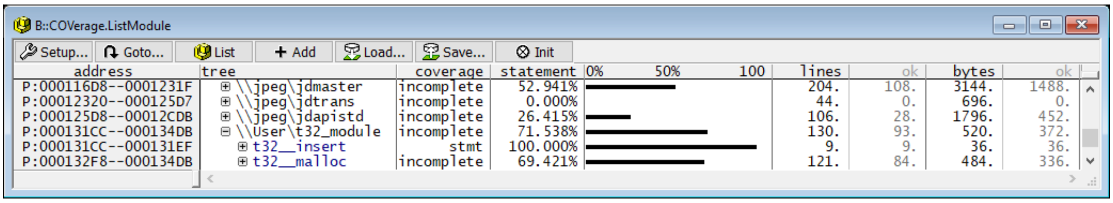
Code that is not part of a source code function is discarded for coverage. If you want to include this code you have to assign a function to it:

sYmbol.INFO *<symbol>*	Display details about a debug symbol.
sYmbol.RANGE(*<symbol>***)**	Returns the address range used by the specified symbol.
sYmbol.NEW.Function *<name> <addressrange>*	Create a function.
sYmbol.NEW.Module *<name> <addressrange>*	Create a module.

Functions created with the **sYmbol.NEW.Function** command are grouped under the module name \\User\Global. No address range is assigned to this module. Alternatively, several functions can be aggregated under a newly created module. An address range has to be assigned to the new module \\Global\<name> when it is created and it then includes all functions that are located within its address range.

```
sYmbol.INFO __malloc

sYmbol.INFO __insert

sYmbol.NEW.Module t32_module P:0x000131cc--0x00134db
```

```
sYmbol.NEW.Function t32__malloc sYmbol.RANGE(__malloc)

sYmbol.NEW.Function t32__insert sYmbol.RANGE(__insert)
```



```
      address       tree                   coverage  statement 0%      50%     100    lines     ok    bytes      ok
 P:000116D8--0001231F  ⊞ \\jpeg\jdmaster    incomplete   52.941%                        204.    108.   3144.   1488.
 P:00012320--000125D7  ⊞ \\jpeg\jdtrans     incomplete    0.000%                         44.      0.    696.      0.
 P:000125D8--00012CDB  ⊞ \\jpeg\jdapistd    incomplete   26.415%                        106.     28.   1796.    452.
 P:000131CC--000134DB  ⊟ \\User\t32_module  incomplete   71.538%                        130.     93.    520.    372.
 P:000131CC--000131EF    ⊞ t32__insert            stmt  100.000%                          9.      9.     36.     36.
 P:000132F8--000134DB    ⊞ t32__malloc      incomplete   69.421%                        121.     84.    484.    336.
```

Depending on the selected source code metric, the assembler functions or the modules are tagged as follows:

Metric	Tag	Description
all source code metrics	incomplete	At least one assembler line within the function is tagged with never, taken or not taken.
Statement	stmt	All assembler lines are tagged with ok.

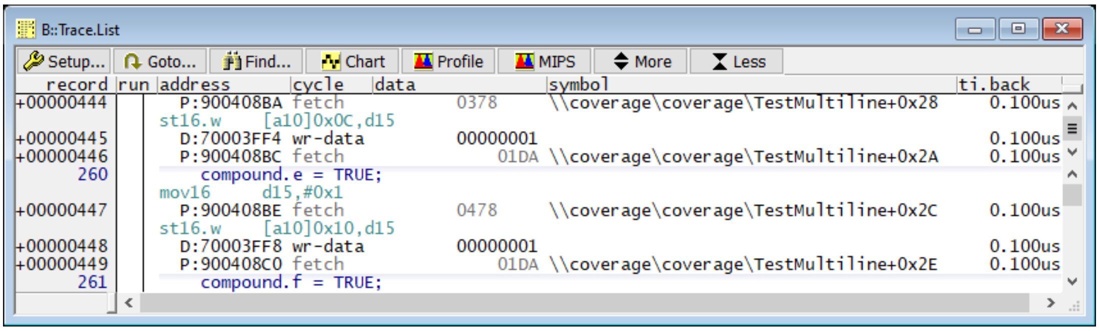
Metric	Tag	Description
Decision	stmt+dc	All assembler lines are tagged with ok.
CONDition	stmt+cc	All assembler lines are tagged with ok.
MCDC	stmt+mc/dc	All assembler lines are tagged with ok.
Function	func	All assembler lines are tagged with ok.
Call	call	All assembler lines are tagged with ok.

Appendix D: Data Coverage

Trace Data Collection

Since off-chip trace ports usually do not have enough bandwidth to make all read/write accesses (and the program flow) visible, they are rather unsuitable for data coverage. For test phases in which testing in the target environment is not yet required, a TRACE32 Instruction Set Simulator can be used well for data coverage.

Since TRACE32 Instruction Set Simulators provide full program and data flow trace based on a bus trace protocol, no special setup is required.



```
    record run address       cycle    data              symbol                                ti.back
+00000444      P:900408BA fetch       0378      \\coverage\coverage\TestMultiline+0x28    0.100us
             st16.w     [a10]0x0C,d15
+00000445      D:70003FF4 wr-data     00000001                                            0.100us
+00000446      P:900408BC fetch           01DA  \\coverage\coverage\TestMultiline+0x2A    0.100us
       260       compound.e = TRUE;
             mov16      d15,#0x1
+00000447      P:900408BE fetch       0478      \\coverage\coverage\TestMultiline+0x2C    0.100us
             st16.w     [a10]0x10,d15
+00000448      D:70003FF8 wr-data     00000001                                            0.100us
+00000449      P:900408C0 fetch           01DA  \\coverage\coverage\TestMultiline+0x2E    0.100us
       261       compound.f = TRUE;
```

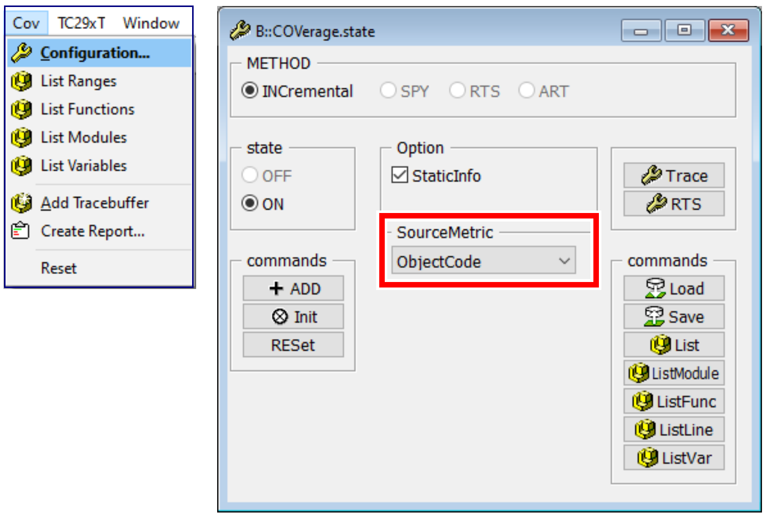
If you want to use an onchip trace or an offchip trace port for data tracing, please refer to the following documents for setup details:

- Arm: **“Training Arm CoreSight ETM Tracing”** (training_arm_etm.pdf), **“Training Cortex-M Tracing”** (training_cortexm_etm.pdf)
- MPC5xxx/SPC5xxx, QorIQ and RH850: **“Training MPC5xxx/SPC5xx Nexus Tracing”** (training_nexus_mpc5500.pdf)
- TriCore: **“Training AURIX Tracing”** (training_aurix_trace.pdf)
- For other processor architectures, please refer to the corresponding **“Processor Architecture Manuals”**.

Please note that data coverage only makes sense if the trace does not contain a high number of **TARGET FIFO OVERFLOWS**.

It is recommended to use incremental coverage for data coverage (see **“Incremental Code Coverage”**, page 77).

If you want to use the trace data stored in the coverage system for data coverage, select the SourceMetric **ObjectCode** in the **COVerage configuration window** or use the command **COVerage.Option SourceMetric** ObjectCode.



The following commands show a tabular analysis:

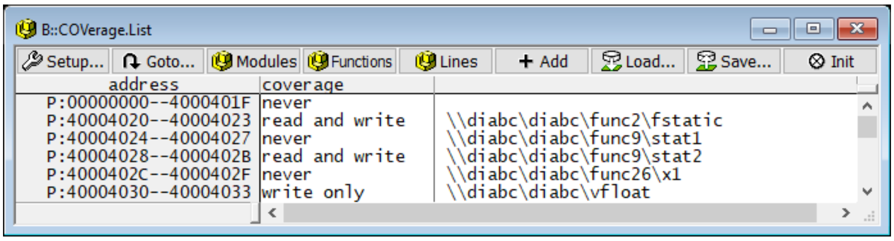
```
COVerage.List

COVerage.ListVar
```

The following command shows the tagging per address.

```
Data.View %Var <address> /COVerage
```

This TRACE32 command shows the coverage tagging on address range level:

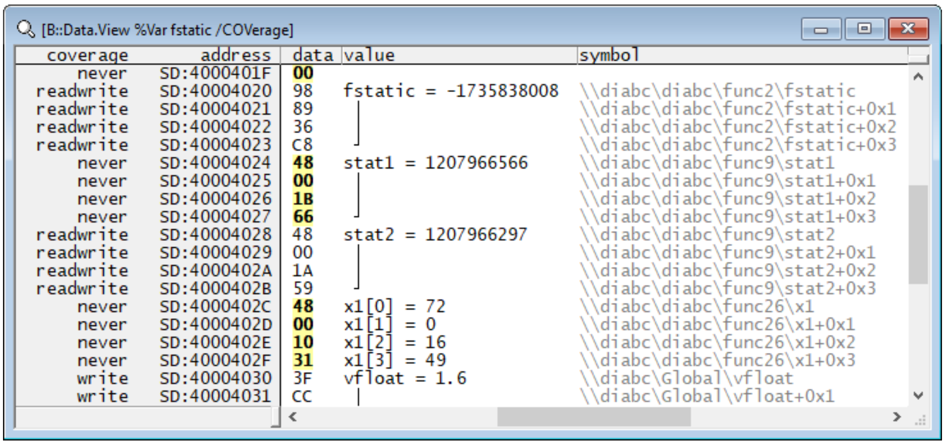
```
COVerage.List
```



```
      address         coverage
 P:00000000--4000401F never
 P:40004020--40004023 read and write   \\diabc\diabc\func2\fstatic
 P:40004024--40004027 never            \\diabc\diabc\func9\stat1
 P:40004028--4000402B read and write   \\diabc\diabc\func9\stat2
 P:4000402C--4000402F never            \\diabc\diabc\func26\x1
 P:40004030--40004033 write only       \\diabc\diabc\vfloat
```

This TRACE32 command shows the coverage tagging at address level starting with the address of the variable fstatic:

```
Data.View %Var fstatic /COVerage
```



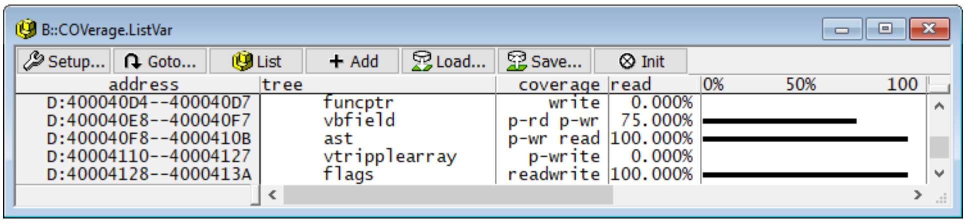
```
  coverage      address  data value                    symbol
     never  SD:4000401F  00
 readwrite  SD:40004020  98   fstatic = -1735838008    \\diabc\diabc\func2\fstatic
 readwrite  SD:40004021  89   |                        \\diabc\diabc\func2\fstatic+0x1
 readwrite  SD:40004022  36   |                        \\diabc\diabc\func2\fstatic+0x2
 readwrite  SD:40004023  C8                            \\diabc\diabc\func2\fstatic+0x3
     never  SD:40004024  48   stat1 = 1207966566       \\diabc\diabc\func9\stat1
     never  SD:40004025  00   |                        \\diabc\diabc\func9\stat1+0x1
     never  SD:40004026  1B   |                        \\diabc\diabc\func9\stat1+0x2
     never  SD:40004027  66                            \\diabc\diabc\func9\stat1+0x3
 readwrite  SD:40004028  48   stat2 = 1207966297       \\diabc\diabc\func9\stat2
 readwrite  SD:40004029  00   |                        \\diabc\diabc\func9\stat2+0x1
 readwrite  SD:4000402A  1A   |                        \\diabc\diabc\func9\stat2+0x2
 readwrite  SD:4000402B  59                            \\diabc\diabc\func9\stat2+0x3
     never  SD:4000402C  48   x1[0] = 72               \\diabc\diabc\func26\x1
     never  SD:4000402D  00   x1[1] = 0                \\diabc\diabc\func26\x1+0x1
     never  SD:4000402E  10   x1[2] = 16               \\diabc\diabc\func26\x1+0x2
     never  SD:4000402F  31   x1[3] = 49               \\diabc\diabc\func26\x1+0x3
     write  SD:40004030  3F   vfloat = 1.6             \\diabc\Global\vfloat
     write  SD:40004031  CC   |                        \\diabc\Global\vfloat+0x1
```

The data addresses are tagged as follow:

readwrite	The data address was read at least once and written at least once.
read	The data address has been read at least once.
write	The data address has been written at least once.
never	The data address was neither read nor written

This TRACE32 command displays the data coverage at variable level.

```
COVerage.ListVar
```



```
   address              tree              coverage  read     0%     50%    100
D:400040D4--400040D7       funcptr           write    0.000%
D:400040E8--400040F7       vbfield      p-rd p-wr   75.000%
D:400040F8--4000410B       ast          p-wr read  100.000%
D:40004110--40004127       vtripplearray   p-write    0.000%
D:40004128--4000413A       flags         readwrite 100.000%
```

Each static variable occupies a fixed address range. This results in the following tagging for variables:

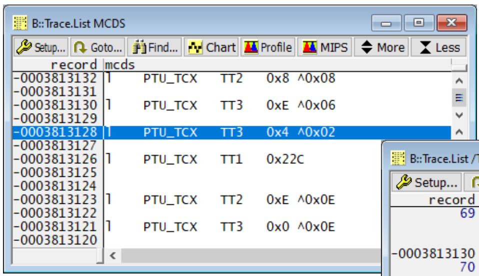
readwrite	Read and write accesses were performed for all addresses within the address range of the variable.
read	Only read accesses were performed for all addresses within the address range of the variable.
write	Only write accesses were performed for all addresses within the address range of the variable.
p-write	Write accesses were performed only to a part of the address range of the variable. No read accesses were performed.
p-read	Read accesses were performed only to a part of the address range of the variable. No write accesses were performed.
p-wr read	Write accesses were performed only to a part of the address range of the variable. Read accesses were performed for all addresses.
p-rd write	Read accesses were performed only to a part of the address range of the variable. Write accesses were performed for all addresses.
p-rd p-wr	Both read and write accesses were performed only to a part of the address range of the variable.
never	Not a single address of the address range of the variable was read or written.

The tags **rdwr ok**, **write ok**, **read ok** and **partial** indicate that TRACE32 cannot clearly recognize whether the address range contains program code or data. Please check your TRACE32 configuration or contact your local technical support.

A complete list of all data coverage tags can be found in **“Appendix E: Data Coverage in Detail”**, page 140.

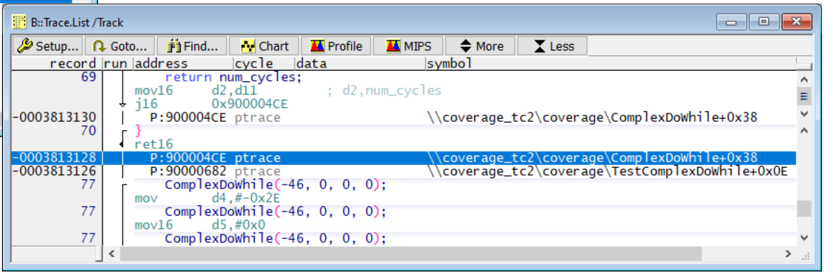
Appendix E: Trace Decoding in Detail

Before the recorded trace data can be analyzed, it must be decoded first.



```
      record mcds
-0003813132 ⌉     PTU_TCX    TT2    0x8  ^0x08
-0003813131
-0003813130 ⌉     PTU_TCX    TT3    0xE  ^0x06
-0003813129
-0003813128 ⌉     PTU_TCX    TT3    0x4  ^0x02
-0003813127
-0003813126 ⌉     PTU_TCX    TT1    0x22C
-0003813125
-0003813124
-0003813123 ⌉     PTU_TCX    TT2    0xE  ^0x0E
-0003813122
-0003813121 ⌉     PTU_TCX    TT3    0x0  ^0x0E
-0003813120
```

Raw trace data



```
      record run address     cycle   data            symbol
          69         return num_cycles;
               mov16     d2,d11          ; d2,num_cycles
               j16       0x900004CE
-0003813130      P:900004CE ptrace                   \\coverage_tc2\coverage\ComplexDoWhile+0x38
          70   }
               ret16
-0003813128      P:900004CE ptrace                   \\coverage_tc2\coverage\ComplexDoWhile+0x38
-0003813126      P:90000682 ptrace                   \\coverage_tc2\coverage\TestComplexDoWhile+0x0E
          77         ComplexDoWhile(-46, 0, 0, 0);
               mov       d4,#-0x2E
          77         ComplexDoWhile(-46, 0, 0, 0);
               mov16     d5,#0x0
          77         ComplexDoWhile(-46, 0, 0, 0);
```

Decoded trace data

Trace Decoding for Static Applications

The object and source code is required to decode trace raw data recorded of static programs.

Decoding in Stopped State for Static Applications

This decoding is used for incremental code coverage and incremental code coverage in stream mode.

TRACE32 state: program execution stopped, no recording of trace data.

TRACE32 can read the object code from the target memory. Links to the source code files are part of the debug symbol information maintained by TRACE32.

Decoding in Running State for Static Applications

This decoding is used in SPY mode code coverage.

TRACE32 state: program execution is running, trace data is recorded, but trace streaming is stalled while trace decoding is performed.

TRACE32 can read the object code from the target memory, if the core allows the debugger to read memory while the program execution is running (see also **Run-time Memory Access**).

However, TRACE32 can decode the trace data much faster if it does not have to access the target memory. That is why it is highly recommended to copy the object code into the **TRACE32 Virtual Memory**. This is achieved by the **/PlusVM** option when the program is loaded. The PlusVM option directs TRACE32 to load the object code into the target memory plus into the TRACE32 virtual memory.

```
Data.LOAD.Elf ~~~~/tricore/coverage_tc2.elf /RelPATH /PlusVM
```

The **Data.COPY** command is another possibility. It allows to copy the content of the target memory directly to the **TRACE32 Virtual Memory**.

Data.Copy *<address_range>* **VM:**

NOTE:	The object code required for trace decoding must be available in the TRACE32 Virtual Memory **before** the program execution and the trace recording is started.

RTS Decoding for Static Applications

This decoding is used in RTS mode code coverage.

TRACE32 state: program execution is running, trace data is recorded and streamed to the host computer.

If trace data is decoded at program runtime and processed while streaming, decoding has to be as fast as possible. An important prerequisite is that the object code is located in the **TRACE32 Virtual Memory**. This is achieved by the **/PlusVM** option when the program is loaded. The PlusVM option directs TRACE32 to load the object code into the target memory plus into the TRACE32 virtual memory.

```
Data.LOAD.Elf ~~~~/tricore/coverage_tc2.elf /RelPATH /PlusVM
```

The **Data.COPY** command is an another possibility. It allows to copy the content of the target memory directly to the **TRACE32 Virtual Memory**.

Data.Copy *<address_range>* **VM:**

NOTE:	The object code required for trace decoding must be available in the TRACE32 Virtual Memory **before** the program execution and the trace recording is started.

Trace Decoding for Applications Using a Rich OS

Also in this case, the object code and source code are needed to decode the trace raw data. But paging used by the operating system makes decoding more complex.

Since the onchip trace logic generates the program flow data based on virtual addresses, TRACE32 has to know the valid memory space for each trace record in order to read the object code from the physical memory for trace decoding. A task or context switch in the trace recording normally identifies the memory space for the subsequent logical addresses.

Decoding in Stopped State (Rich OS)

This decoding is used for incremental code coverage and incremental code coverage in stream mode.

TRACE32 state: program execution stopped, no recording of trace data.

Trace decoding is performed in three steps:

1. TRACE32 reads the current task list and all task page tables with the help of the TRACE32 OS Awareness from the target, when the program execution is stopped.
2. Task/context switches from the trace recording are decoded with the help of the task list.
3. The object code for each task is then read with the help of its page table. Links to the source code files are part of the debug symbol information, which TRACE32 maintains for each memory space.

   Reading the object code fails, when a task/context switch from the trace recording can not be decoded with the help of the current task list, e.g. because the task was terminated.

Decoding in Running State (Rich OS)

This decoding is used in Spy mode code coverage.

TRACE32 state: program execution is running, trace data is recorded, but trace streaming is stalled while trace decoding is performed.

TRACE32 has no access to the current task list and the task page tables while the program execution is running. The **TRACE32 Virtual Memory** must contain the task list, all task page tables and the object code to enable TRACE32 to decode the raw trace data.

This requires a complex setup. Please contact the Lauterbach support in this case.

RTS Decoding (Rich OS)

This decoding is used in RTS mode code coverage.

TRACE32 state: program execution is running, trace data is recorded and streamed to the host computer.

TRACE32 has no access to the current task list and the task page tables while the program execution is running. The **TRACE32 Virtual Memory** must contain the task list, all task page tables and the object code to enable TRACE32 to decode the raw trace data.

This requires a complex setup. Please contact the Lauterbach support in this case.

Appendix F: Coding Guidelines

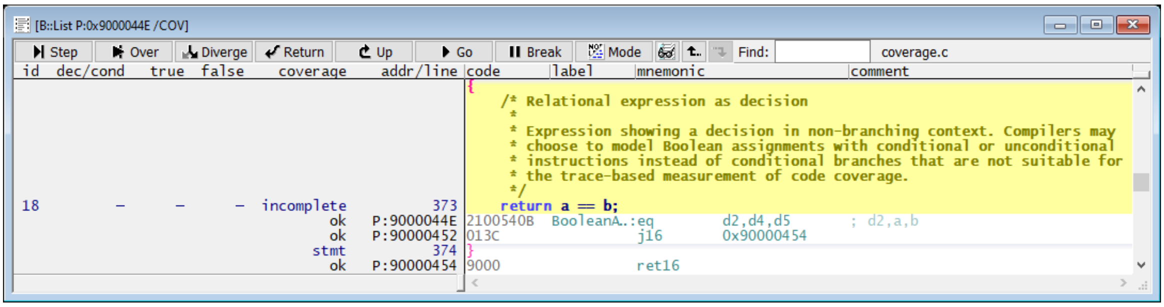
The following coding guidelines are recommended for full decision and condition coverage as well as for MC/DC. If you follow these coding guidelines you avoid false negative results. False negative means that a decision/conditions is tagged as incomplete although coverage has already been achieved.

Nevertheless, it is possible that the compiler itself generates such constructs at high optimization levels.

Avoid Simple Decisions in Assignment Context

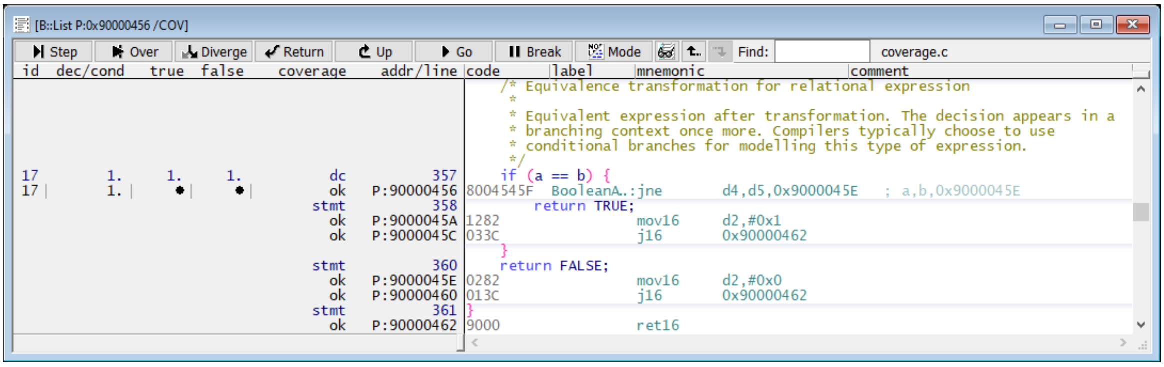
It is likely that these conditions are not represented by a conditional branch/instruction at object code level.

In this example no conditional branch/instruction was generated for the condition `a==b`.



```
id  dec/cond   true  false     coverage     addr/line code          label        mnemonic                    comment
                                                      {
                                                          /* Relational expression as decision
                                                           *
                                                           * Expression showing a decision in non-branching context. Compilers may
                                                           * choose to model Boolean assignments with conditional or unconditional
                                                           * instructions instead of conditional branches that are not suitable for
                                                           * the trace-based measurement of code coverage.
                                                           */
18         -       -      -  incomplete          373      return a == b;
                                     ok   P:9000044E 2100540B  BooleanA..:eq       d2,d4,d5        ; d2,a,b
                                     ok   P:90000452 013C                  j16     0x90000454
                                   stmt          374 }
                                     ok   P:90000454 9000                  ret16
```

It is recommended to write the source code in a way that ensures that the conditional branches/instructions required for the trace-based code coverage are generated.



```
id  dec/cond   true  false     coverage     addr/line code          label        mnemonic                    comment
                                                          /* Equivalence transformation for relational expression
                                                           *
                                                           * Equivalent expression after transformation. The decision appears in a
                                                           * branching context once more. Compilers typically choose to use
                                                           * conditional branches for modelling this type of expression.
                                                           */
17        1.      1.     1.          dc          357      if (a == b) {
17 |      1. |     ● |    ● |        ok   P:90000456 8004545F  BooleanA..:jne      d4,d5,0x9000045E   ; a,b,0x9000045E
                                   stmt          358          return TRUE;
                                     ok   P:9000045A 1282                  mov16   d2,#0x1
                                     ok   P:9000045C 033C                  j16     0x90000462
                                                          }
                                   stmt          360      return FALSE;
                                     ok   P:9000045E 0282                  mov16   d2,#0x0
                                     ok   P:90000460 013C                  j16     0x90000462
                                   stmt          361 }
                                     ok   P:90000462 9000                  ret16
```

A few examples:

```
; source code not suitable for          ; source code suitable for
; trace-based code coverage             ; trace-based code coverage

return a == b;                          if (a == b) {
                                            return TRUE;
                                        }
                                        return FALSE;
```

```
; source code not suitable for         ; source code suitable for
; trace-based code coverage            ; trace-based code coverage

identity(a != b);                      tmp = FALSE;
                                       if (a != b) {
                                            tmp = TRUE;
                                       }
                                       identity(tmp);
```

```
; source code not suitable for         ; source code suitable for
; trace-based code coverage            ; trace-based code coverage

return (a >= b) ? a : b;               if (a >= b) {
                                            return a;
                                       }
                                       return b;
```

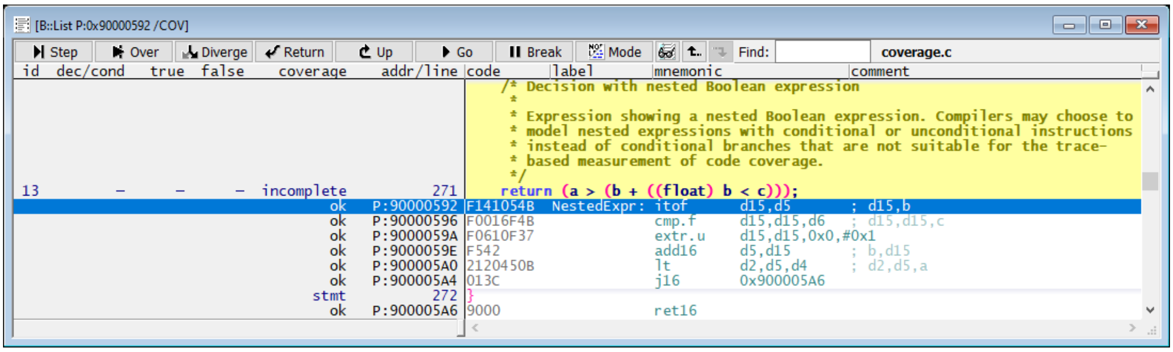
Avoid Nesting of Decisions

It is very likely that not all conditions are represented by a conditional branch/instruction at object code level.

This is illustrated by the following example:

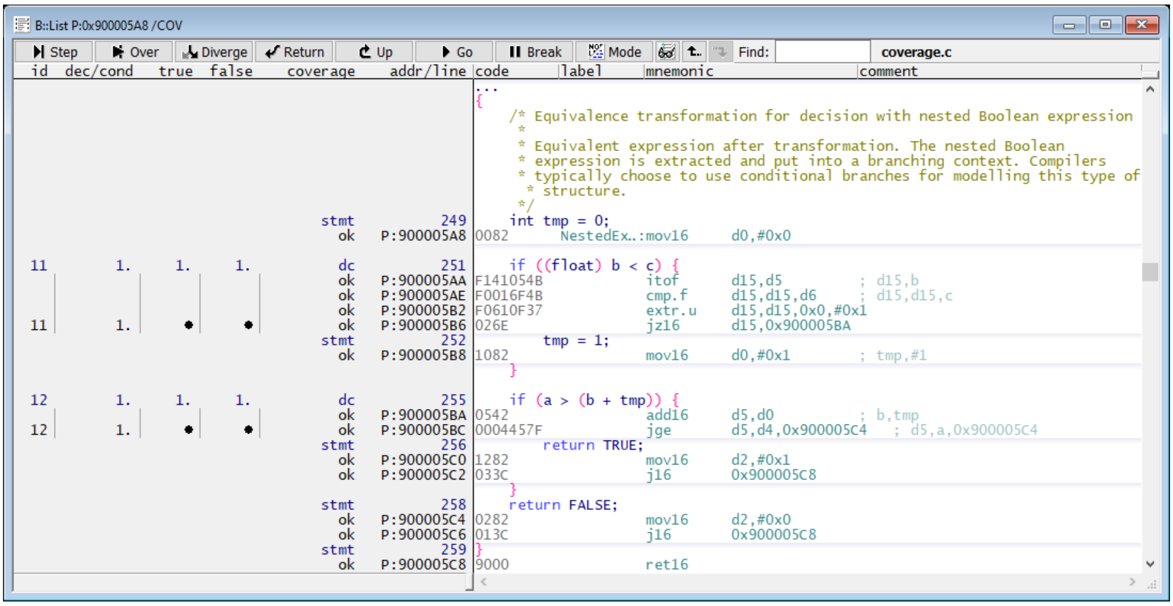
```
; source code not suitable for         ; source code suitable for
; trace-based code coverage            ; trace-based code coverage

return a > (b + (b && c));             if (b && c) {
                                            tmp = 1;
                                       }

                                       if (a > (b + tmp)) {
                                            return TRUE;
                                       }
                                       return FALSE;
```

In this example no conditional branches/instructions were generated for the conditions.



```
id  dec/cond   true  false    coverage    addr/line |code          |label      |mnemonic              |comment
                                                          /* Decision with nested Boolean expression
                                                           *
                                                           * Expression showing a nested Boolean expression. Compilers may choose to
                                                           * model nested expressions with conditional or unconditional instructions
                                                           * instead of conditional branches that are not suitable for the trace-
                                                           * based measurement of code coverage.
                                                           */
13          -     -     -  incomplete         271         return (a > (b + ((float) b < c)));
                                   ok  P:90000592 F141054B  NestedExpr: itof       d15,d5        ; d15,b
                                   ok  P:90000596 F0016F4B              cmp.f      d15,d15,d6    ; d15,d15,c
                                   ok  P:9000059A F0610F37              extr.u     d15,d15,0x0,#0x1
                                   ok  P:9000059E F542                  add16      d5,d15        ; b,d15
                                   ok  P:900005A0 2120450B              lt         d2,d5,d4      ; d2,d5,a
                                   ok  P:900005A4 013C                  j16        0x900005A6
                                 stmt         272 }
                                   ok  P:900005A6 9000                  ret16
```

If the code is written in a way that suits for trace-based code coverage, all necessary conditional branches/instructions were generated.



```
id  dec/cond   true  false    coverage    addr/line |code          |label      |mnemonic              |comment
                                                     ...
                                                     {
                                                         /* Equivalence transformation for decision with nested Boolean expression
                                                          *
                                                          * Equivalent expression after transformation. The nested Boolean
                                                          * expression is extracted and put into a branching context. Compilers
                                                          * typically choose to use conditional branches for modelling this type of
                                                           * structure.
                                                          */
                                 stmt         249        int tmp = 0;
                                   ok  P:900005A8 0082        NestedEx..:mov16      d0,#0x0

11  |       1. |    1. |    1. |   dc         251        if ((float) b < c) {
    |          |       |       |   ok  P:900005AA F141054B            itof       d15,d5        ; d15,b
    |          |       |       |   ok  P:900005AE F0016F4B            cmp.f      d15,d15,d6    ; d15,d15,c
    |          |       |       |   ok  P:900005B2 F0610F37            extr.u     d15,d15,0x0,#0x1
11  |       1. |     ● |     ● |   ok  P:900005B6 026E                jz16       d15,0x900005BA
                                 stmt         252            tmp = 1;
                                   ok  P:900005B8 1082                mov16      d0,#0x1       ; tmp,#1
                                                         }

12  |       1. |    1. |    1. |   dc         255        if (a > (b + tmp)) {
    |          |       |       |   ok  P:900005BA 0542                add16      d5,d0         ; b,tmp
12  |       1. |     ● |     ● |   ok  P:900005BC 0004457F            jge        d5,d4,0x900005C4   ; d5,a,0x900005C4
                                 stmt         256            return TRUE;
                                   ok  P:900005C0 1282                mov16      d2,#0x1
                                   ok  P:900005C2 033C                j16        0x900005C8
                                                         }
                                 stmt         258        return FALSE;
                                   ok  P:900005C4 0282                mov16      d2,#0x0
                                   ok  P:900005C6 013C                j16        0x900005C8
                                 stmt         259    }
                                   ok  P:900005C8 9000                ret16
```

Appendix G: Object Code Coverage Tags in Detail

Standard Tags

Standard tagging applies to all core architectures and all trace protocols. The only exception are Arm/Cortex cores that use the protocols Arm-ETMv1 or Arm-ETMv3, as well as Arm-ETMv4. However, for the Arm-ETMv4 protocol, this only applies if no trace information about the execution of conditional non-branch instructions is generated in order to save bandwidth (command **ETM.COND OFF**).

The following tags are used for object code coverage tagging:

Tag	Tagging object	Description
ok	conditional branch	The conditional branch has be at least once *taken* and *not taken*.
	conditional instruction	The object code instruction has been executed at least once with its condition code true and once with its condition code false.
	all other object code instructions	The object code instruction has been executed at least once.
taken	conditional branch	The conditional branch has be at least once *taken*, but never *not taken*.
	conditional instruction	The object code instruction has been executed at least once with its condition code true, but never with its condition code false.
not taken	conditional branch	The conditional branch has be at least once *not taken*, but never *taken*.
	conditional instruction	The object code instruction has been executed at least once with its condition code false, but never with its condition code true.
never	all object code instructions	The object code instruction has never been executed.

The following tags apply for analysis at the source code, function or module level:

Tag	Tagging object	Description
ok	range of object code instructions	All object code instructions within the range are tagged with ok.
partial	range of object code instructions	Not all object code instructions within the range are tagged with ok.
branches	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *taken* and one that is only *not taken*.
taken	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *taken*.
not taken	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *not taken*.
never	range of object code instructions	Not a single object code instruction within the range has been executed.

Tags for Arm-ETMv1/v3/v4 for Arm/Cortex Architecture

The following tags are used for object code coverage tagging:

Tag	Tagging object	Description
ok	conditional branch	The conditional branch has be at least once taken and not taken.
	conditional instruction	The object code instruction has been executed at least once with its condition code true and once with its condition code false.
	all other object code instructions	The object code instruction has been executed at least once.

Tag	Tagging object	Description
only exec	conditional branch	The conditional branch has be at least once taken, but never not taken.
	conditional instruction	The object code instruction has been executed at least once with its condition code true, but never with its condition code false.
not exec	conditional branch	The conditional branch has be at least once not taken, but never taken.
	conditional instruction	The object code instruction has been executed at least once with its condition code false, but never with its condition code true.
never	all object code instructions	The object code instruction has never been executed.

The following tags apply for analysis at the source code, function or module level:

Tag	Tagging object	Description
ok	range of object code instructions	All object code instructions within the range are tagged with ok.
partial	range of object code instructions	Not all object code instructions within the range are tagged with ok.
cond exec	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *only exec* and one that is only *not exec*.
only exec	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *only exec*.
not exec	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *not exec*.
never	range of object code instructions	Not a single object code instruction within the range has been executed.

Appendix E: Data Coverage in Detail

The data addresses are tagged as follow:

readwrite	The data address was read at least once and written at least once.
read	The data address has been read at least once.
write	The data address has been written at least once.
never	The data address was neither read nor written

Each static variable occupies a fixed address range. This results in the following tagging for variables:

readwrite	Read and write accesses were performed for all addresses within the address range of the variable.
read	Only read accesses were performed for all addresses within the address range of the variable.
write	Only write accesses were performed for all addresses within the address range of the variable.
p-write	Write accesses were performed only to a part of the address range of the variable. No read accesses were performed.
p-read	Read accesses were performed only to a part of the address range of the variable. No write accesses were performed.
p-wr read	Write accesses were performed only to a part of the address range of the variable. Read accesses were performed for all addresses.
p-rd write	Read accesses were performed only to a part of the address range of the variable. Write accesses were performed for all addresses.
p-rd p-wr	Both read and write accesses were performed only to a part of the address range of the variable.
never	Not a single address of the address range of the variable was read or written.

rdwr ok	The address range achieved full object code coverage, and at least one read and one write access occurred to address range.
write ok	The address range achieved full object code coverage, and at least one write access occurred to address range.

read ok	The address range achieved full object code coverage, and at least one read access occurred to address range.
partial	The address range did not achieve full object code coverage. The amount of read and write accesses that have taken place is not further specified.

The coverage status of **HLL source code statements** that have associated data values is indicated by the following tags if a **data trace** is available:

- **rdwr ok**: The HLL source code statement(s) have been fully covered. All associated assembly instructions have been fully covered and at least one read and write access to the data values has been recorded.
- **write ok**: The HLL source code statement(s) have been fully covered. All associated assembly instructions have been fully covered and at least one write access to the data values has been recorded.
- **read ok**: The HLL source code statement(s) have been fully covered. All associated assembly instructions have been fully covered and at least one read access to the data values has been recorded.
- **partial**: The HLL source code statement(s) have not been fully covered. At least one of the associated assembly instructions has not been fully covered. The amount of read and write accesses that have taken place is not further specified.
- **readwrite**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been read and written at least once.
- **write**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been written at least once and not read.
- **read**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been read at least once and not written.
- **p-rd write**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been written at least once. In addition at least one data value has been read.
- **p-wr read**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been read at least once. In addition at least one data value has been written.
- **p-rd p-wr**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and at least one of the data values has been read and one written.
- **p-write**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and at least one of the data values has been written.
- **p-read**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and at least one of the data values has been read.
- **never**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and neither read nor write accesses to the data values have been recorded.