LAUTERBACH
DEVELOPMENT TOOLS

Application Note for Trace-Based Code Coverage

MANUAL

Application Note for Trace-Based Code Coverage

Application Note for Trace-Based Code Coverage

Version 10-Mar-2025

History

20-Feb-2025	Added 'Keep Access to Measurement Settings for Later Review' chapter with a table linking measurement settings to code coverage metrics.
13-Feb-2025	Chapter 'Code Coverage Evaluation in TRACE32' was updated.
11-Feb-2025	The chapter 'Continuous Code Coverage Measurement in SYP Mode' was updated.
29-Jan-2025	In the chapter 'Code Coverage Measurement', the measurement details for both incremental code coverage variants and continuous code coverage in RTS mode updated.
17-Jan-2025	Chapter 'Steps in Preparation for Code Coverage Measurement' updated.
17-Jan-2025	Chapter 'Selecting the Right Code Coverage Measurement Variant' updated. RTS code coverage now supports all source metrics. Chapter 'TRACE32 Trace Tool solutions for code coverage' added.
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.

- **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.

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.

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. The code coverage measurement 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. Consequently, TRACE32's trace-based code coverage cannot be conducted on production code.

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. This TRACE32's lightweight instrumentation has only a minimal impact on code size and timing behavior.

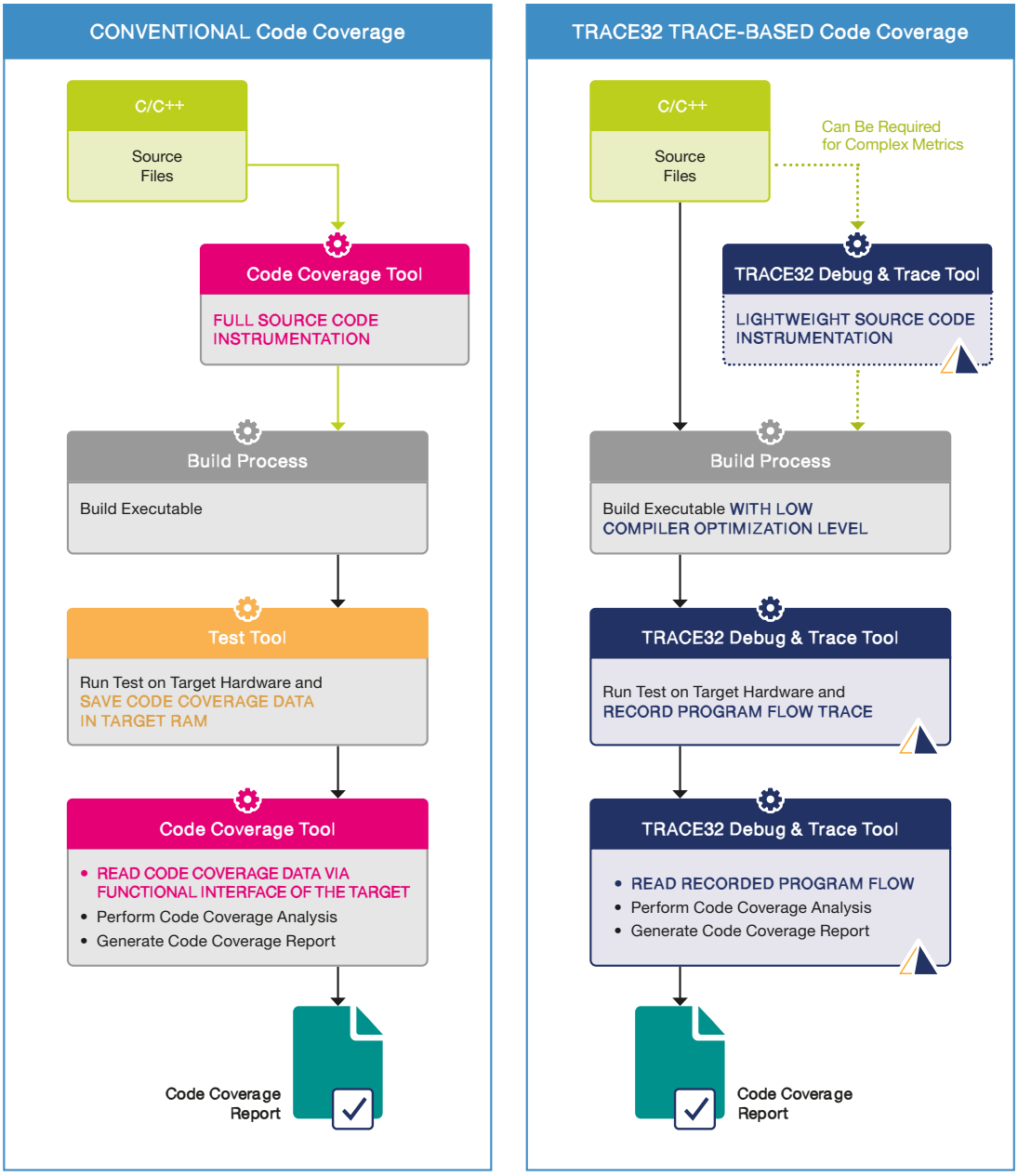


Figure: Workflow comparison, conventional code coverage vs. TRACE32 trace-based code coverage.

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. The safety standards allow this for the test phases software unit and module integration testing. See also **TRACE32 Instruction Set Simulator and ISO 26262**.

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 approaches for measuring code coverage:

Incremental Code Coverage

This method follows a two-step process—RECORDING and PROCESSING—repeated in sequence until sufficient data is collected.

RECORDING: The program is executed, and its flow is captured in the trace memory. Program execution is halted when the trace memory becomes full.

PROCESSING: The trace data is read, code coverage is calculated, and the results are saved in the TRACE32 Code Coverage System.

Continuous Code Coverage

This method caluclates code coverage while the program is running. It requires processors or chips that support off-chip trace. The following tasks occur simultaneously:

- Execute the program and record the program flow in the trace memory.
- Stream the recorded program flow to the host.
- Calculate the code coverage results and save them in the TRACE32 Code Coverage System.

Continuous code coverage is faster and easier to set up since processing steps run simultaneously. However, it is only effective up to a certain bandwidth limit. In contrast, incremental code coverage is slower and requires more complex scripts due to its sequential nature. Nevertheless, it is universally compatible with any TRACE32 trace solution.

Evaluation of the Code Coverage Measurements

TRACE32 provides two approaches for evaluating code coverage:

- **Evaluation in a Web Browser**

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

  Code coverage is typically achieved incrementally rather than in a single test run. To create a consolidated report, multiple measurements must be merged. This involves two steps:

  - Export each individual code coverage result from the TRACE32 Coverage System to a JSON file. (The JSON file contains source code level coverage results but does not include object code.)
  - Merge the JSON files from different measurements and generate an HTML file to evaluate the desired code coverage metrics.

- **Evaluation in TRACE32**

  For object code-based coverage metrics—such as object code coverage and object code-based decision coverage—individual test results must be merged and analyzed directly within the TRACE32 PowerView GUI.

The TRACE32 Code Coverage System does not include track which core executed a specific object code instruction.

AMP Debug Configuration

Typically, each TRACE32 instance independently measures and evaluates code coverage, with a separate final report generated for each instance.

For code coverage metrics such as statement coverage, decision coverage, condition coverage, modified condition/decision coverage (MC/DC), call coverage, and function coverage, it is possible to merge data from multiple TRACE32 instances when the test scenario requires it. However, it's important to note that the code coverage data does not specify which TRACE32 instance or core executed a particular object code instruction.

SMP/iAMP Debug Configuration

For setups where multiple cores are debugged within a single TRACE32 instance, the TRACE32 Code Coverage System includes all relevant details to map object code to the corresponding source code. This includes identifiers such as **OS space ID, hypervisor machine ID**, and **iAMP machine ID**, ensuring a comprehensive overview of source code execution.

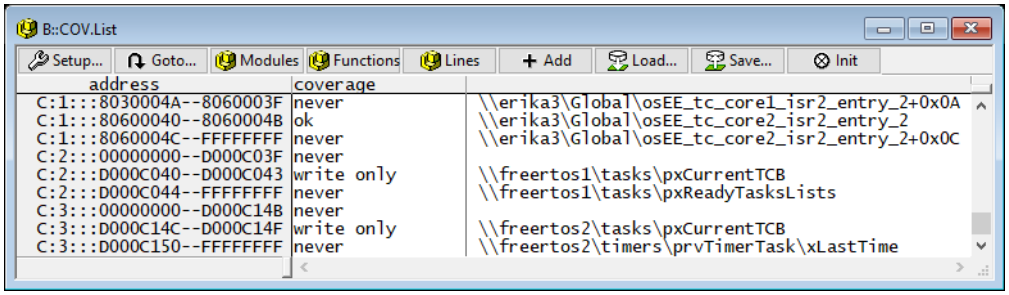


Figure: The screenshot displays iAMP machine IDs within the TRACE32 Code Coverage System.

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, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 126.

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.

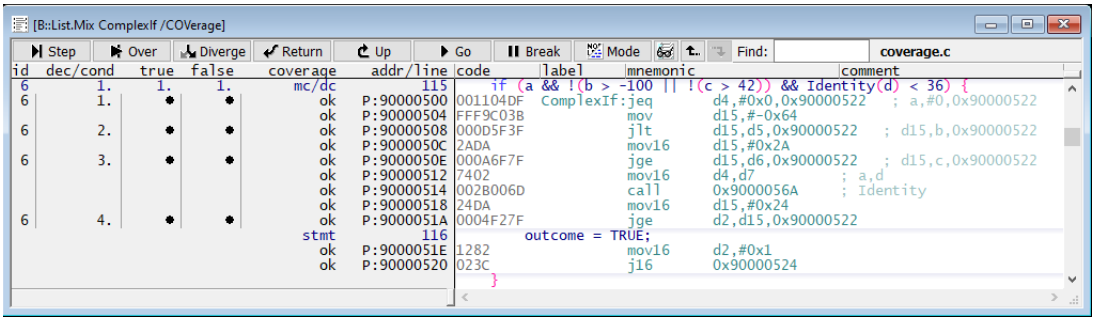


Figure: This screenshot illustrates the mapping between conditional branches in the object code, tagged as derived from the program flow trace, to the respective source code lines representing a decision, thereby tagging the decision line for MC/DC coverage.

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 143 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 21.

  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.

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

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 17** 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 17** 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 17** 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 17** 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 17** 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

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

Equivalent if-then expression

```
if (color == RED) {
    offset = 10;
}
else if (color == BLUE) {
    offset = 8;
}
else if (color == ORANGE) {
    offset = 6;
}
else if (color == YELLOW) {
    offset = 2;
}
else if (color == GREEN) {
    offset = 2;
}
else {
    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.

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.

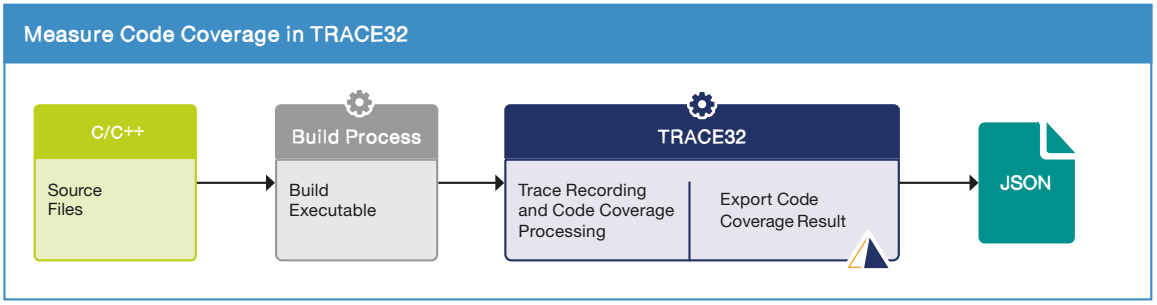


Figure: After generating the appropriate executable, code coverage measurement can be conducted in TRACE32. The resulting data must be exported as a JSON file.

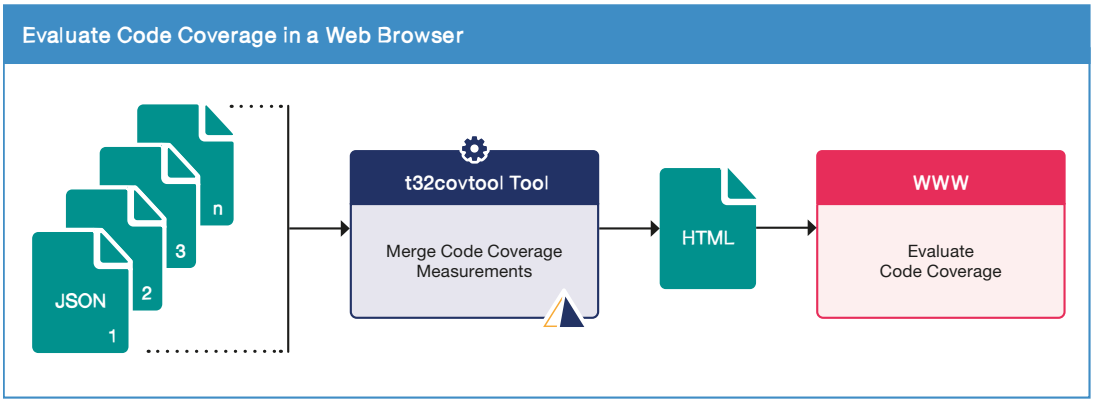


Figure: The data resulting from multiple code coverage measurements can be summarized in an HTML file and intuitively evaluated in a web browser.

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 code coverage measurement variant**

   Choose the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, 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 and Object Code Coverage"**, page 72.

4. **Set up and execute code coverage measurement**

   Configure the code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.
   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

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 **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

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 instruction 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 instruction 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 SMP Debugging, page 139 (training_debugger_smp.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 with stmt.

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

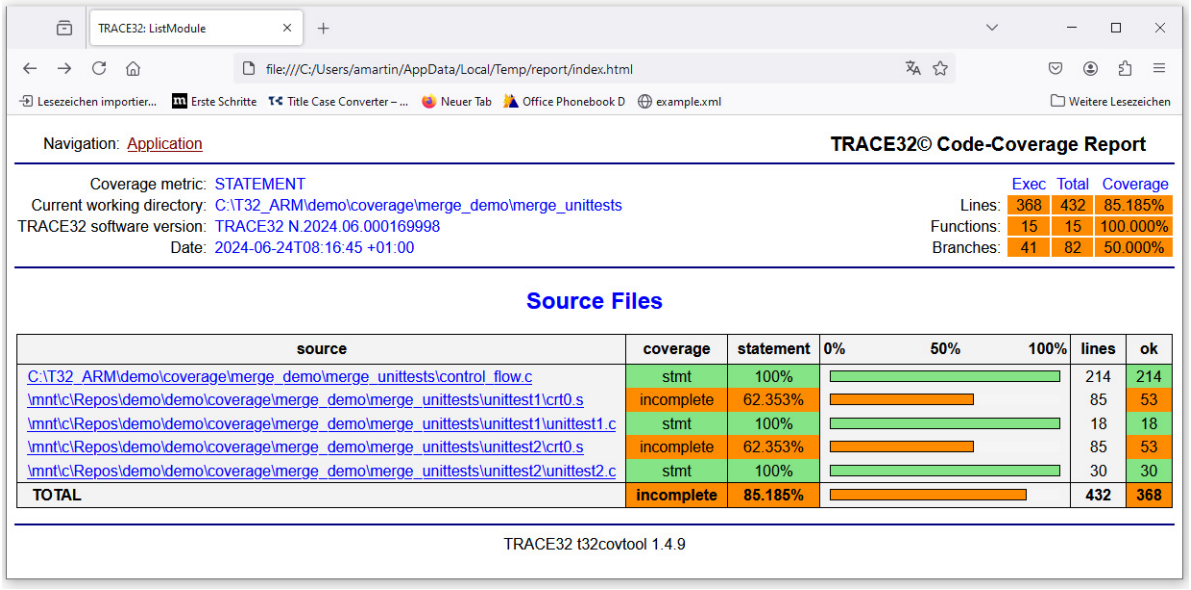


Figure: Statement coverage evaluation in a web browser.

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.

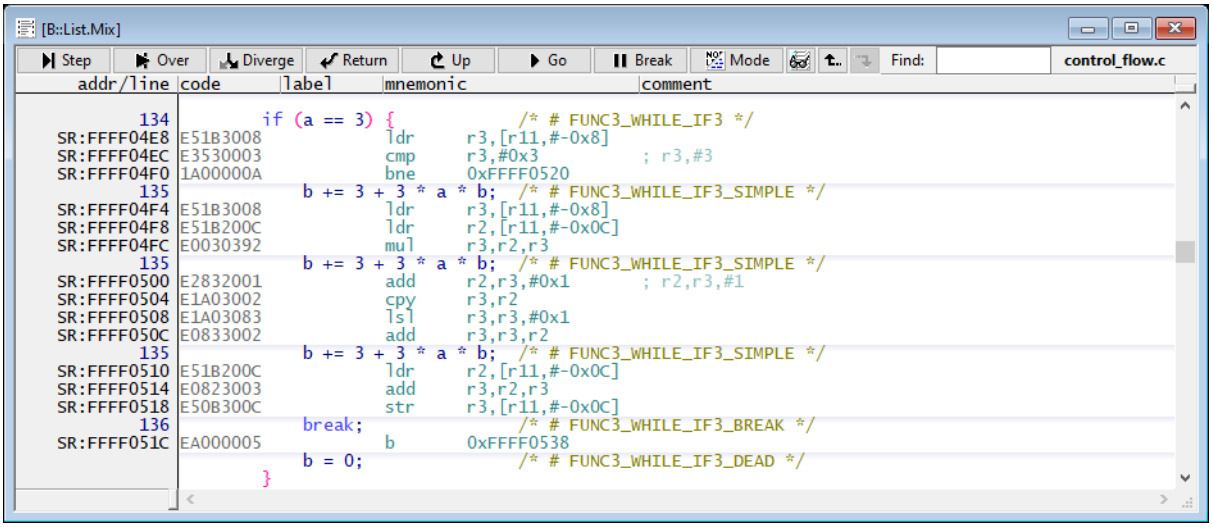


Figure: In TRACE32, the source code statement `b = 0;` does not have line numbers.

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.

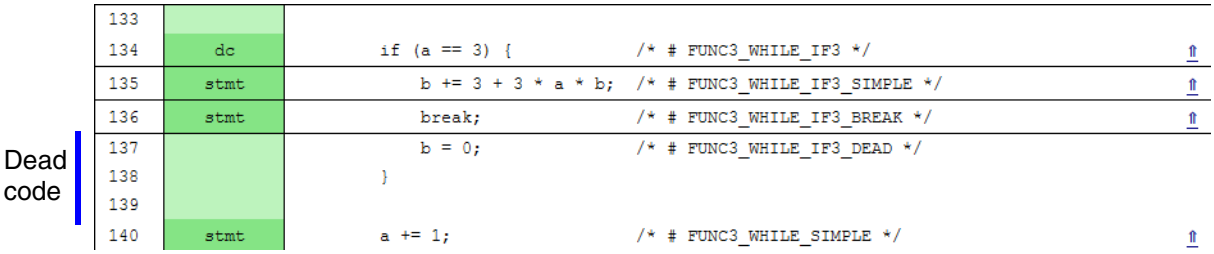


Figure: In the TRACE32 Code Coverage Report the statements `b = 0;` is displayed along with the next statement. It is shown in a lighter color.

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.

Before starting the evaluation for condition coverage, it is recommended to review chapter **"MC/DC, Condition and Decision Coverage"**, page 17.

To perform a condition coverage evaluation, follow these steps.

1. **Build the executable**

   When performing condition coverage, 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 18. Your choice of mode will depend on your application specifics. Refer to **"Decide on the Appropriate Code Coverage Mode"**, 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 **"General Recommendations for the Build Toolchain"**, page 43.

2. **Choose a code coverage measurement variant**

   Choose the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, page 60 for assistance in decision-making.

3. **Load relevant files into TRACE32**

   Load the files relevant for condition coverage into TRACE32, see **"Preparation for MC/DC, Condition and Decision Coverage"**, page 74. 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 code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.
   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

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 **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107

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.

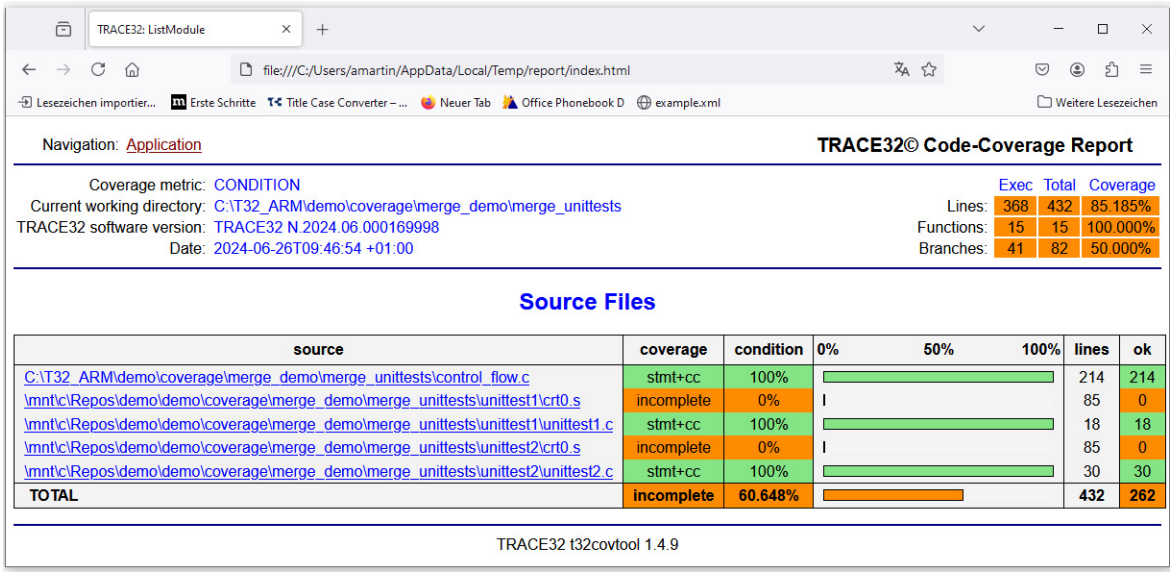


Figure: Condition coverage evaluation in a web browser.

Before starting the evaluation for decision coverage, it is recommended to review chapter **"MC/DC, Condition and Decision Coverage"**, page 17.

To perform a decision coverage evaluation, follow these steps.

1. **Build the executable**

   When performing decision coverage, 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 18. Your choice of mode will depend on your application specifics. Refer to **"Decide on the Appropriate Code Coverage Mode"**, 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 **"General Recommendations for the Build Toolchain"**, page 43.

2. **Choose a code coverage measurement variant**

   Choose the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, 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 74. 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 code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.

   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

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 **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

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.

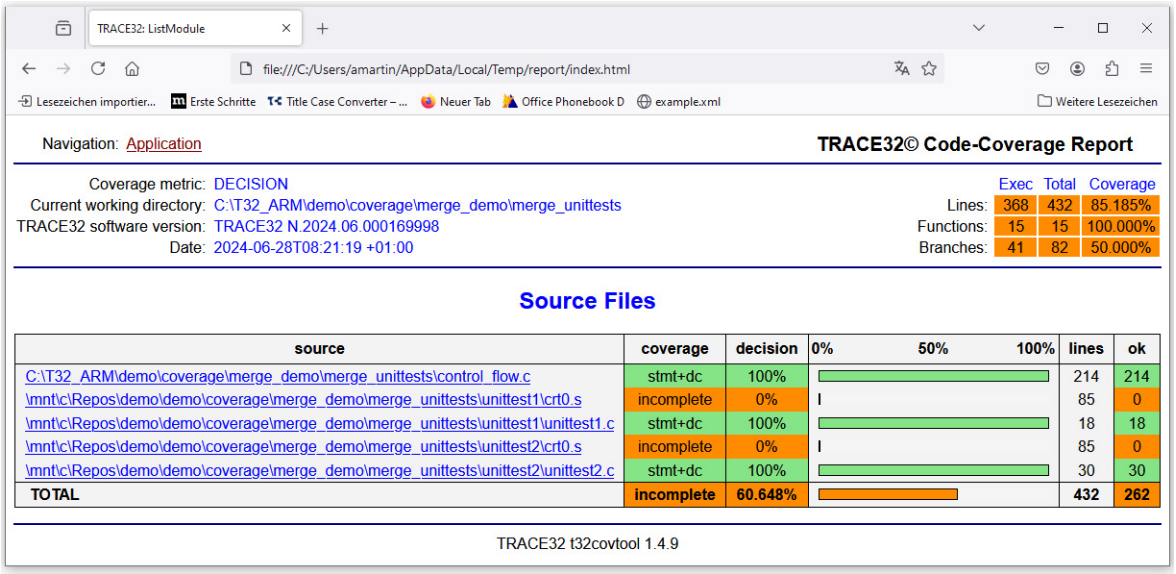


Figure: Decision coverage evaluation in a web browser.

Before starting the evaluation for Modified Condition/Decision Coverage, it is recommended to review chapter **"MC/DC, Condition and Decision Coverage"**, page 17.

To perform MC/DC, follow these steps.

1. **Build the executable**

   When performing MC/DC, 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 18. Your choice of mode will depend on your application specifics. Refer to **"Decide on the Appropriate Code Coverage Mode"**, 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 **"General Recommendations for the Build Toolchain"**, page 43.

2. **Choose a code coverage measurement variant**

   Choose the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, 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 74. 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 code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.

   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

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 **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107

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.

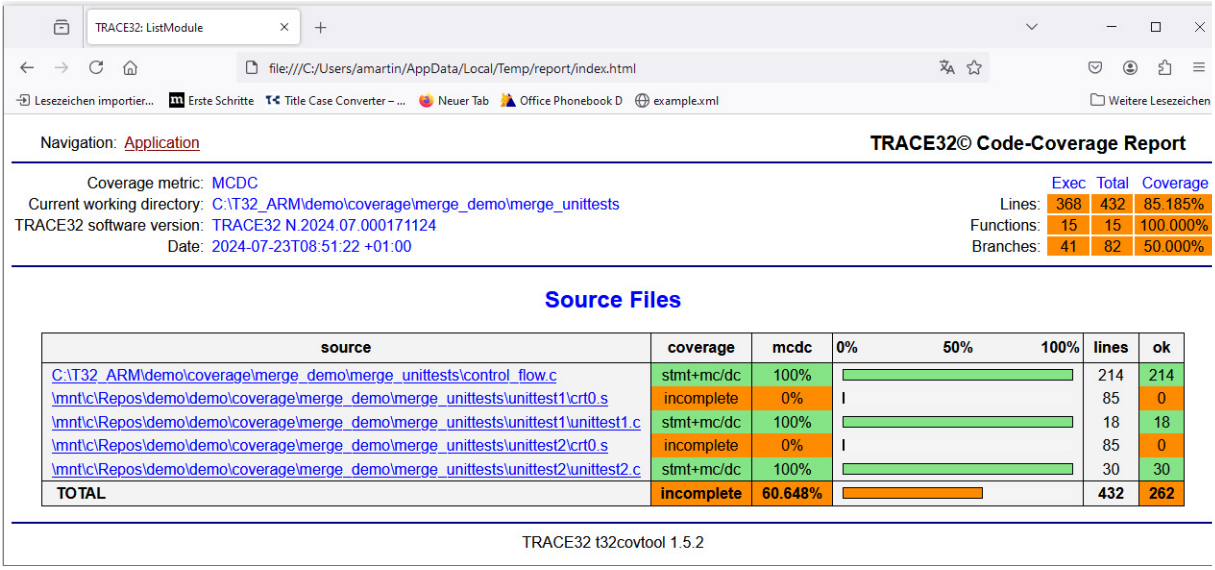


Figure: MC/DC evaluation in a web browser.

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

2. **Choose a code coverage measurement variant**

   Choose the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, 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 and Object Code Coverage"**, page 72.

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

   Configure the code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.

   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

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 **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

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.

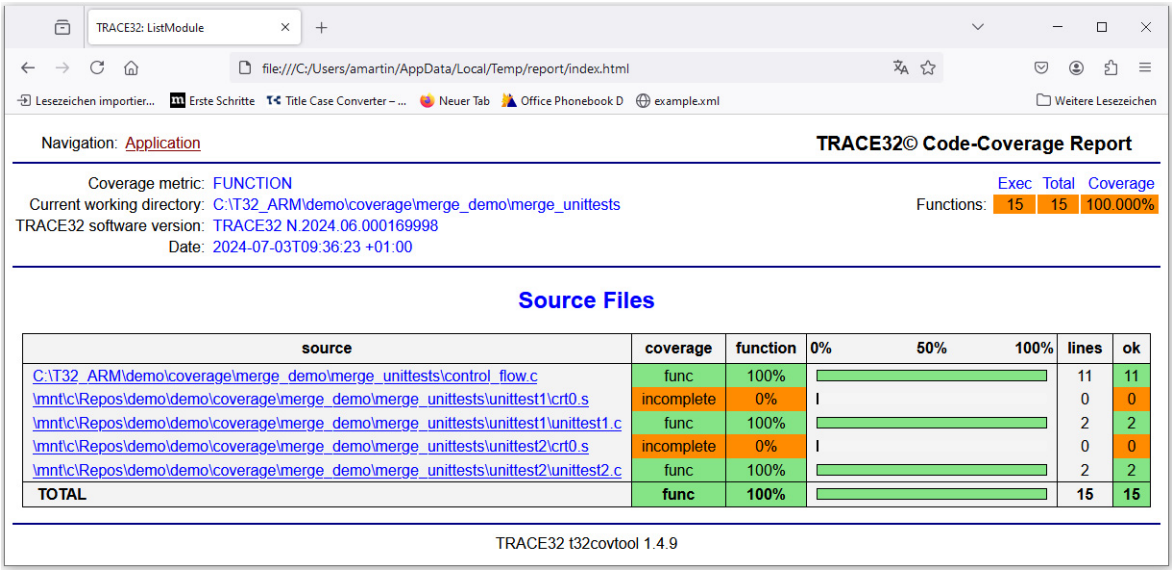


Figure: Function coverage evaluation in a web browser.

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

2. **Choose a code coverage measurement variant**

   Choose the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, 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 73.

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

   Configure the code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.

   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

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 **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

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.

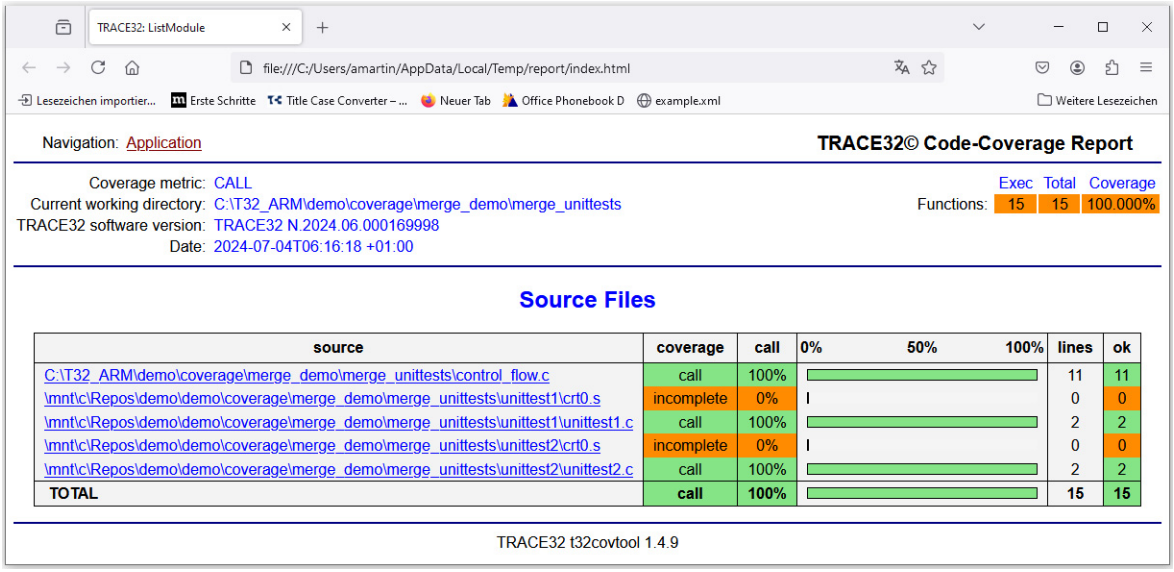


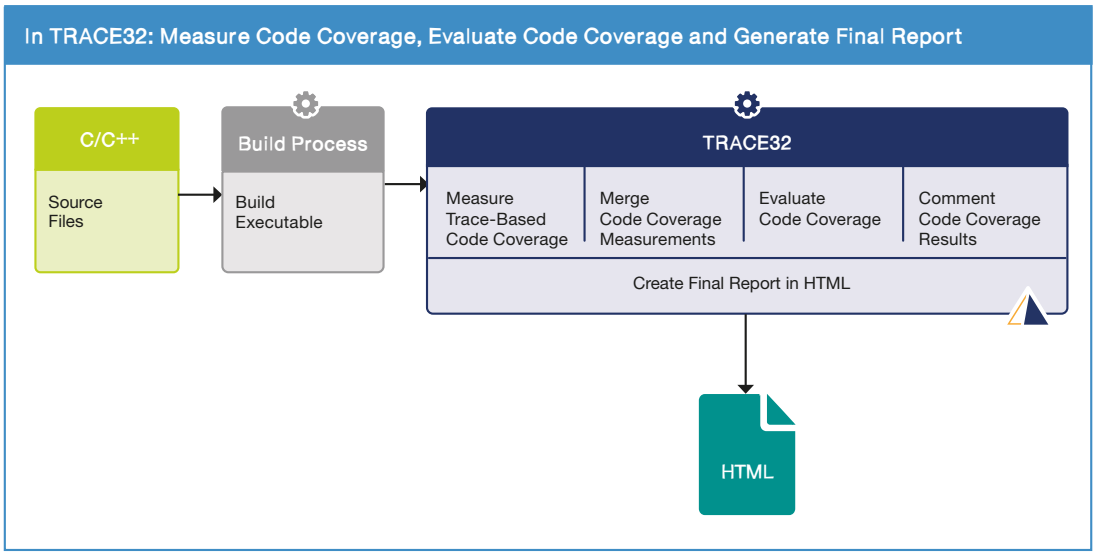
Figure: Call coverage evaluation in a web browser.

Workflows for Object Code Metrics

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

General Procedure

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



To perform a object code 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 code coverage measurement variant**

   Select the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, 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 and Object Code Coverage"**, page 72.

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

   Configure the code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.

   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

5. **Merge 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: Merge Multiple Object Code Based Measurements"**, page 128.

6. **Evaluate object code coverage.**

   Evaluation details can be found at **"Object Code Coverage"**, page 110.

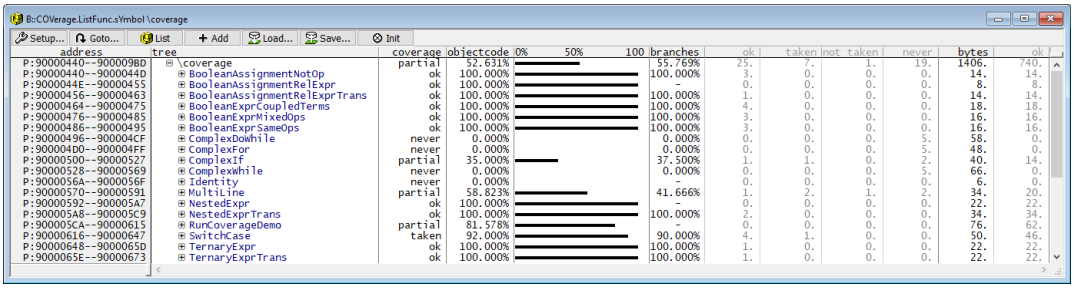


Figure: Object code coverage evaluation in TRACE32.

7. **Comment uncovered code**

   Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 124.

8. **Generate final HTML report**

   Generate your final code coverage report as described **"Appendix A: TRACE32 Coverage Report Utility"**, page 126.

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

   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 143.

2. **Choose a code coverage measurement variant**

   Select the variant that best fits your test scenario. Refer to **"Selecting the Right Code Coverage Measurement Variant"**, 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 and Object Code Coverage"**, page 72.

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

   Configure the code coverage measurement variant selected in step 2, then perform the measurement.

   - For general information on trace configuration and recording, refer to **"Best Practices for Trace Recording"**, page 65.

   - Detailed instructions for configuring the selected code coverage variant and performing the measurement can be found in **"Code Coverage Measurement"**, page 83.

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

   Ensure you only merge test runs carried out with the identical executable(s). Instructions for this process can be found in **"Appendix B: Merge Multiple Object Code Based Measurements"**, page 128.

6. **Evaluate object code coverage**

   Evaluation details can be found at **"Object Code Based (ocb) Decision Coverage"**, page 116.

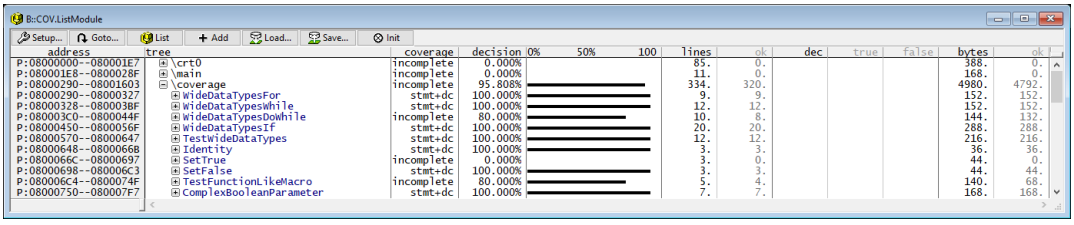


Figure: ocb decision coverage evaluation in TRACE32.

7. **Comment uncovered code**

   Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 124.

8. Generate final HTML report

Generate your final code coverage report as described **“Appendix A: TRACE32 Coverage Report Utility”**, page 126.

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.

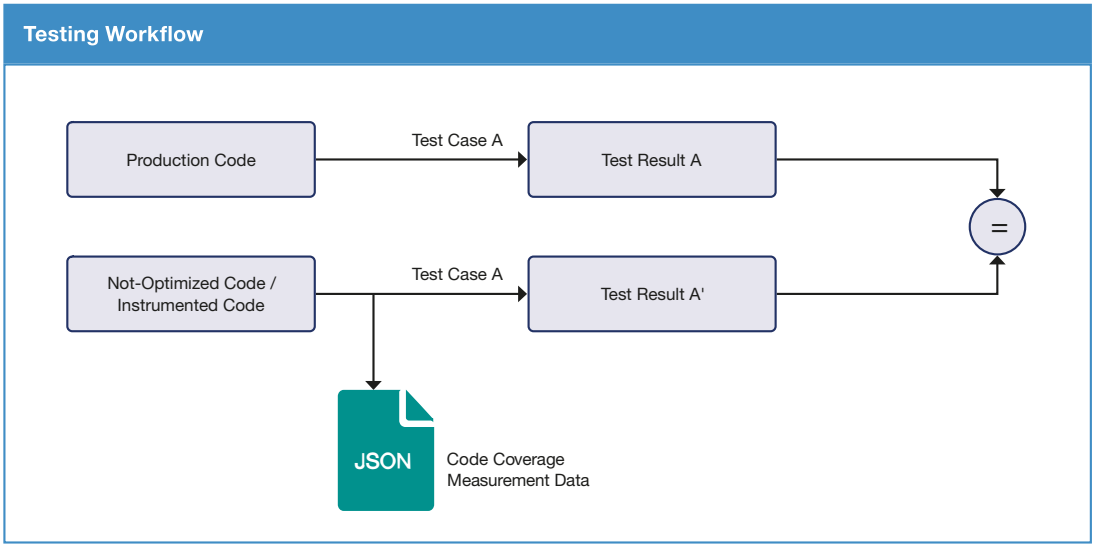
- Source code instrumentation

  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	—	—

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.



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.

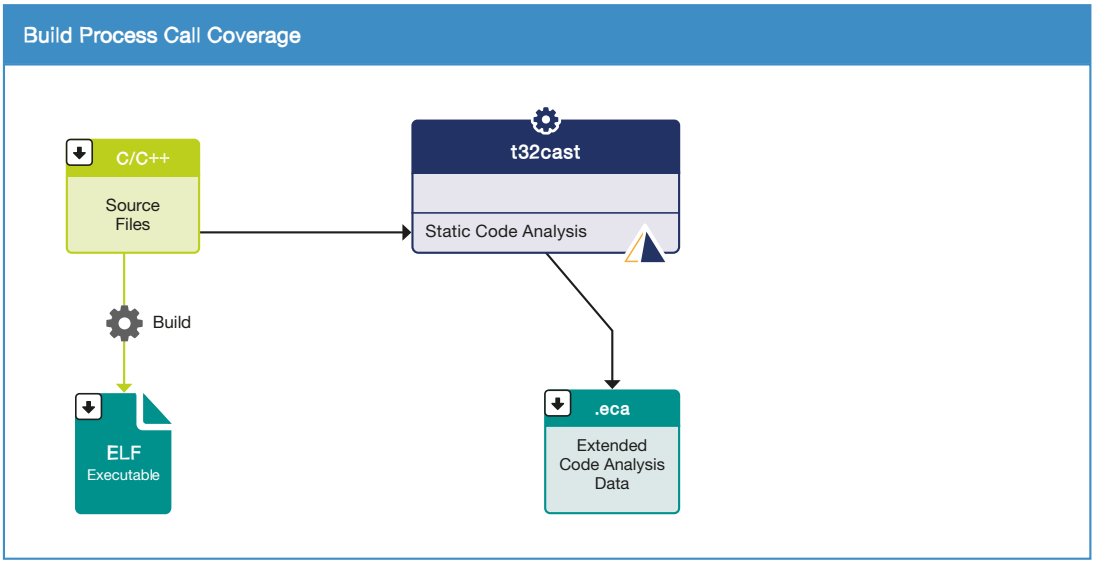


Figure: Build process for call coverage; all input/outputs of the build process that need to be loaded to TRACE32 for call coverage measurement are marked in this figure with an arrow pointing downwards.

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

- **"Build Process 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.

Decide on the Appropriate Code Coverage Mode

As detailed in **"Multiple Code Coverage Modes"**, page 17, 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.

Decision Making Process

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
   ```

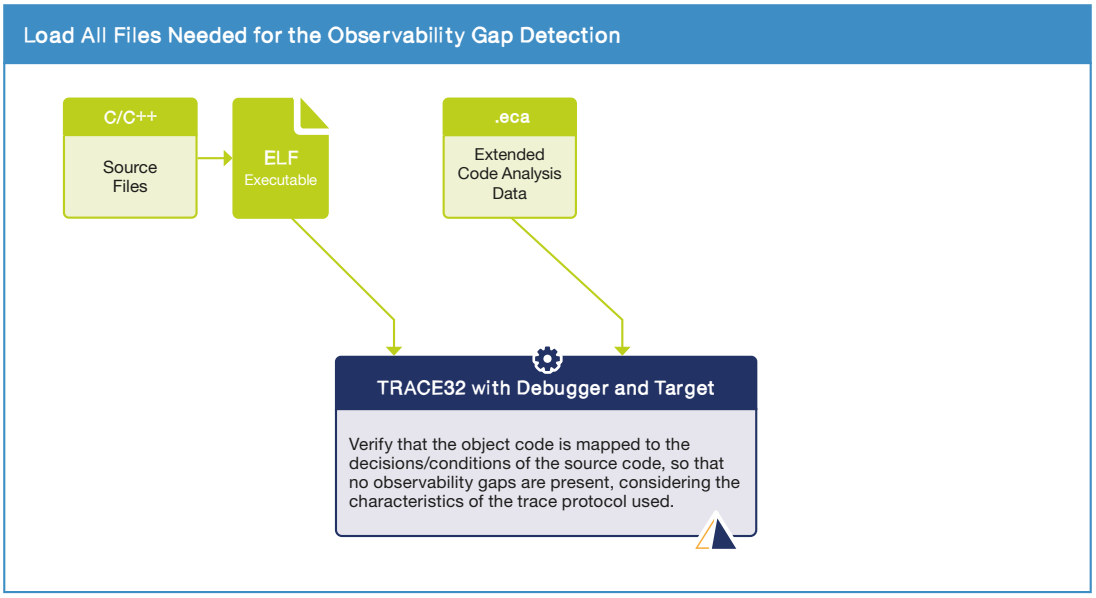
   More details can be found in **"Command Line Parameters of t32cast"** in Application Note for t32cast, page 10 (app_t32cast.pdf).

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



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.

TRACE32 detects the trace protocol based on the selected CPU (**SYStem.CPU()**). For trace-based code coverage, conditional branches and instructions are particularly important. The protocol usually determines whether it generates information about conditional instructions in addition to branches—

this is typically not configurable. The only exception is ETMv4 for Cortex-M/R cores, where visibility of conditional non-branch instructions can be enabled using **ETM.COND ALL**. We recommend enabling this setting to reduce observability gaps.

```
; ETMv4 only
; ETM.COND ALL

; 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
```

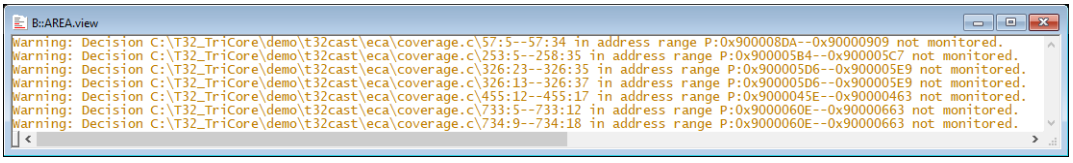


Figure: A warning is displayed in the message area for each condition/decision where no mapping can be established between the source code and the object code.

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

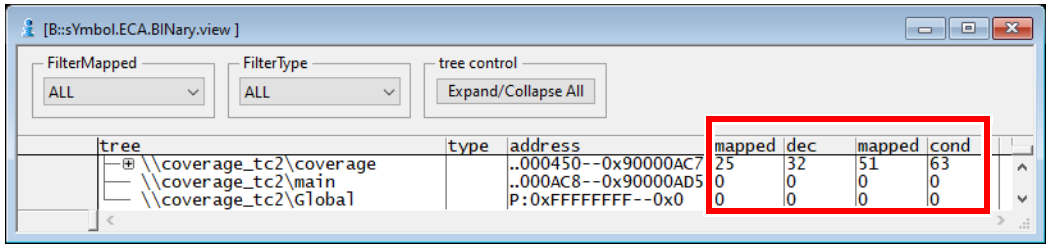


Figure: The **mapped/dec** columns indicate the number of decisions in the software module (dec) and how many were successfully mapped. The **mapped/cond** columns show the number of conditions (cond) in the software module and how many were successfully mapped.

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 17.

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 **"Decision Making Process"**, 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 for Cortex-M and Cortex-R No for PTM and ETMv4.0 for Cortex-A and other Arm Cores [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 Code Coverage Mode — Targeted Instrumentation/No Instrumentation

In addition to the C/C++ source files, TRACE32 requires the following inputs for code coverage measurement. These files must be generated by the build process:

- 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

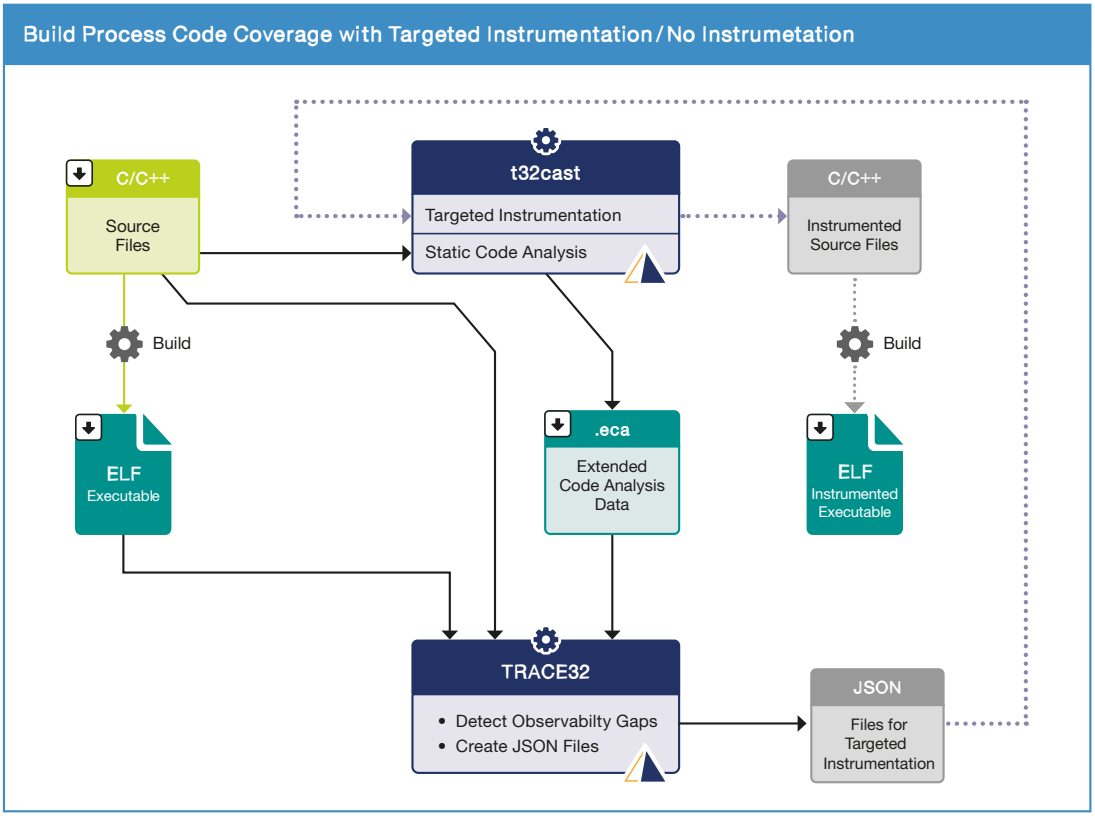


Figure: All inputs/outputs of the build process that may need to be loaded into TRACE32 for coverage measurement in code coverage mode target instrumentation/no instrumentation are indicated in this figure by a downward-pointing arrow.

The proposed build process always generates both a non-instrumented and an instrumented executable. If no observability gaps are detected, the two executables will be identical.

To generate all the necessary files for TRACE32 code coverage measurement, the build process must be extended 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 prepare targeted instrumentation if required**

   Use TRACE32 to determine if an instrumented executable is needed. If so, generate the necessary supporting files. One approach to prepare targeted instrumentation is to call TRACE32 from the Makefile using a script, such as `export_gaps.cmm`, which handles all this. There are two ways to call TRACE32 in a testing environment:

   Interactive Connection Mode (with connection script, here `iss.cmm`)

   ```
   t32marm.exe -e ../common/iss.cmm -s ../common/export_gaps.cmm myelf.elf
   ```

   Classic Connection Mode (with config file, here `trace32.cfg`)

   ```
   t32marm.exe -c ../common/trace32.cfg -s ../common/export_gaps.cmm myelf.elf
   ```

   Please refer to **"Command Line Arguments for Starting TRACE32"** in TRACE32 Installation Guide, page 53 (installation.pdf) for details.

   A brief overview of the supporting files:

   Flag file

   The `og_detected.txt` flag file, located in the directory of the instrumented source files, is one method used during the code coverage measurement to track that an instrumented executable has been generated for code coverage mode targeted instrumentation. This is just one approach—you are free to implement an alternative method to track this.

   Files with observability gaps

   For each C/C++ source file, a corresponding JSON file is created that contains the observability gaps for that file. If a source file has no observability gaps, a dummy JSON file is still generated.

The script, in our example `export_gaps.cmm`, that runs in TRACE32 must include the following steps.

```
PRIVATE &elf_file
ENTRY %LINE "&elf_file"

PRIVATE &instrumented_tree
&instrumented_tree="./instrumented_tree"

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

; load the non instrumented elf executable
Data.LOAD.Elf "&elf_file"

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

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

; ETMv4 for Cortex-M and Cortex-R only
; ETM.COND ALL

; configure TRACE32 for observability gap detection
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; perform observabilty gap detection
sYmbol.ECA.BINary.PROCESS

; create flag file if observability gaps were detected
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
)
; generate JSON files for targeted instrumentation
sYmbol.ECA.BINary.EXPORT.AdJoinGAPS
```

3.A If no observabilty gaps where detected, the non instrumented executable `my_app.elf` can be used for the code coverage measurement.

3.B If observabilty gaps where detected, 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).

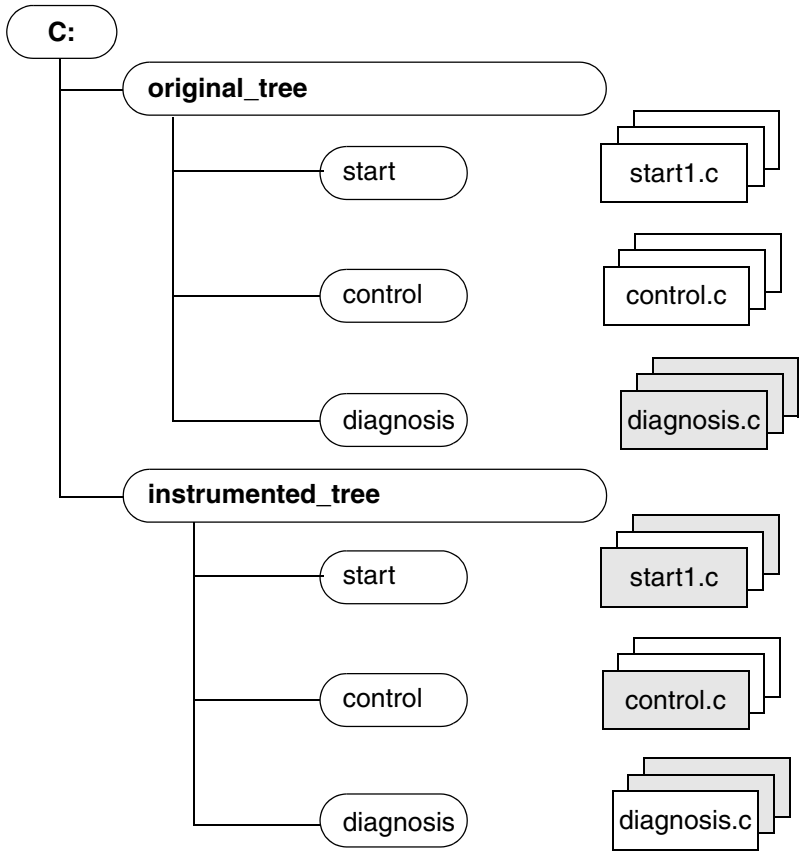


Figure: The instrumentation does not add any extra lines of source code. By preserving the structure of the **original_tree** directory in the **instrumented_tree** directory, TRACE32 can be configured to use the original, non-instrumented sources during testing.

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 as input for targeted instrumentation
; and instrument all decisions for which an observabiltiy gap was
; detected

; source files without observabiltiy gaps should be simply
; copied to instrumented_tree
t32cast instrument --mode=mcdc
--filter=original_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 Once the targeted instrumentation is complete, the instrumented executable must be generated.

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

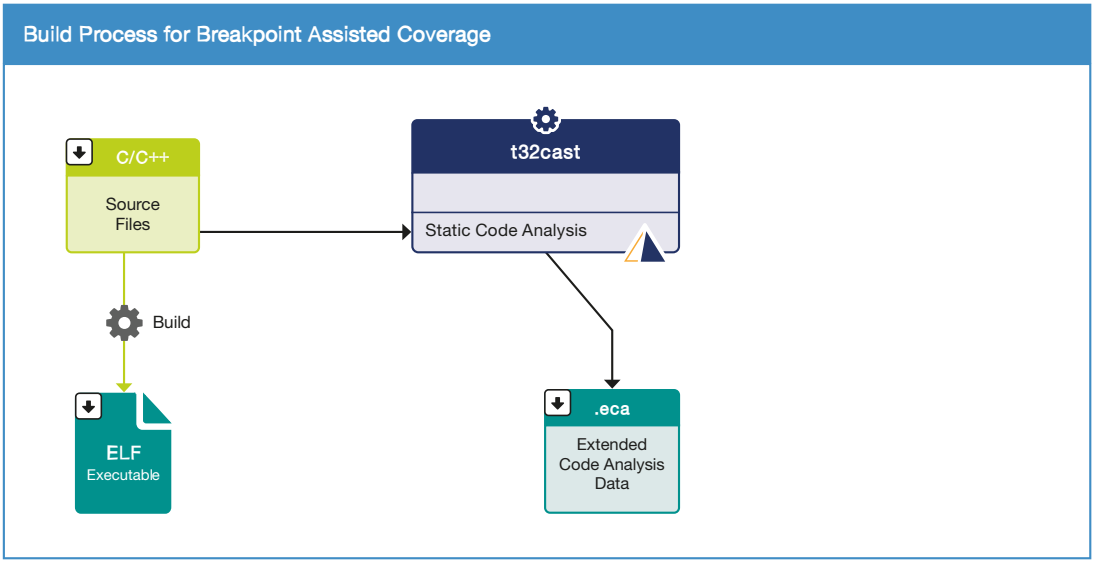


Figure: All input/outputs of the build process that are required for coverage measurement in breakpoint assisted code coverage mode are marked with an arrow pointing downwards.

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).

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

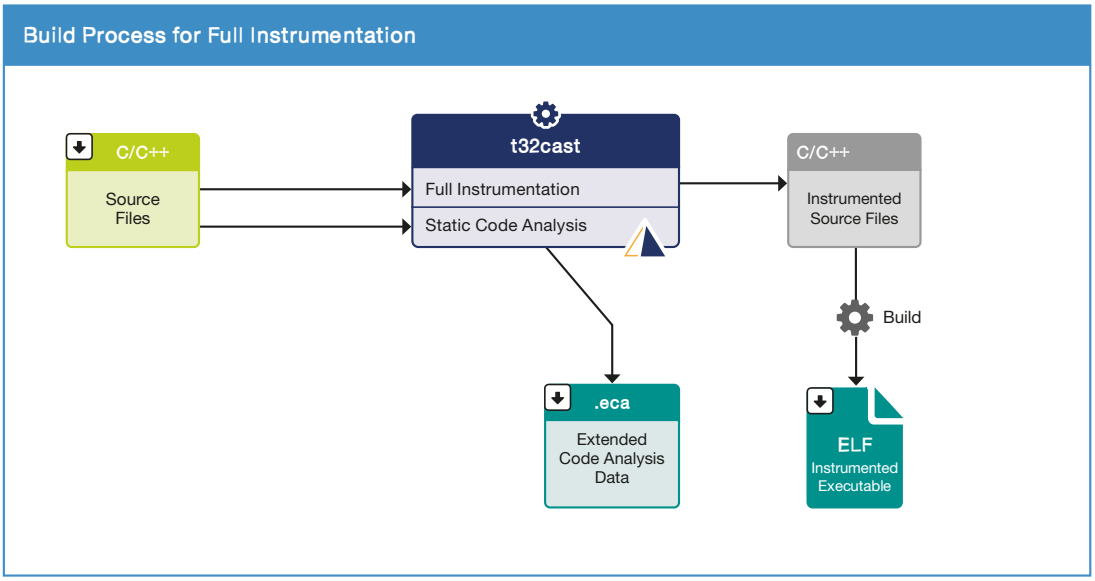


Figure: All input/outputs of the build process that are required for coverage measurement in code coverage mode full instrumentation are marked with an arrow pointing downwards.

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.

Selecting the Right Code Coverage Measurement Variant

Overview Table

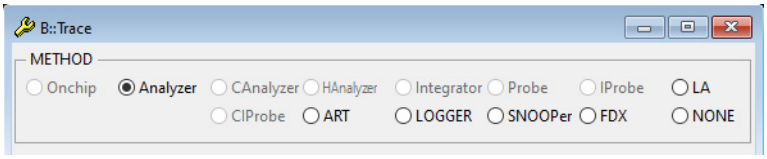
The following table aims to assist new users in choosing the most suitable code coverage measurement variant. It provides a simplified overview, intentionally avoiding complex details for easier understanding.

Code Coverage Measurement Variant	Incremental in Leash Mode (fallback)	Incremental in STREAM Mode	Continuous SPY	Continuous RTS
Variant Description	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.
TRACE32 Trace Solutions	- Onchip trace - PowerTrace - µTrace Cortex-M - µTrace RISC-V 32-Bit - CombiProbe for Cortex-M - CombiProbe for RISC-V 32-Bit - Tracing via USB stack - TRACE32 ISS/ART - Trace of virtual targets	- 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.	

TRACE32 Trace Tool Solutions for Code Coverage

Let's take a closer look at the TRACE32 Trace Solutions. From a technical point of view, all trace solutions that enable almost seamless recording of program execution on one or more cores are suitable for TRACE32 Code Coverage.

TRACE32 Trace Solutions are represented in the TRACE32 PowerView GUI through the trace METHOD. The currently selected trace method can be viewed in the **Trace.state** window.



Solutions for Incremental Code Coverage in Leash Mode

Trace METHODS suitable for incremental code coverage in Leash mode are:

TRACE32 Trace Solution	TRACE32 Trace Method
Onchip Trace	**Onchip**
PowerTrace The term 'PowerTrace' encompasses all Lauterbach products that are labeled as POWER TRACE. TRACE32 PowerTrace supports both parallel and serial trace ports, depending on the tool configuration.	**Analyzer**
µTrace The term µTrace refers to all Lauterbach products labeled µTRACE.	**CAnalyzer**
CombiProbe The term CombiProbe refers to all Lauterbach products labeled COMBIPROBE. However, using the CombiProbe for code coverage requires a core trace recording of the core architecture under test.	**CAnalyzer**
Tracing via USB stack Some core architectures that support debugging over the USB stack also provide the capability to stream core trace data to the host computer via the USB stack.	**HAnalyzer**
TRACE32 ISS TRACE32 PowerView features an integrated instruction set simulator that supports bus tracing, making it suitable for measuring code coverage. Lauterbach offers a TRACE32 ISS for most supported core architectures.	**Analyzer**

TRACE32 Trace Solution	TRACE32 Trace Method
TRACE32 ART ART (Advanced Register Trace) is a form of single-step assembler tracing and should only be used for code coverage measurement if a TRACE32 Instruction Set Simulator is not available for the target core architecture. Since it significantly slows down program execution, this solution is only suitable for unit testing.	**ART**
Code Coverage with Virtual Targets Virtual targets have significant limitations in trace capture, as it considerably slows down program execution and requires substantial memory resources. To improve performance, Lauterbach collaborates with virtual target vendors, such as Synopsys, to offload certain code coverage measurements to the virtual target itself. Currently, this approach is effective only for object code coverage.	**Analyzer**

Solutions for Incremental Coverage in STREAM Mode and Continuous Code Coverage

Trace METHODS suitable for incremental code coverage in STREAM mode and continuous code coverage measurements are:

TRACE32 Trace Solution	TRACE32 Trace Method
PowerTrace The term 'PowerTrace' encompasses all Lauterbach products that are labeled as POWER TRACE. TRACE32 PowerTrace supports both parallel and serial trace ports, depending on the tool configuration. You can use incremental code coverage in STREAM mode and continuous code coverage only if the average data rate at the trace port remains below the average trace streaming rate of the PowerTrace module. The average trace streaming rate for your PowerTrace module is available at **"Trace Modules"** in TRACE32 Terminology, page 10 (trace32_terms.pdf).	**Analyzer**
µTrace The term µTrace refers to all Lauterbach products labeled µTRACE.	**CAnalyzer**
CombiProbe The term CombiProbe refers to all Lauterbach products labeled COMBIPROBE. However, using the CombiProbe for code coverage requires a core trace recording of the core architecture under test.	**CAnalyzer**

In most cases, TRACE32 automatically detects the available trace solution and selects the appropriate trace METHOD. For all other cases, the trace method can be manually set using the **Trace.METHOD** command.

Best Practices for Trace Recording

The following recommendations apply to the Trace Methods **Analyzer** (excluding TRACE32 ISS), **CAnalyzer**, and **HAnalyzer**.

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 trace infrastructure:

- 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:

- **"Training Cortex-M Tracing"** (training_cortexm_etm.pdf)
- MPC5xxx/SPC5xxx, QorIQ and RH850: **"Training MPC5xxx/SPC5xx Nexus Tracing"** (training_nexus_mpc5500.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:

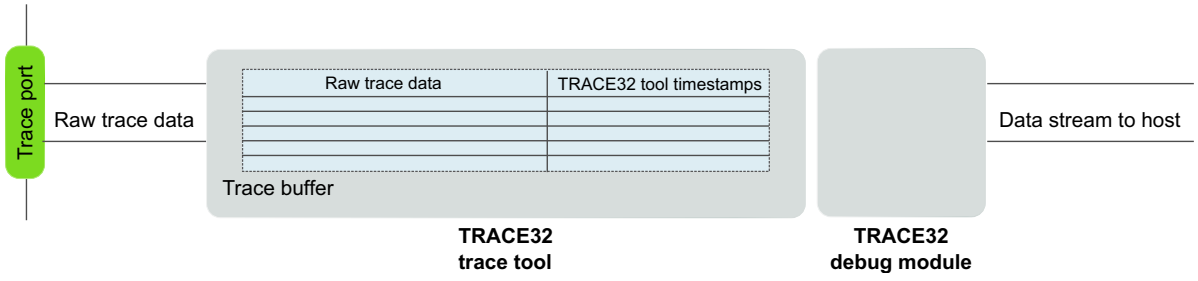
```
Go
Break
SILENT.Trace.Find FLOWERROR /ALL
IF FOUND.COUNT()!=0.
(
  PRIVATE &msg
  &msg="FLOWERRORS were found in the analyzed trace recording."
  &msg="&msg It is recommended to check"
  &msg="&msg if the trace recording works properly."
  ECHO FOUND.COUNT() "&msg"
)
ELSE
(
  ECHO "The analyzed trace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

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

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

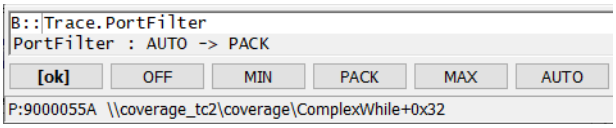
Disable Timestamps for Trace Streaming

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.



Steps in Preparation for Code Coverage Measurement

General Overview

This chapter outlines the steps required to set up code coverage measurement for each specific metric. These steps include:

1. **Loading all necessary files**

   The files needed depend on the code coverage metric used.

   If you plan to use continuous measurement, code coverage is processed during program execution. To enable continuous coverage measurement, it's essential to load the object code into **TRACE32 Virtual Memory**, which resides on the host computer, to ensure optimal performance. In this case, reload the code using the **/VM** option.

```
; load the executable to the TRACE32 Virtual Memory
; for continuous code coverage measurement
Data.LOAD.Elf "my_app.elf" /VM
```

2. **Excluding alignment padding**

   Many core architectures require data to be aligned at specific memory addresses for efficient access. To maintain proper function alignment, extra bytes — containing no executable code — are often inserted at the end of functions. These extra bytes must be excluded to achieve accurate code coverage measurement.

Additional step for MC/DC, condition coverage, and decision coverage

3. Customizing special tagging options

TRACE32 has predefined default settings for tagging optimized-out decisions, infinite loops, and linear decisions. Review these settings and adjust them as needed to suit your requirements.

The following tagging option must be reviewed and adjusted if necessary.

- **IGNOREDEAD**

  Source code decisions that have been optimized out by the compiler are ignored and thus not tagged by default. For a detailed explanation of the command, see **COVerage.Option IGNOREDEAD**.

- **IGNOREINF**

  Infinite loops are decisions that are always evaluated as true, meaning they will consistently be tagged as incomplete. By default, they are treated as statements and tagged accordingly. For a detailed explanation of the command, see **COVerage.Option IGNOREINF**.

- **IGNORELINEAR**

  This option controls how decisions within assignments, nested assignments, returns, and decisions used as function parameters are tagged. By default, they are tagged in the same manner as all other decisions. For a detailed explanation of the command, see **COVerage.Option IGNORELINEAR**.

4. Configuring the TRACE32 Code Coverage Modes

For the code coverage metrics MC/DC (Modified Condition/Decision Coverage), condition coverage, and decision coverage, various modes are available. These include code coverage measurement with targeted or no instrumentation, breakpoint-assisted code coverage measurement, and full-instrumentation code coverage measurement. To configure TRACE32 for a specific mode, you need to explicitly specify the data to be analyzed during code coverage processing:

Trace: Include conditional branches and, when applicable, conditional instructions recorded in the trace for the code coverage calculation.
Command: **sYmbol.ECA.BINary.ControlFlowMode.Trace**

INSTR: Include code Instrumentation probes within the source code in the code coverage calculation.
Command: **sYmbol.ECA.BINary.ControlFlowMode.INSTR**

Break: Account for status information captured when TRACE32 briefly halts at a code coverage breakpoint during the calculation.
Command: **sYmbol.ECA.BINary.ControlFlowMode.Break**

Maintaining Access to Measurement Setup for Later Evaluation

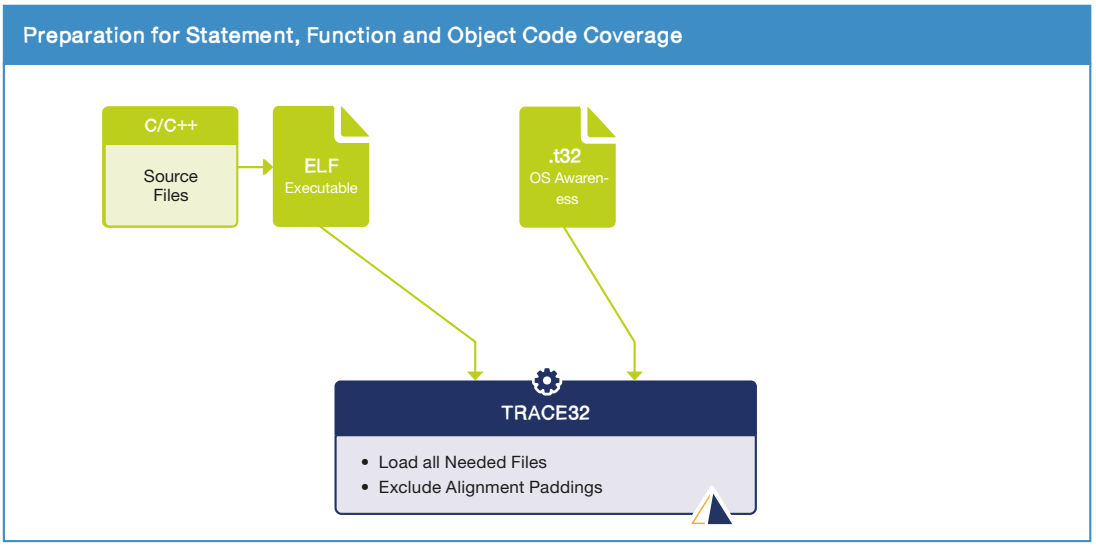
Since code coverage measurement is conducted in TRACE32, while the evaluation — covering statement, call, and function coverage, as well as decision, condition, and modified condition/decision coverage (MC/DC) — takes place in a web browser, it is crucial for the evaluator to have access to key information about the measurement setup. The following elements are particularly important:

- Access to the loaded .elf file
- Knowledge of the source code version used to generate the .elf file
- Access to the startup script used to configure TRACE32 for code coverage measurement

The following table illustrates which measurement setup is best suited for each code coverage metric.

	Statement Function	**Call**	**Condition Decision MC/DC**
non-instrumented executable + ControlFlowMode Trace	**best**	unsuitable	unsuitable
non-instrumented executable + .eca files + ControlFlowMode Trace	possible	**best**	only applicable if there are no observability gaps or if existing gaps are to be addressed through methods other than code instrumentation
non-instrumented executable + .eca files + ControlFlowMode Trace + ControlFlowMode Break	possible	possible	suitable only if you absolutely do not want to use instrumentation
lightweight instrumented executable + .eca files + ControlFlowMode Trace + ControlFlowMode INSTR	possible	possible	**best**
fully instrumented executable + .eca files + ControlFlowMode INSTR	not recommended	not recommended	as a fallback if all else fails

With the **ClipSTORE ECA** command, code coverage setup commands from the **sYmbol.ECA.BINary** command group can be transferred into the clip text.



The following files need to be loaded into TRACE32:

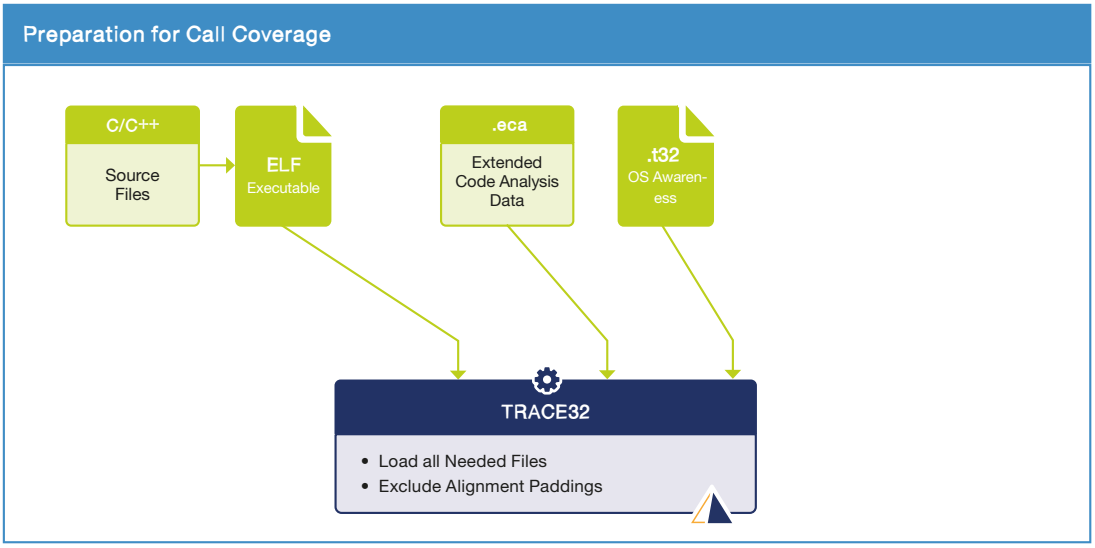
- Executable, which includes paths to all source files
- TRACE32 OS Awareness, if an operating system is used by the target application

After loading the files, alignment padding must be excluded from the code coverage measurement.

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
; for continuous code coverage only
Data.LOAD.Elf "my_app.elf" /VM

; load the OS Awareness if needed
TASK.CONFIG myos.t32

; exclude alignment padding from code coverage measurement
sYmbol.CLEANUP.AlignmentPaddings
```

Preparation for Call Coverage



The following files need to be loaded into TRACE32:

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

After loading the files, alignment padding must be excluded from the code coverage measurement.

The following commands can be used for this purpose:

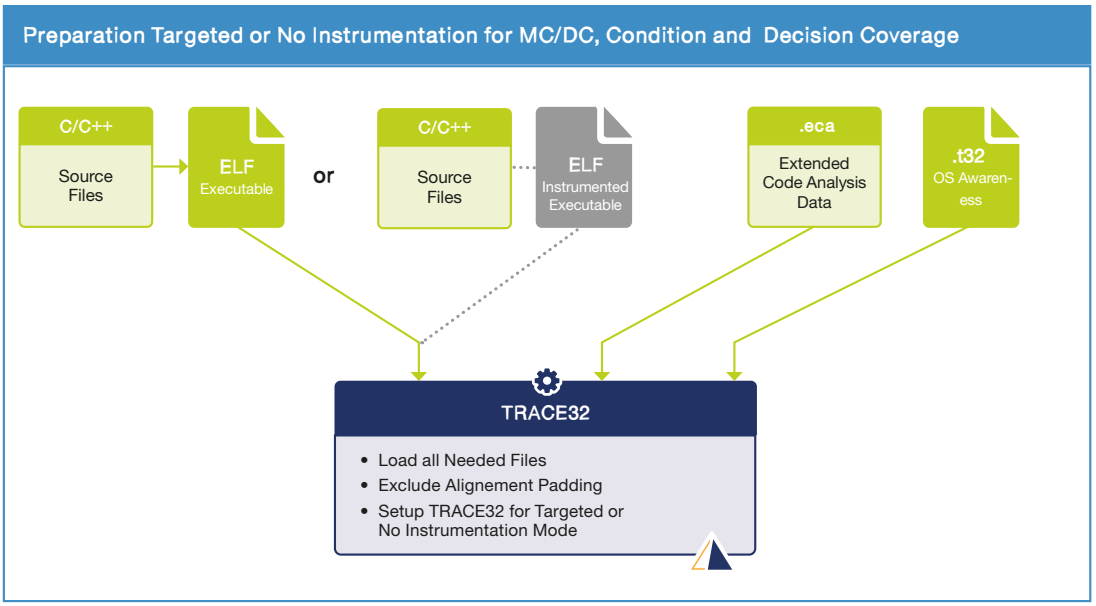
```
; basic debug and trace setup

; 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
; for continuous code coverage only
Data.LOAD.Elf "my_app.elf" /VM

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

; load the OS Awareness is needed
TASK.CONFIG myos.t32

; exclude alignment paddings from code coverage measurement
sYmbol.CLEANUP.AlignmentPaddings
```

Preparation for MC/DC, Condition and Decision Coverage

The preparation is different for the individual TRACE32 Code Coverage Modes:

- **"Preparation for Code Coverage with Targeted Instrumentation/No Instrumentation"**, page 74.
- **"Preparation for Code Coverage with Breakpoints (Code in RAM)"**, page 77.
- **"Preparation for Code Coverage with Breakpoints (Code in Flash)"**, page 79.
- **"Preparation for Code Coverage with Full Instrumentation"**, page 80

Preparation for Code Coverage with Targeted Instrumentation/No Instrumentation



The Targeted Instrumentation/No Instrumentation code coverage modes must be set up as follows:

1. **Load the executable**

   The presence of the **flag file** `og_detected.txt` indicates whether an instrumented or non instrumented executable should be loaded for code coverage measurement.

If an instrumented executable is loaded, the source file paths must be adjusted to point to the original sources. 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)

2. **Load the generated .eca file**

3. **Load the TRACE32 OS Awareness**

   Load the TRACE32 OS Awareness if an operating system is used by the target application.

4. **Exclude alignment padding**

   After loading the files, alignment padding must be excluded from the code coverage measurement.

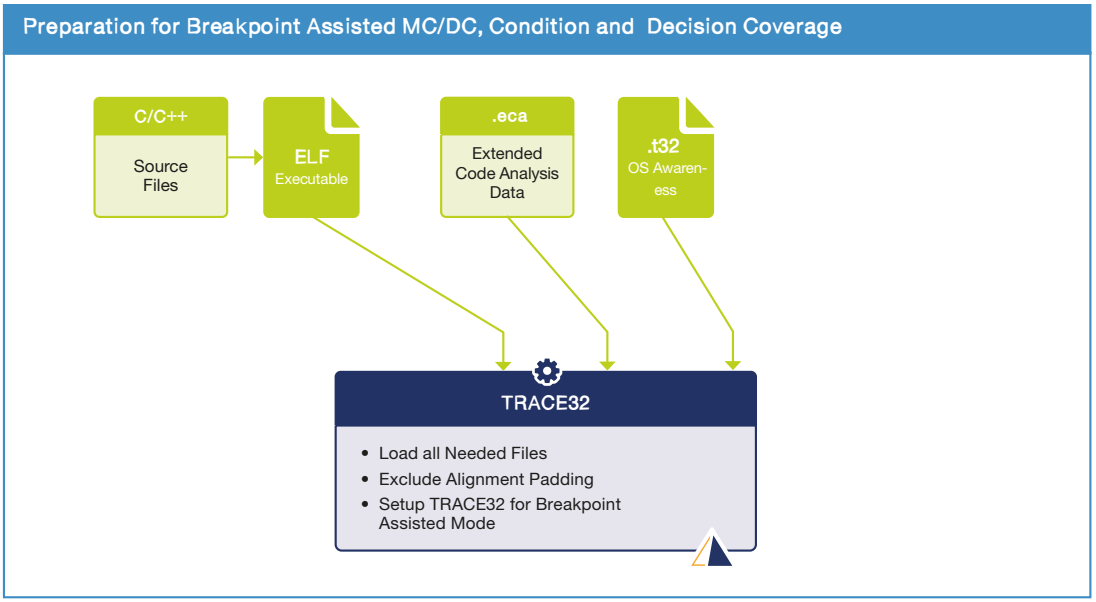
5. **Setup TRACE32 for targeted instrumentation/no instrumentation code coverage**

   In both modes, conditional branches and, when possible, conditional instructions recorded in the trace (Trace) serve as input data for code coverage processing. When targeted instrumentation is used, the instrumentation probes within the object code (INSTR) are additionally included to the input data. A static preprocessing is required in both cases.

The following framework can be used for this purpose:

```
; basic debug and trace setup

; Load the appropriate executable based on the presence of the flag file
IF FILE.EXIST(og_detected.txt)
(
   Data.LOAD.Elf "my_app_targeted.elf"
   ; mirror the instrumented executable to the TRACE32 Virtual Memory
   ; for continuous code coverage only
   Data.LOAD.Elf "my_app_targeted.elf" /VM

   ; translate the links in 'my_app_targeted.elf' to point to the original source
   ; files, allowing to test with the original sources
   sYmbol.SourcePATH.Translate "./instrumented_tree" "./original_tree"
    PRINT "Executable with targeted instrumentation loaded."
)
ELSE
(
   Data.LOAD.Elf "my_app.elf"
   ; mirror the executable to the TRACE32 Virtual Memory
   ; for continuous code coverage only
   Data.LOAD.Elf "my_app.elf" /VM
   PRINT "Non instrumented executable loaded."
)

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

; load the OS Awareness is needed
TASK.CONFIG myos.t32

; exclude alignment padding from code coverage measurement
sYmbol.CLEANUP.AlignmentPaddings
```

```
; if you used ETM.COND ALL for ETMv4 on Cortex-M and Cortex-R during the build
; process, you must also use it for code coverage measurement
; ETM.COND ALL

; configure code coverage mode

; use conditional branches and, where possible, conditional instructions
; in trace recording for code coverage measurement
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; use instrumentation probes in "my_app_targeted.elf" for code coverage
; measurement
IF FILE.EXIST(og_detected.txt)
(
   sYmbol.ECA.BINary.ControlFlowMode.INSTR ON
)

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

; everything should be set up to ensure there are no observability gaps,
; but double-check it
IF sYmbol.ECA.BINary.GAPNUMBER()>0.
(
   PRINT sYmbol.ECA.BINary.GAPNUMBER() " observability gaps detected. \
   Please check the remaining observability gaps."
)
```



To configure the breakpoint-assisted code coverage mode when the object code is located in RAM, follow these steps:

1. **Load required Files**

   The following files must be loaded into TRACE32:

   - Not-instrumented executable, which includes the links to all source files.

   - Generated .eca files.

   - TRACE32 OS Awareness, if the target application uses an operating system.

2. **Exclude alignment padding**

   After loading the files, ensure that alignment padding is excluded from the code coverage measurement.

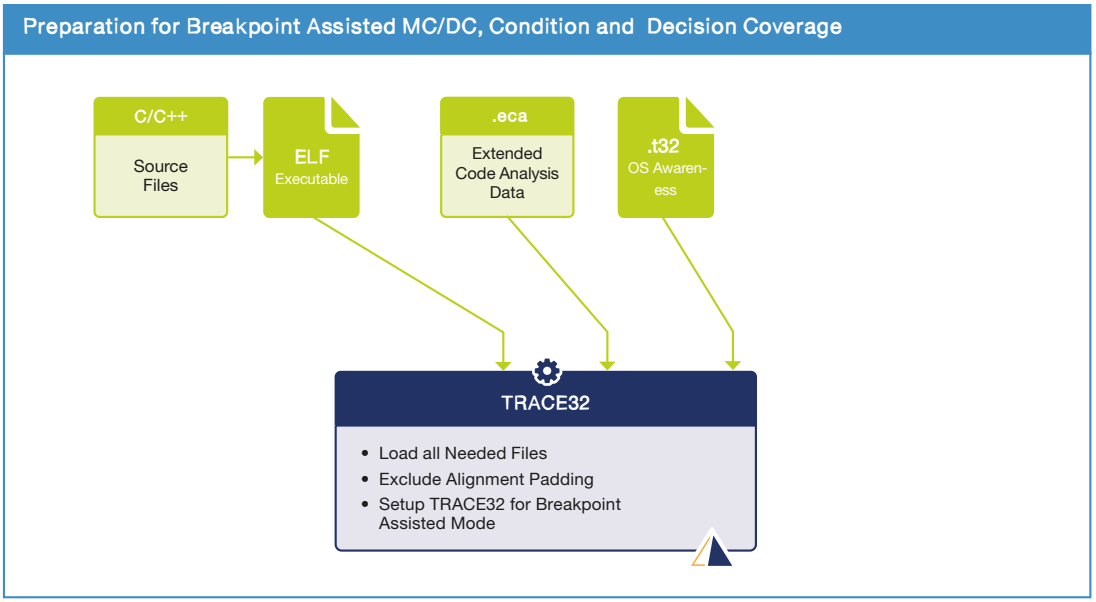
3. **Setup TRACE32 for breakpoint assisted code coverage**

   In this mode, conditional branches and, where possible, conditional instructions recorded in the trace (Trace) serve as input data for code coverage processing, along with status information recorded at code coverage breakpoints (Break). Static preprocessing is required to set the necessary breakpoints at the decisions/conditions where observability gaps have been detected.

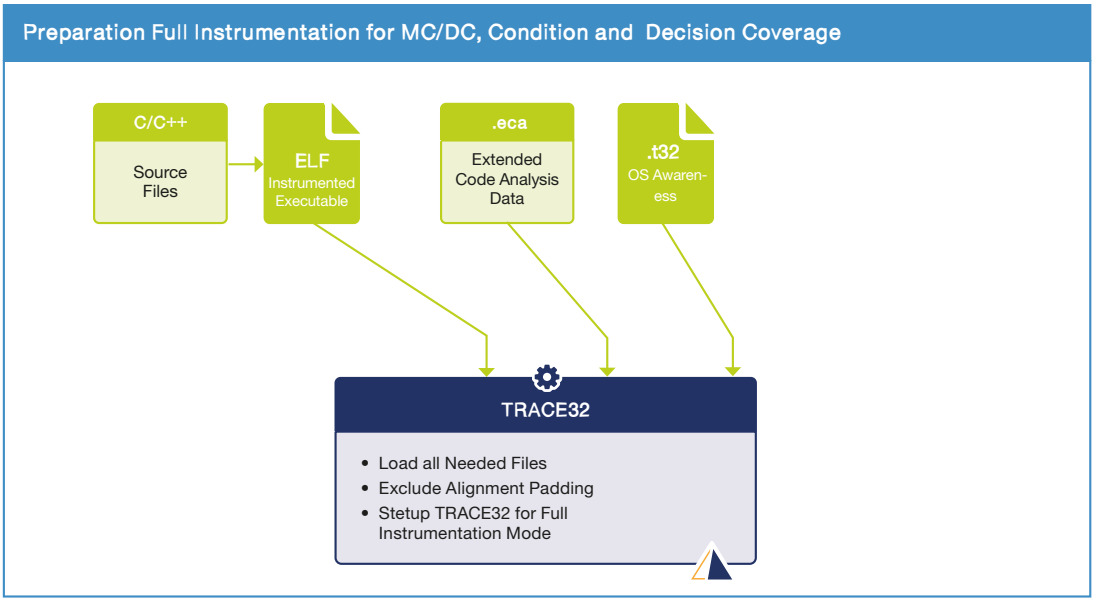
   For completeness, it should be noted that the status information checked when a code coverage breakpoint briefly stops program execution is stored within the internal TRACE32 code coverage system.

The following framework can be used for this purpose:

```
; basic debug and trace setup

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

; mirror the executable to the TRACE32 Virtual Memory
; for continuous code coverage only
Data.LOAD.Elf "my_app.elf" /VM

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

; load the OS Awareness is needed
TASK.CONFIG myos.t32

; exclude alignment paddings from code coverage measurement
sYmbol.CLEANUP.AlignmentPaddings

; configure code coverage mode

; use conditional branches and, where possible, conditional instructions
; in trace recording for code coverage measurement
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; set a code coverage breakpoint at all decisions/conditions where an
; observability gap was detected during static preprocessing
; and use the status information recorded during program execution
; as input for the code coverage processing
sYmbol.ECA.BINary.ControlFlowMode.Break ON

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

; display list of code coverage breakpoints
Break.List
```



To configure the breakpoint-assisted code coverage mode when the object code is located in Flash, follow these steps:



To set up the full instrumentation code coverage mode, follow these steps:

1. **Load required Files**

   The following files must be loaded into TRACE32:

   - Instrumented executable, which includes the links to all source files. If an instrumented executable is loaded, the source file paths must be adjusted to point to the original sources. 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.

   - TRACE32 OS Awareness, if the target application uses an operating system.

2. **Exclude alignment padding**

   After loading the files, ensure that alignment padding is excluded from the code coverage measurement.

3. **Setup TRACE32 for full instrumentation mode**

   In this mode, the instrumentation probes embedded in the object code (INSTR) serve as the input data for code coverage analysis.

The following framework can be used for this purpose:

```
; basic debug and trace setup

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

; mirror the instrumented executable to the TRACE32 Virtual Memory
; for continuous code coverage only
Data.LOAD.Elf "my_app_targeted.elf" /VM

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

; translate the links in 'my_app_full.elf' to point to the original source
; files, allowing to test with the original sources

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

; load the OS Awareness if needed
TASK.CONFIG myos.t32

; exclude alignment paddings from code coverage measurement
sYmbol.CLEANUP.AlignmentPaddings

; configure code coverage mode

; don't use conditional branches and, where possible, conditional
; instructions in trace recording for code coverage measurement
sYmbol.ECA.BINary.ControlFlowMode.Trace OFF
```

```
; use instrumentation probes in "my_app_full.elf" for code coverage
; measurement
sYmbol.ECA.BINary.ControlFlowMode.INSTR ON

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

; everything should be set up to ensure there are no observability gaps,
; but double-check it
IF sYmbol.ECA.BINary.GAPNUMBER()>0.
(
   PRINT sYmbol.ECA.BINary.GAPNUMBER() " observability gaps detected. \
   Please check the remaining observability gaps."
)
```

Code Coverage Measurement

This chapter provides detailed instructions for performing the code coverage measurement in then following variants.

Incremental Code Coverage Measurement in Leash Mode

Core Principles

Incremental coverage in Leash mode is compatible with all processor architectures that provide program flow information recorded to a trace buffer, as well as with all TRACE32 configurations (see **"Solutions for Incremental Code Coverage in Leash Mode"**, page 62). This method serves as a dependable fallback solution that can be applied in the vast majority of scenarios.

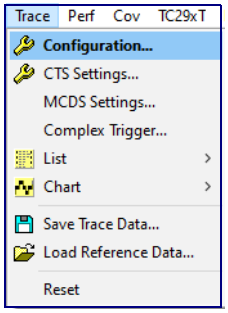
Measurement Steps

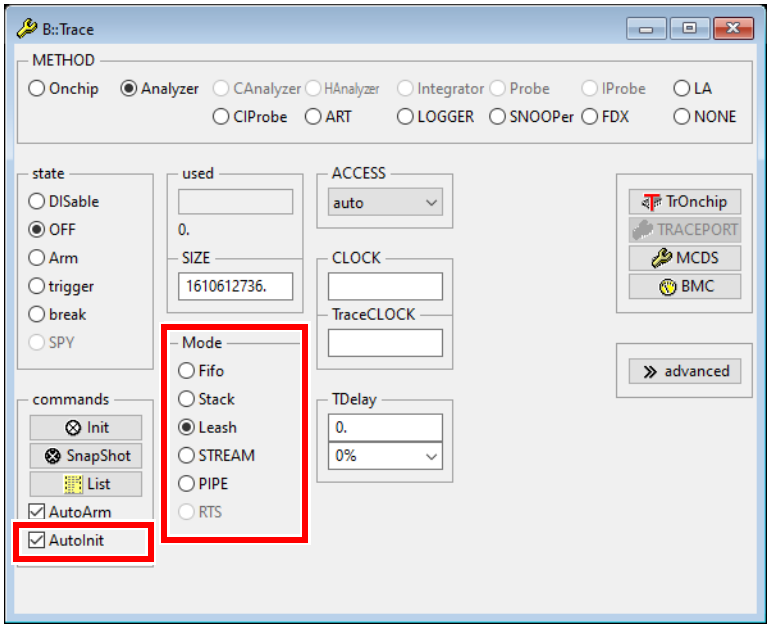
1. **Set up the trace recording.**

   - Configure the trace to Leash Mode via the **Trace configuration** window or by executing the command **Trace.Mode Leash**. This ensures the target halts when the trace buffer is nearly full, preventing data loss.

   - If Leash Mode is unavailable, consider using **Stack** or **FIFO** mode.

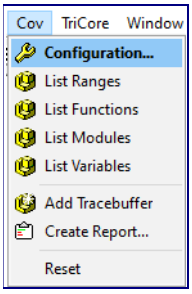
- Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to clear the trace buffer before each recording.

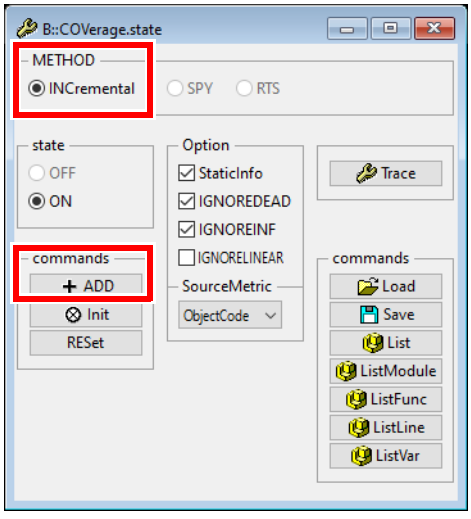




2. Set up code coverage.

Open the **COVerage configuration** window to configure the code coverage. Ensure that the coverage METHOD INCremental is configured. Alternatively, you can use the **COVerage.METHOD INCremental** command to achieve this.





3. Start program execution.

Start program execution and allow it to run until it halts.

4. Calculate code coverage results.

Use the **ADD** button in the **COVerage configuration** window or the command **COVerage.ADD** to calculate code coverage across all metrics and add the result to the TRACE32 Code Coverage System. If .eca files were loaded when the code coverage measurement was prepared, the

command will output warnings regarding any detected observability gaps. These warnings are relevant only for MC/DC (Modified Condition/Decision Coverage), condition coverage, and decision coverage; they can be disregarded for other code coverage metrics.

5. **Repeat steps 3 and 4.**

   Continue executing the program and adding results until you have collected sufficient code coverage data.

6. **Export code coverage results.**

   Use the command **COVerage.EXPORT.JSONE** to export the code coverage result. The Lauterbach command-line tool, **t32covtool**, enables the merging of coverage results taken at different times, with various builds, and under differing target configurations. Additionally, it can generate an HTML file for detailed code coverage evaluation in a web browser. For more details, refer to **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

7. **Evaluate code coverage results in TRACE32.**

   The code coverage results for the metrics object code and object code-based (OCB) decision coverage are not exported when using the command **COVerage.EXPORT.JSONE**. These metrics must be analyzed and evaluated directly in TRACE32. For more details, see **"Code Coverage Evaluation in TRACE32"**, page 110.

   However, intermediate results for other code coverage metrics can also be inspected within TRACE32, see **"Evaluation of Intermediate Results"**, page 122.

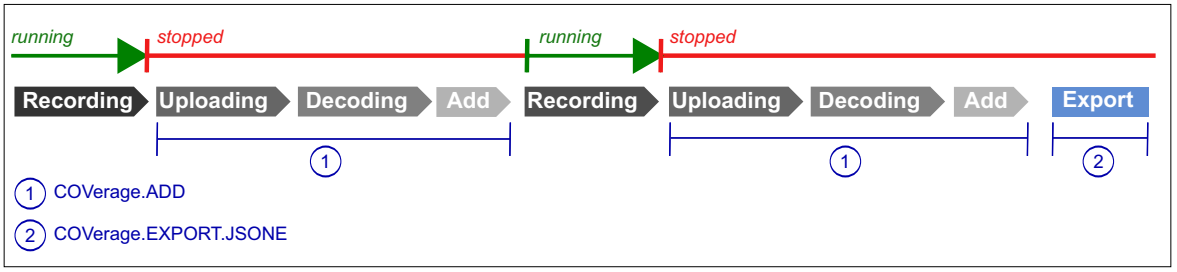
Measurement Script

The code coverage measurement can be automated by creating a PRACTICE script. It is assumed that the preconditions listed in **"Best Practices for Trace Recording"**, page 65 are satisfied before running the script.

```
// prepare trace recording
Trace.Mode Leash
Trace.AutoInit ON

// perform trace recording and code coverage calculation
COVerage.METHOD INCremental
RePeaT 10.
(
    Go.direct
    WAIT !STATE.RUN()
    COVerage.ADD
)

// export results
COVerage.EXPORT.JSONE coverage_data1.json /NoISTAT

// or view results in TRACE32
// COVerage.Option.SourceMetric <metric>
// COVerage.ListFunc
```

Measurement Diagram

This diagram aims to simplify the comparison of various code coverage measurement variants.



A key feature of incremental code coverage is base mode is that individual steps are executed sequentially. While the program is running, trace information is **recorded**. After the program is halted, 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.
- Code coverage is calculated based on the recorded program flow and **added** to the TRACE32 Code Coverage System.

When the code coverage measurement is complete, the results can be **exported** for further processing and evaluation.

Core Principles

When using TRACE32 trace hardware for recording, it is possible to stream the trace data directly to a file on the host file system. For more details about the supported TRACE32 Trace Tool solutions and their maximum streaming rates, refer to **"Solutions for Incremental Coverage in STREAM Mode and Continuous Code Coverage"**, page 63.

Streaming trace data to the host computer allows for extended recording durations. However, decoding large volumes of raw trace data into a program flow trace and calculating the code coverage can be time-consuming. To address this, TRACE32 offers two alternative methods to optimize the process:

1. **Continuous Code Coverage in RTS Mode**

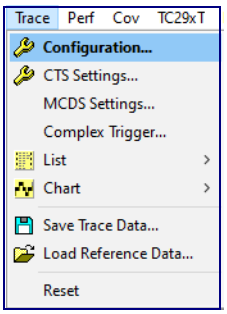
   RTS mode decodes the trace data and preprocesses it for code coverage calculation during recording. This method is compatible with all major architectures. For additional details, see **"Continuous Code Coverage Measurement in RTS Mode"**, page 92.

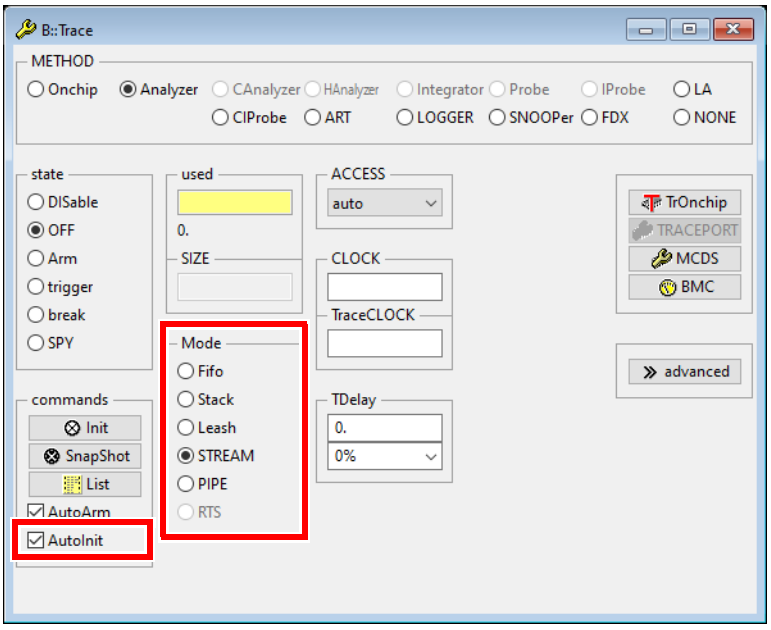
2. **Continuous Code Coverage in SPY Mode**

   If RTS mode is not supported for your architecture, SPY mode code coverage is a viable alternative. For more information, refer to "**"Continuous Code Coverage Measurement in SPY Mode"**, page 97.

1. **Set up the trace recording.**

   - Set the trace to STREAM Mode either via the **Trace Configuration** window or via the **Trace.Mode STREAM** command.
   - Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to clear the trace buffer before each recording.





By default, TRACE32 opens a streaming file in the temporary files directory (**OS.PresentTemporaryDirectory()**). Optionally, you can specify a different file using the **Trace.STREAMFILE** command. For optimal performance, it is recommended to use the fastest available drive on the host system. Example:

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

You can limit the maximum size of the streaming file with the **Trace.STREAMFileLimit** command. Example:

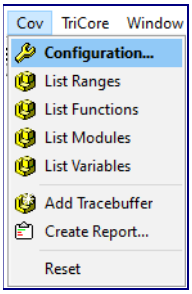
```
; limit the size of the streaming file to 5 GBytes
Trace.STREAMFileLimit 5000000000.
```

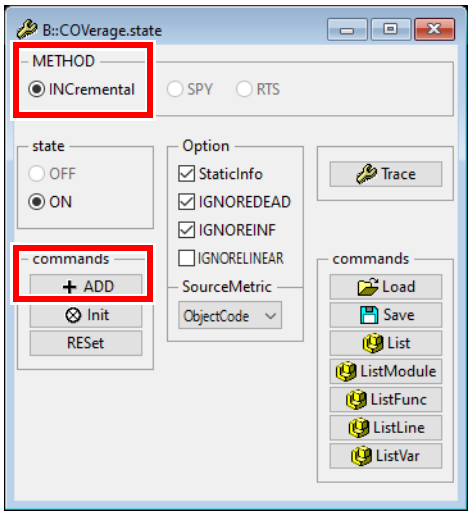
The trace recording stops when the streaming file reaches the specified size limit.

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, but no timestamps. For additional details, refer to the chapter **"Disable Timestamps for Trace Streaming"**, page 66.

2. Set up code coverage.

- Open the **COVerage configuration** window to configure the code coverage. Ensure that the coverage METHOD INCremental is configured. Alternatively, you can use the **COVerage.METHOD INCremental** command to achieve this.





3. Start and stop the program execution.

To perform code coverage calculation, the program execution on the target must be stopped. There are several ways to achieve this:

- The user can manually stop program execution.
- A breakpoint can be set to halt program execution at a specific point.
- Alternatively, a script can be used to stop program execution after a defined period of time.

4. Calculate code coverage results.

Use the **ADD** button in the **COVerage configuration** window or the command **COVerage.ADD** to calculate code coverage across all metrics and add the result to the TRACE32 Code Coverage System. If .eca files were loaded when the code coverage measurement was prepared, the

command will output warnings regarding any detected observability gaps. These warnings are relevant only for MC/DC (Modified Condition/Decision Coverage), condition coverage, and decision coverage; they can be disregarded for other code coverage metrics.

5. **Export code coverage results.**

   Use the command **COVerage.EXPORT.JSONE** to export the code coverage result. The Lauterbach command-line tool, **t32covtool**, enables the merging of coverage results taken at different times, with various builds, and under differing target configurations. Additionally, it can generate an HTML file for detailed code coverage evaluation in a web browser. For more details, refer to **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

6. **Evaluate code coverage results in TRACE32.**

   The code coverage results for the metrics object code and object code-based (OCB) decision coverage are not exported when using the command **COVerage.EXPORT.JSONE**. These metrics must be analyzed and evaluated directly in TRACE32. For more details, see **"Code Coverage Evaluation in TRACE32"**, page 110.

   However, intermediate results for other code coverage metrics can also be inspected within TRACE32, see **"Evaluation of Intermediate Results"**, page 122.

Measurement Script

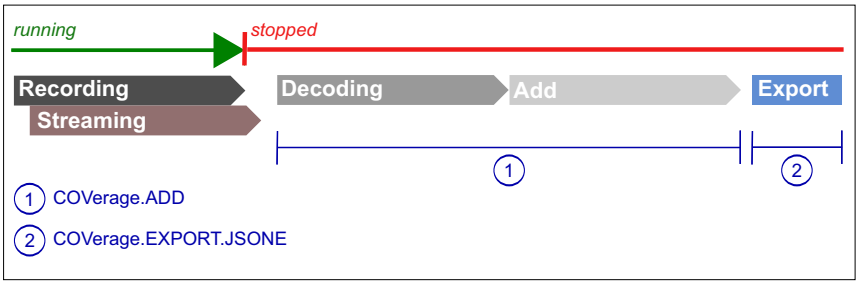
The code coverage measurement can be automated by creating a PRACTICE script. It is assumed that the preconditions listed in **"Best Practices for Trace Recording"**, page 65 are satisfied before running the script.

```
// prepare trace recording
Trace.Mode STREAM
Trace.STREAMFile "D:\streamfile.t32"
Trace.STREAMFileLimit 5000000000.

Trace.AutoInit ON

Trace.PortFilter MAX
// perform trace recording and code coverage calculation
Go
WAIT 10.s
Break
COVerage.ADD

// export results
COVerage.EXPORT.JSONE coverage_data1.json /NoISTAT

// or view results in TRACE32
// COVerage.Option.SourceMetric <metric>
// COVerage.ListFunc
```

Measurement Diagram

This diagram aims to simplify the comparison of various code coverage measurement variants.

Incremental code coverage in streaming mode is exclusively available when using TRACE32 trace hardware. This configuration enables the recording of substantially more trace data during a single test run. However, it's essential to recognize that the streaming rate has an upper limit.



During program execution, trace information is recorded and streamed to the host computer. Once the program is halted, the **COVerage.ADD** command performs the following tasks:

- **Decodes** the raw trace data to reconstruct the complete program flow.
- Calculates code coverage from the recorded program flow and **add** it into the TRACE32 Code Coverage System.

For long recordings, the **COVerage.ADD** command can take a significant amount of time to complete.

After the code coverage measurement is finalized, the results can be **exported** for further processing and evaluation.

Continuous Code Coverage Measurement in RTS Mode

Core Principles

Continuous Code Coverage in RTS Mode calculates code coverage during trace recording, delivering complete results in the TRACE32 Code Coverage system immediately after program execution halts. Since trace data is typically discarded after code coverage calculation, the measurement time is not limited. However, if trace data needs to be saved for diagnostic purposes, storage space is required, which may impose a restriction on recording time.

This mode, however, is limited to specific processor architectures and trace protocols, including:

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

If RTS is not supported for your architecture, SPY Mode Code Coverage may serve as an alternative. For details, see **"Continuous Code Coverage Measurement in SPY Mode"**, page 97.

RTS requires a TRACE32 trace hardware. Additionally, trace data must be streamed to the host file system without issues. For more information on trace streaming requirements, refer to the **Trace.Mode STREAM** command description.

Measurement Steps

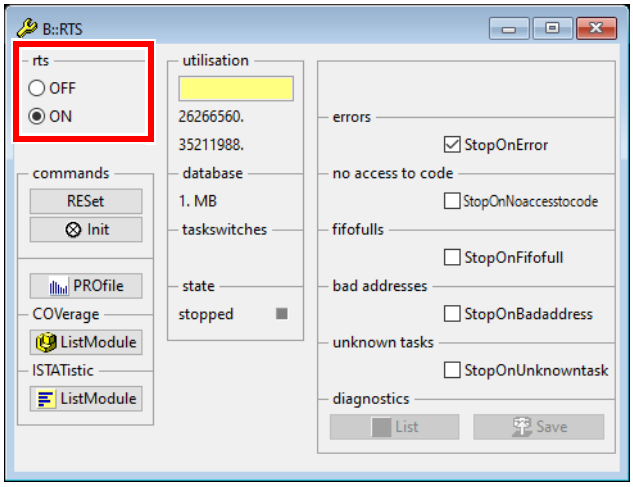
1. **Setup RTS.**

   - To calculate code coverage during recording, it is essential to prepare the process thoroughly in advance to ensure the required performance. Two conditions must be fulfilled.

   A. As part of the preparation for continuous code coverage, the object code was already be loaded into the TRACE32 Virtual Memory. Refer to the section **"Steps in Preparation for Code Coverage Measurement"**, page 68 for more details.

   B. Static preprocessing for code coverage calculation must now be performed using the **sYmbol.ECA.BINary.PROCESS** command.

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



2. Start and stop the program execution.

3. Export code coverage results.

Use the command **COVerage.EXPORT.JSONE** to export the code coverage result. The Lauterbach command-line tool, **t32covtool**, enables the merging of coverage results taken at different times, with various builds, and under differing target configurations. Additionally, it can generate an HTML file for detailed code coverage evaluation in a web browser. For more details, refer to **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

4. Evaluate code coverage results in TRACE32.

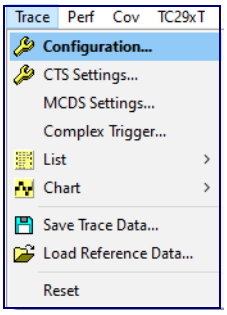
The code coverage results for the metrics object code and object code-based (OCB) decision coverage are not exported when using the command **COVerage.EXPORT.JSONE**. These metrics must be analyzed and evaluated directly in TRACE32. For more details, see **"Code Coverage Evaluation in TRACE32"**, page 110.

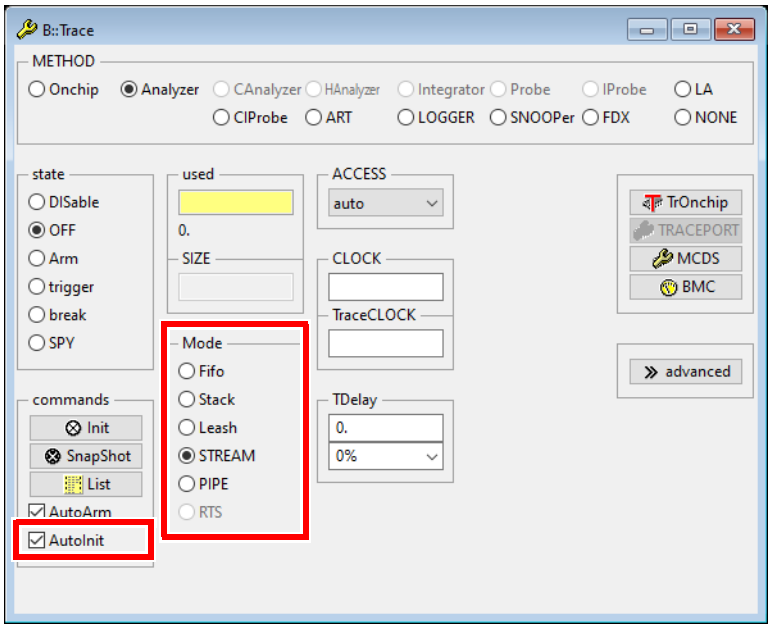
However, intermediate results for other code coverage metrics can also be inspected within TRACE32. In RTS mode, you can even inspect intermediate results while trace recording is ongoing. Instead of waiting until the code coverage evaluation to identify unexecuted code paths and initiate additional measurements, users can inspect critical points during the measurement process and take immediate action—such as stimulating the system—to ensure and verify that the code is executed. For details refer to **"Evaluation of Intermediate Results"**, page 122.

By default, the trace data is discarded after calculating code coverage. To save it for diagnostic purposes, insert the following before step 1.

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

- Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to clear the trace buffer before each recording.





By default, TRACE32 opens a streaming file in the temporary files directory (**OS.PresentTemporaryDirectory()**). Optionally, you can specify a different file using the **Trace.STREAMFILE** command. For optimal performance, it is recommended to use the fastest available drive on the host system. Example:

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

You can limit the maximum size of the streaming file with the **Trace.STREAMFileLimit** command. Example:

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

The trace recording stops when the streaming file reaches the specified size limit.

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, but no timestamps. For additional details, refer to the chapter **"Disable Timestamps for Trace Streaming"**, page 66.

Measurement Script

The code coverage measurement can be automated by creating a PRACTICE script. It is assumed that the preconditions listed in **"Best Practices for Trace Recording"**, page 65 are satisfied before running the script

Use this setup only if you need the trace recording for diagnostic purposes.

```
Trace.AutoInit ON

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

Trace.PortFilter MAX
```

Here the standard script for continuous code coverage in RTS mode.

```
// prepare RTS mode
sYmbol.ECA.BINary.PROCESS
RTS.ON

Go
WAIT 10.s
Break

// export results
COVerage.EXPORT.JSONE coverage_data1.json /NoISTAT

// or view results in TRACE32
// COVerage.Option.SourceMetric <metric>
// COVerage.ListFunc
```

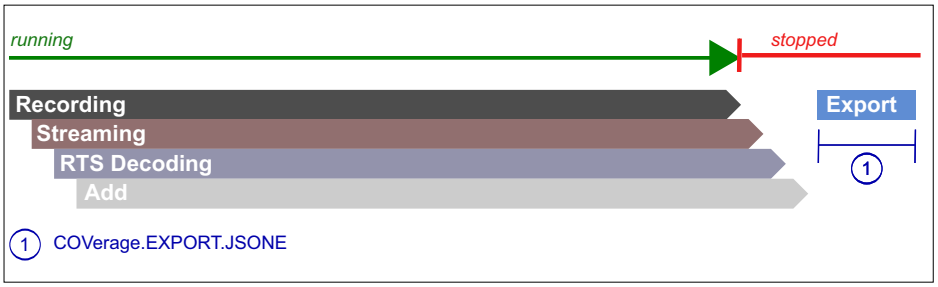
Measurement Diagram

This diagram aims to simplify the comparison of various code coverage measurement variants.

The key advantage of RTS mode code coverage is that all steps are executed in parallel. This enables the rapid processing of large volumes of trace data, with code coverage results available immediately after program execution stops.

The following steps are performed concurrently with the trace recording:

- Raw trace data is **streamed** to the host computer, with the option to save it to a streaming file.
- The raw trace data are **decoded** to reconstruct the program flow
- Code coverage is calculated.
- The code coverage results are **added** to the TRACE32 Coverage System.



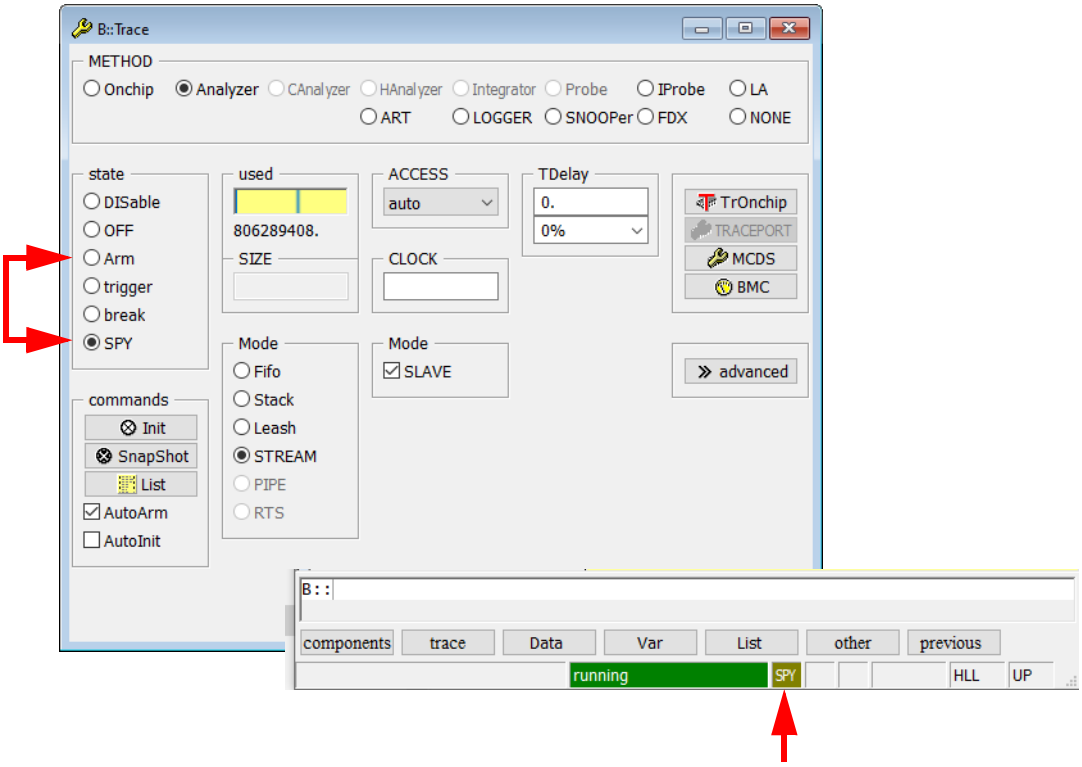
Core Principles

Continuous Code Coverage in SPY Mode calculates code coverage during trace recording and provides complete results in the TRACE32 Code Coverage System after a brief post-processing delay once program execution stops.

SPY Mode requires a TRACE32 trace hardware. Additionally, trace data must be streamed to the host file system without issues. For more information on trace streaming requirements, refer to the **Trace.Mode STREAM** command description.

The key concept is to periodically interrupt trace streaming to calculate intermediate code coverage results, which slightly reduces the available bandwidth for trace streaming. The current trace state switches between Arm and SPY.

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



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

Note that TRACE32 does not suspend trace streaming unless the trace memory of the TRACE32 trace tool, acting as a large FIFO, is less than 50% full.

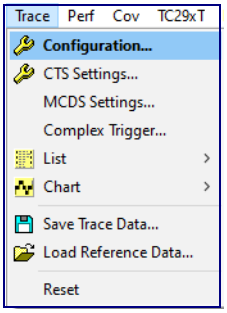
1. **Setup SYP mode.**

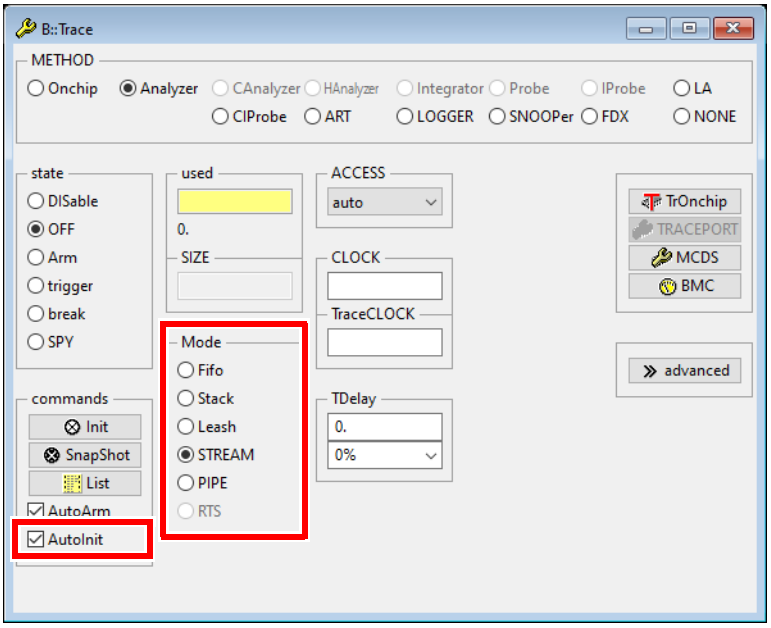
- To calculate code coverage during recording, it is essential to prepare the process thoroughly in advance to ensure the required performance. Two conditions must be fulfilled.

A. As part of the preparation for continuous code coverage, the object code was already be loaded into the TRACE32 Virtual Memory. Refer to the section **"Steps in Preparation for Code Coverage Measurement"**, page 68 for more details.

B. Static preprocessing for code coverage calculation must now be performed using the **sYmbol.ECA.BINary.PROCESS** command.

- Set the trace to STREAM Mode either via the **Trace Configuration** window or via the **Trace.Mode STREAM** command.
- Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to clear the trace buffer before each recording.





By default, TRACE32 opens a streaming file in the temporary files directory (**OS.PresentTemporaryDirectory()**). Optionally, you can specify a different file using the **Trace.STREAMFILE** command. For optimal performance, it is recommended to use the fastest available drive on the host system. Example:

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

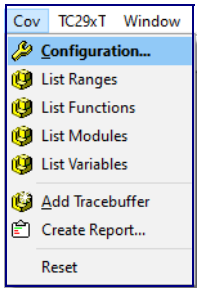
You can limit the maximum size of the streaming file with the **Trace.STREAMFileLimit** command. Example:

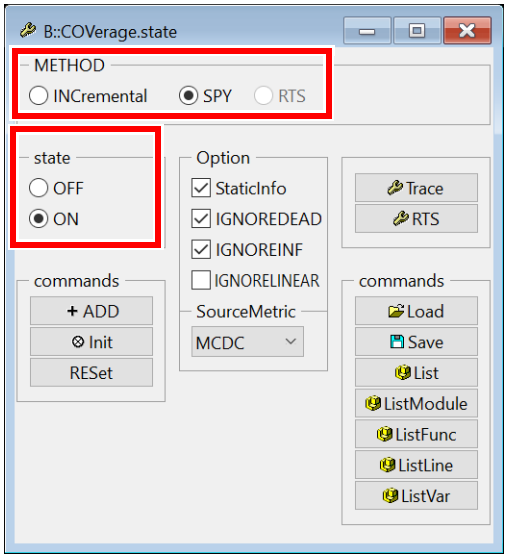
```
; limit the size of the streaming file to 5 GBytes
Trace.STREAMFileLimit 5000000000.
```

The trace recording stops when the streaming file reaches the specified size limit.

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, but no timestamps. For additional details, refer to the chapter **"Disable Timestamps for Trace Streaming"**, page 66.

- Set the coverage method to SPY by using the command **COVerage.METHOD SPY** or by selecting **SPY** in the **COVerage configuration** window.
- Enable **SPY** mode code coverage by the command **COVerage.ON** or by selecting the **ON** radio button in the state field.





2. **Start and stop the program execution.**

3. **Export code coverage results.**

   Use the command **COVerage.EXPORT.JSONE** to export the code coverage result. The Lauterbach command-line tool, **t32covtool**, enables the merging of coverage results taken at different times, with various builds, and under differing target configurations. Additionally, it can generate an HTML file for detailed code coverage evaluation in a web browser. For more details, refer to **"Code Coverage Evaluation Outside TRACE32 - t32covtool"**, page 107.

4. **Evaluate code coverage results in TRACE32.**

   The code coverage results for the metrics object code and object code-based (OCB) decision coverage are not exported when using the command **COVerage.EXPORT.JSONE**. These metrics must be analyzed and evaluated directly in TRACE32. For more details, see **"Code Coverage Evaluation in TRACE32"**, page 110.

   However, intermediate results for other code coverage metrics can also be inspected within TRACE32. In SPY mode, you can even inspect intermediate results while trace recording is ongoing. Instead of waiting until the code coverage evaluation to identify unexecuted code paths and initiate additional measurements, users can inspect critical points during the measurement process and take immediate action—such as stimulating the system—to ensure and verify that the code is executed. For details refer to **"Evaluation of Intermediate Results"**, page 122.

Measurement Script

The code coverage measurement can be automated by creating a PRACTICE script. It is assumed that the preconditions listed in **"Best Practices for Trace Recording"**, page 65 are satisfied before running the script.

```
// prepare SPY mode
Trace.AutoInit ON

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

Trace.PortFilter MAX

COVerage.METHOD SPY
COVerage.ON

sYmbol.ECA.BINary.PROCESS

Go
WAIT 10.s
Break

// export results
COVerage.EXPORT.JSONE coverage_data1.json /NoISTAT

// or view results in TRACE32
// COVerage.Option.SourceMetric <metric>
// COVerage.ListFunc
```

Measurement Diagram

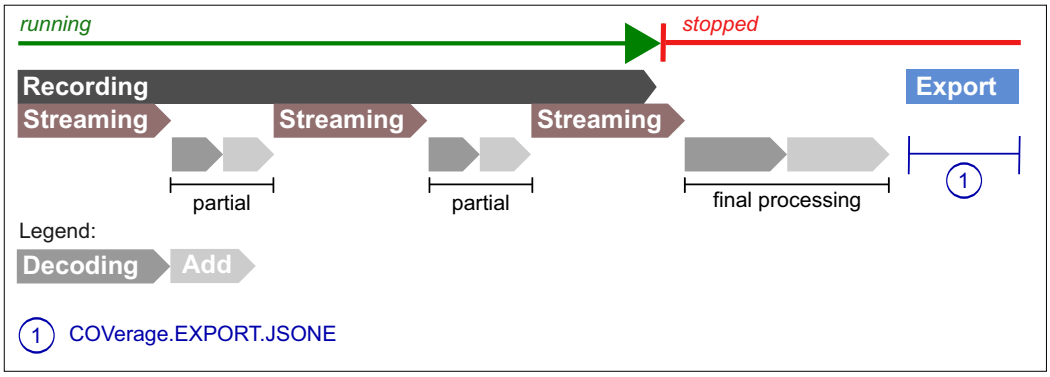
This diagram aims to simplify the comparison of various code coverage measurement variants.

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 calculate the code coverage and **add** the results to the TRACE32 Code Coverage System

TRACE32 dynamically adjusts the periodic suspension of trace streaming to ensure a gap-free trace while processing as much raw trace data as possible.

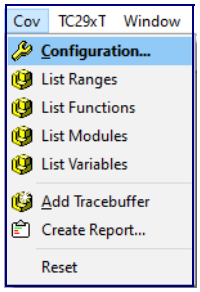


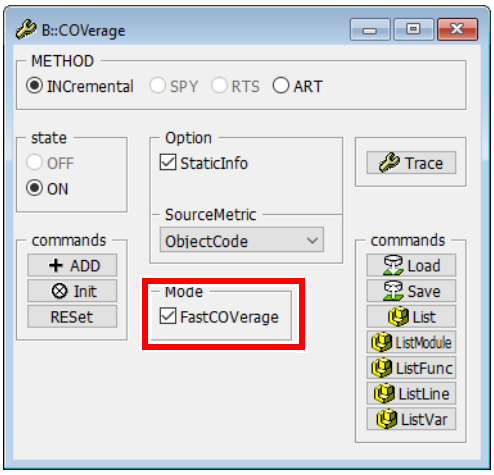
Tracing the program execution on a virtual target slows down its performance. To minimize this impact, Lauterbach works closely together with manufacturers such as Synopsys. The basic idea is that some parts of the code coverage processing are offloaded to the virtual target. This information is uploaded 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”.





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

- Object code coverage (see **“Object Code Coverage”**, page 110)
- Decision coverage (ocb) (see **“Object Code Based (ocb) Decision Coverage”**, page 116)

An example script might look like this:

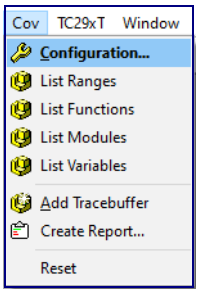
```
COVerage.RESet
COVerage.METHOD INCremental
COVerage.Mode FastCOVerage ON

Go

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

Break

COVerage.Add

COVerage.ListFunc
```

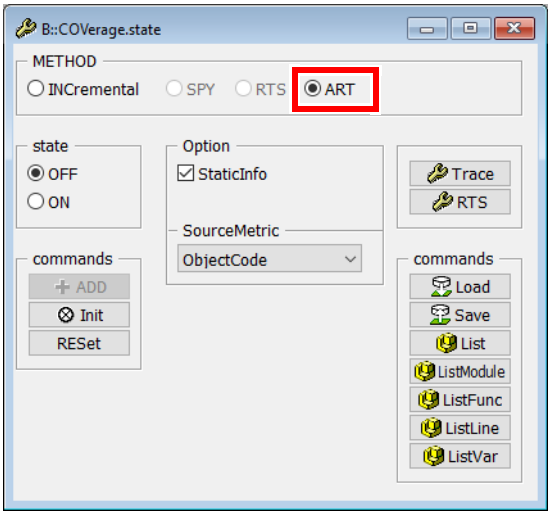
Details about the code coverage analysis itself are provided in the chapter **“Code Coverage Evaluation Outside TRACE32 - t32covtool”**, page 107.

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.





Trace data recorded with ART can be used for:

- Object code coverage (see **"Object Code Coverage"**, page 110)
- Decision coverage (ocb) (see **"Object Code Based (ocb) Decision Coverage"**, page 116)

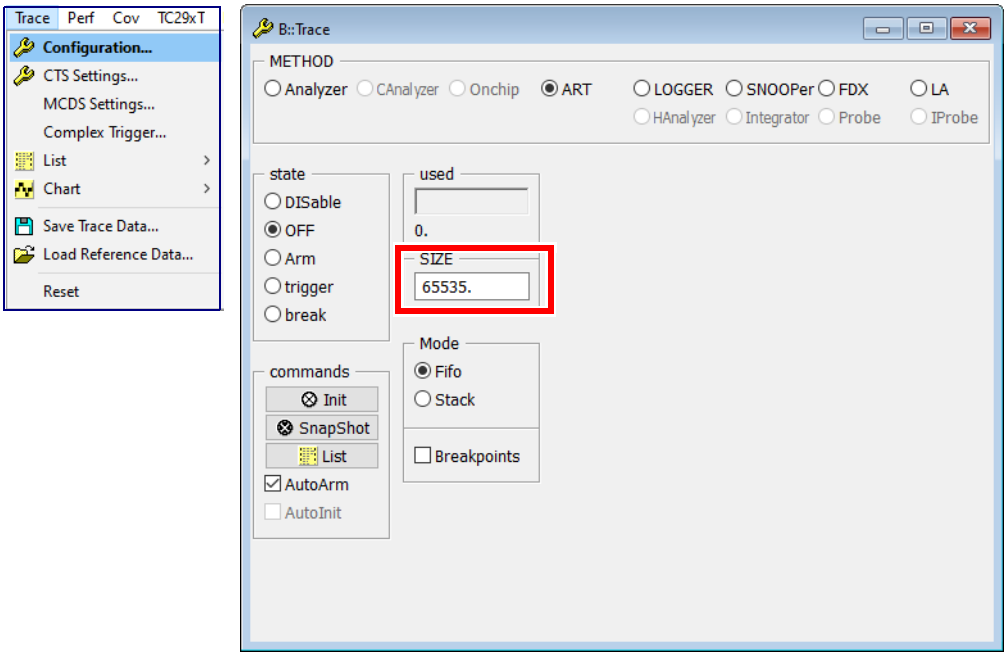
Where possible, it is recommended to use the TRACE32 Instruction Set Simulator with **Trace.METHOD Analyzer** instead of ART. This has a better performance and supports all code coverage metrics.

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.

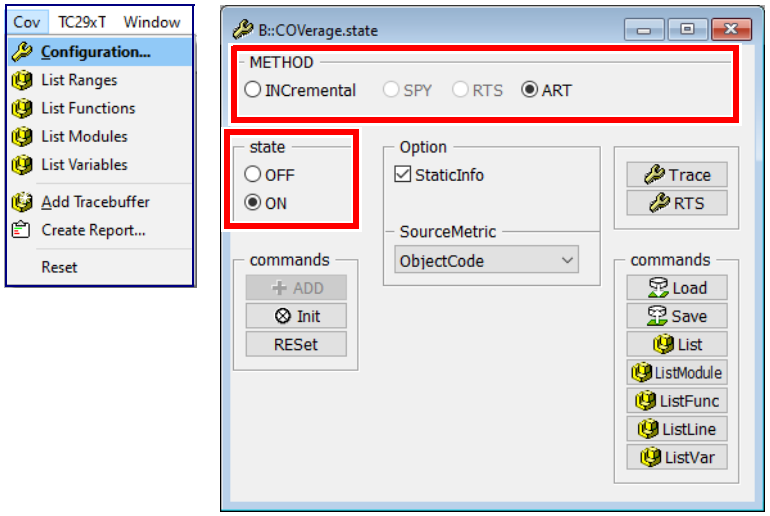
1. Select **Trace.METHOD ART** in the **Trace configuration** window.

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.

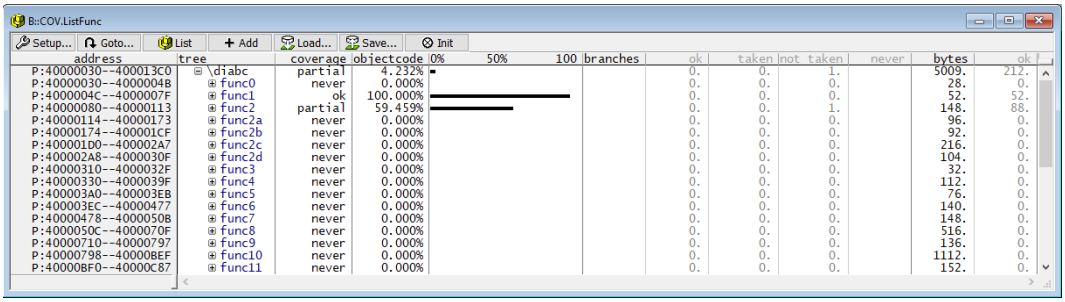


3. Set **COVerage.METHOD ART** in the **COVerage configuration** window.

4. Enable ART code coverage with **COVerage.ON**.



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 Outside TRACE32 - t32covtool"**, page 107.

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 Outside TRACE32 - t32covtool

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.

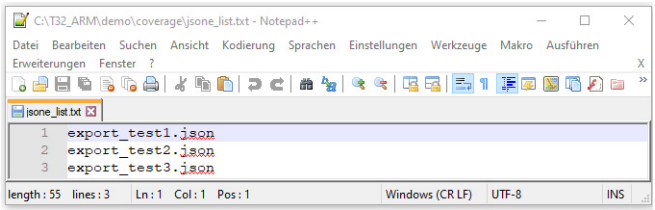
As already described, the **COVerage.EXPORT.JSONE** command allows you to export information on the functions and source code lines from the code coverage system to a JSON file. Lauterbach offers the command line tool **t32covtool** to merge the exported results and/or create an overall report. t32covtool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

-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`.

Code Coverage Evaluation in TRACE32

The two object code-based code coverage metrics – object code coverage and object code based (ocb) decision coverage – need to be evaluated in TRACE32. Other source code based code coverage metrics should preferably be evaluated outside of TRACE32 using a web browser.

In certain cases, it can be useful to evaluate the results of the current measurement directly in TRACE32 for metrics such as statement coverage, decision coverage, condition coverage, MC/DC, call coverage, and function coverage. This is particularly helpful for determining whether specific source code sections of interest were executed in the current measurement. Instead of exporting, post-processing, and analyzing coverage results in a web browser, TRACE32 enables direct evaluation, saving time. Additionally, it provides insights into how source level coverage relates to the execution of the underlying object code.

Object Code Coverage

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**

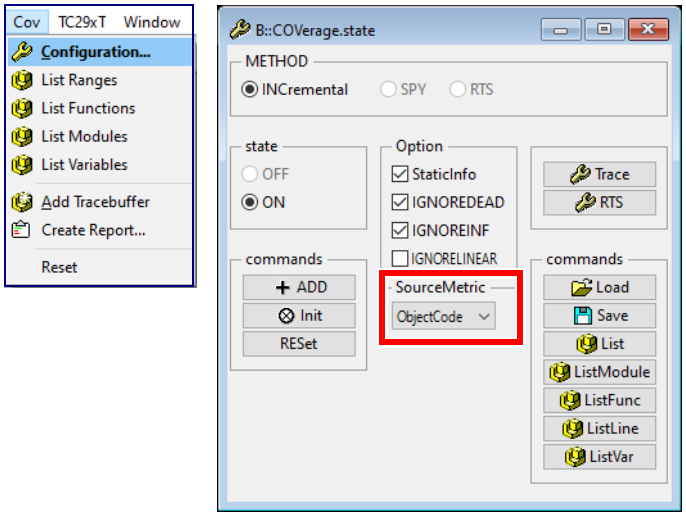
  This is the tagging scheme for all trace protocols that provide details on the execution of conditional branches and conditional instructions. Refer also to the **COVerage.INFO** command. Currently, this tagging scheme is used only for Arm/Cortex cores with Arm-ETMv1 or Arm-ETMv3 protocols, as well as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

- **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 146.

The TRACE32 Code Coverage System provides results for all metrics. To evaluate code coverage for the ObjectCode metric in TRACE32, select **SourceMetric ObjectCode** in the **COVerage configuration window** or use the command: **COVerage.Option SourceMetric ObjectCode.**



The following commands provide a tabular analysis:

COVerage.ListModule

COVerage.ListFunc

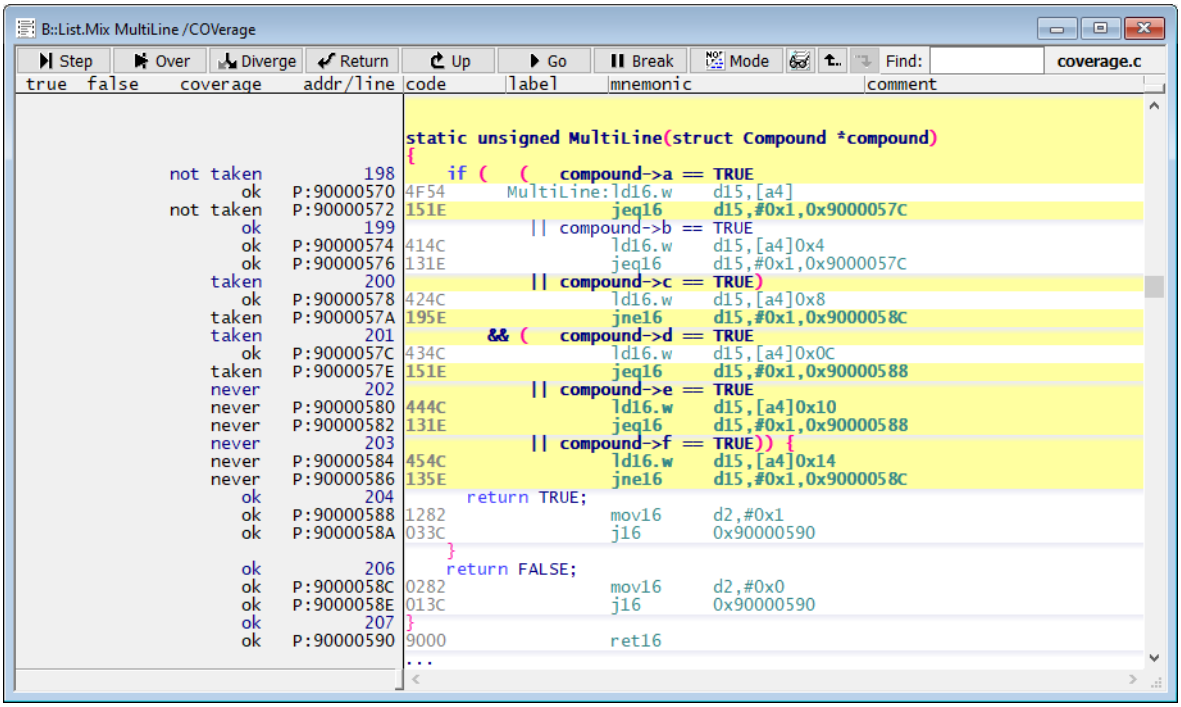
COVerage.ListLine

The following command displays tagging at both the source code and object code levels:

List.Mix /COVerage

This TRACE32 command displays tagging for the *MultiLine* function, applying the ObjectCode metric:

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

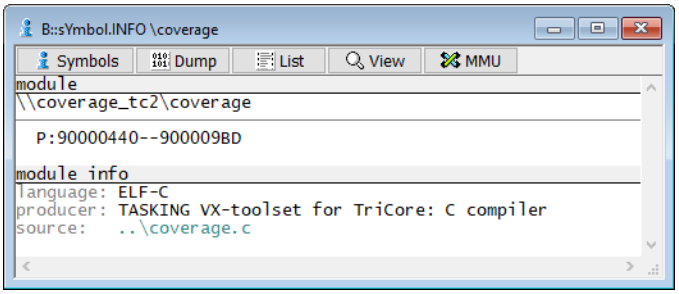


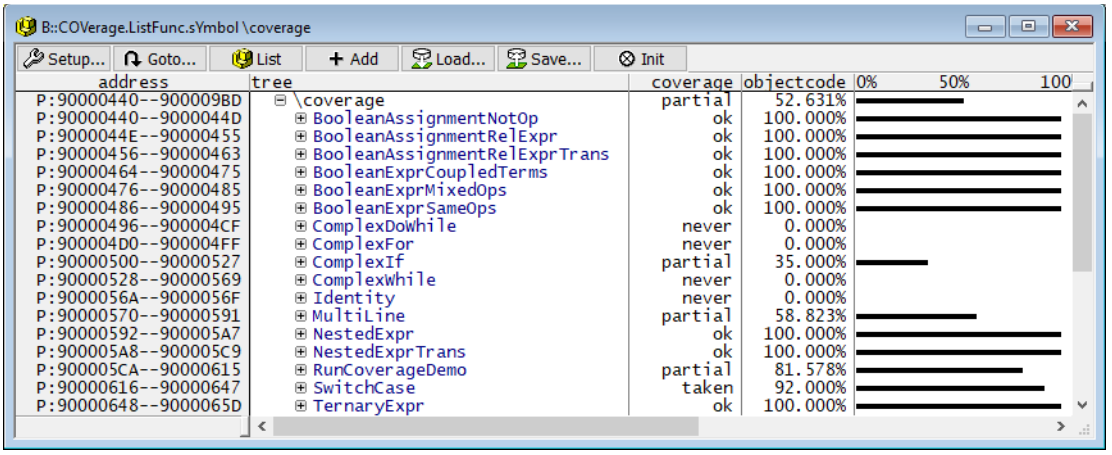
The preceding screenshot was taken using the Infineon TriCore™ debugger. Its instruction set only supports conditional branches, but does not include conditional instructions. As a result, the following tagging is used for the ObjectCode metric:

ok	The object code instruction is considered fully covered. If the instruction is a conditional branch, it is tagged as ok if it has been both taken and not taken at least once. All other object code instructions are tagged as 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 as taken if it has been taken at least once but never not taken.
not taken	If the object code is a conditional branch, it is tagged as not taken if it has been not taken at least once but never taken.

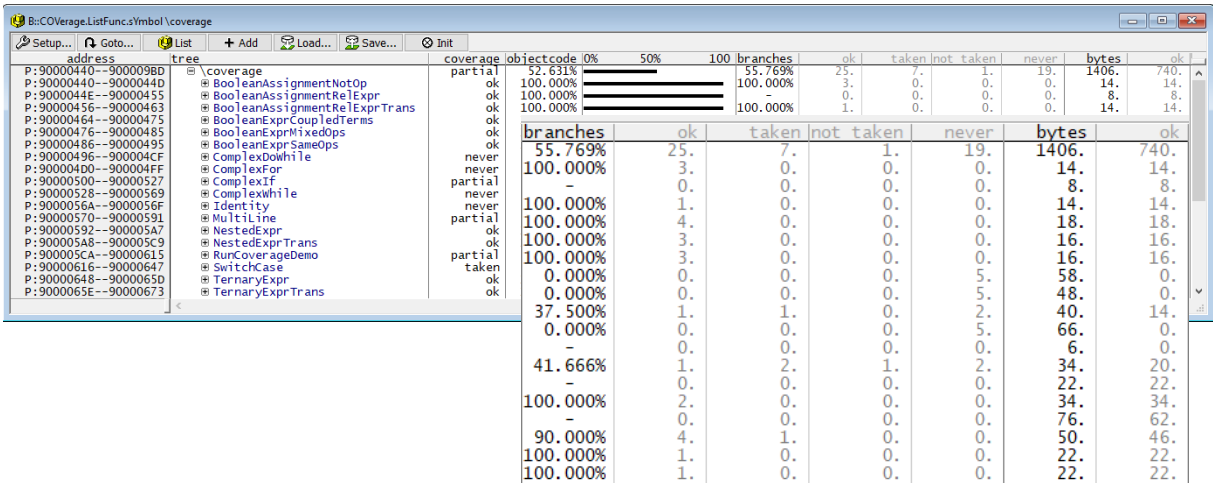
This TRACE32 command provides a tabular analysis of all functions within the 'coverage' module. Typically, a module corresponds to a source code file.

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





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



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

Example Script

```
// Demo script "~~/demo/coverage/mcdc/measure_mcdc.cmm"

// Select code coverage metric ObjectCode
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
```

Lauterbach no longer officially recommends this metric. However, it may still be useful for addressing specific issues. If such a situation occurs, our support team will recommend this metric to you.

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 for a source code line is achieved when all object code instructions generated from that line are tagged as ok.

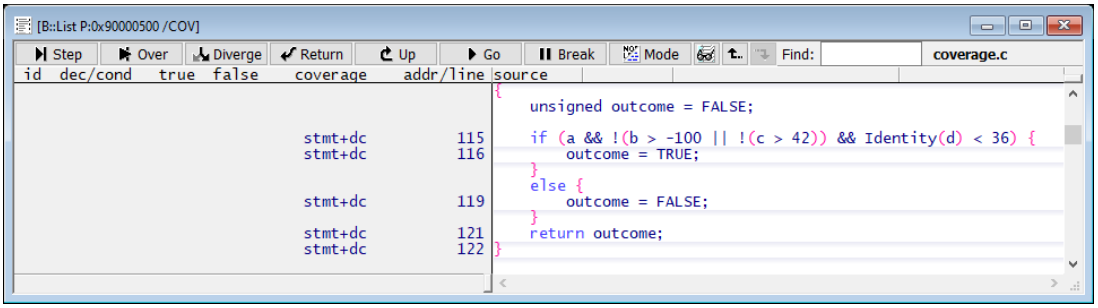
However, the following should be taken into account:

Unoptimized code may lead to false negatives, where decisions are marked as incomplete even though decision coverage has already been achieved. This means that ocb decision coverage might require more test cases than full decision coverage.

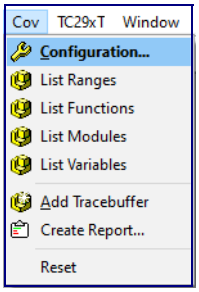
Optimized code may lead to false positives if a condition is no longer represented by a conditional branch/instruction or if the trace protocol does not provide information about the execution of conditional instructions. A false positive means that decision coverage is indicated too early.

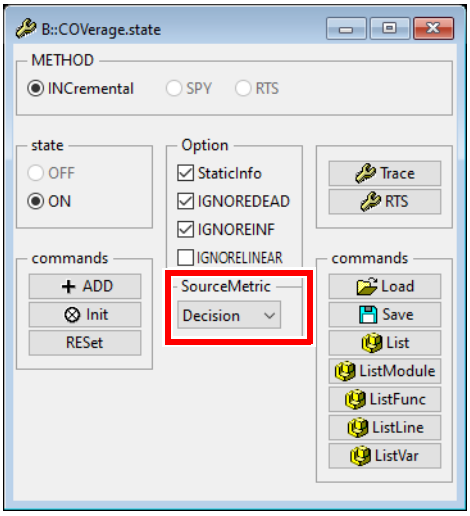
Since the source code is not analyzed for ocb decision coverage, TRACE32 does not know where decisions are located. Therefor source code lines are tagged as follows:

- **stmt+dc | incomplete**



The TRACE32 code coverage system provides results for all metrics. To evaluate code coverage using the ocb decision coverage metric in TRACE32, select **SourceMetric Decision** in the **COVerage configuration window** or use the command: **COVerage.Option SourceMetric Decision.** It is not possible to specifically select ocd decision coverage in TRACE32. Instead, TRACE32 automatically applies it if no .eca data was loaded during the preparation for code coverage measurement.





The following commands provide a tabular analysis:

COVerage.ListModule

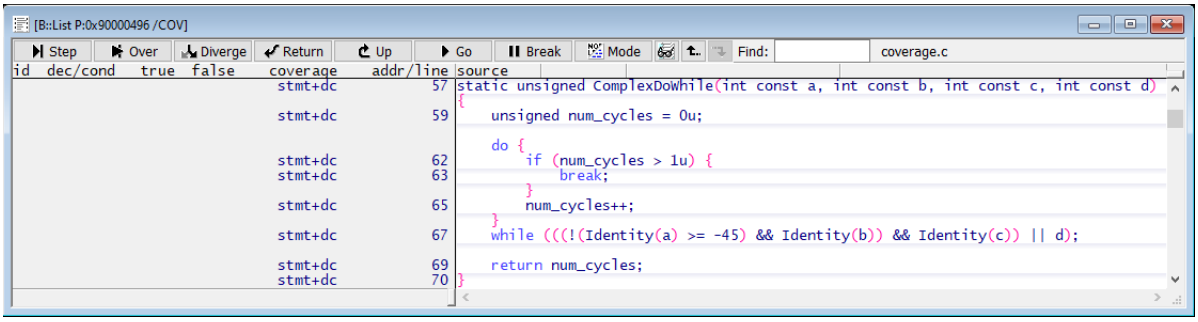
COVerage.ListFunc

Use the following command to display source code tagging:

List.Hll /COVerage

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

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



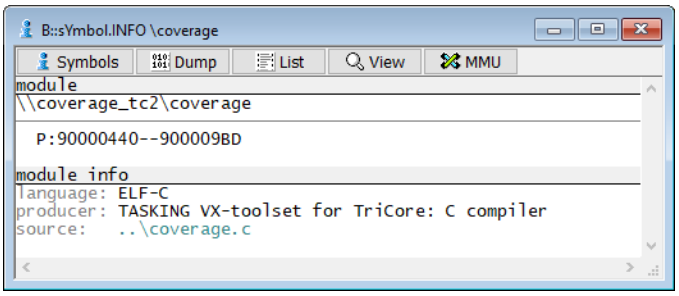
Source code lines are tagged as follows:

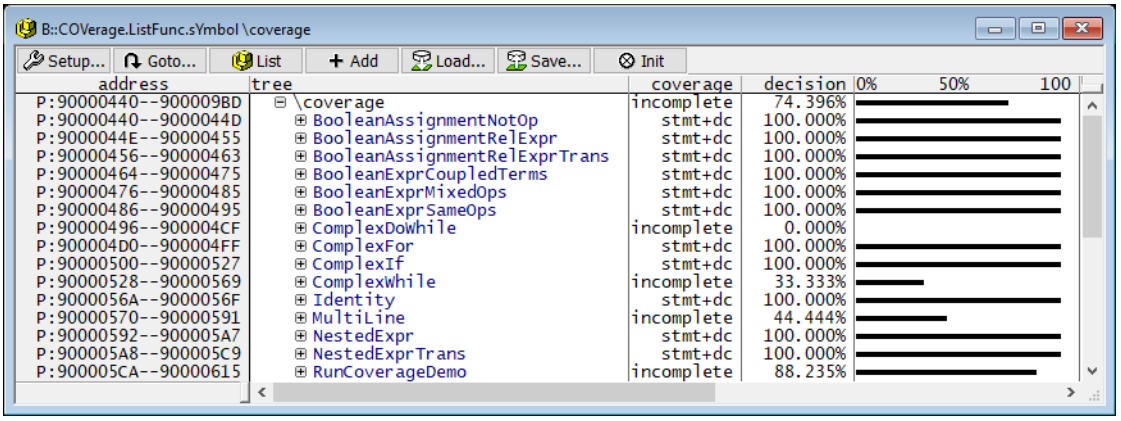
stmt+dc	The source code line has achieved full object code coverage, ensuring either decision or statement coverage.
incomplete	The source code line has not achieved full object code coverage, so it cannot be determined whether statement or decision coverage has been met.

Object code instructions receive ObjectCode tagging when ocb decision coverage is applied.

This TRACE32 command provides a tabular analysis of all functions within the 'coverage' module. Typically, a module corresponds to a source code file.

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

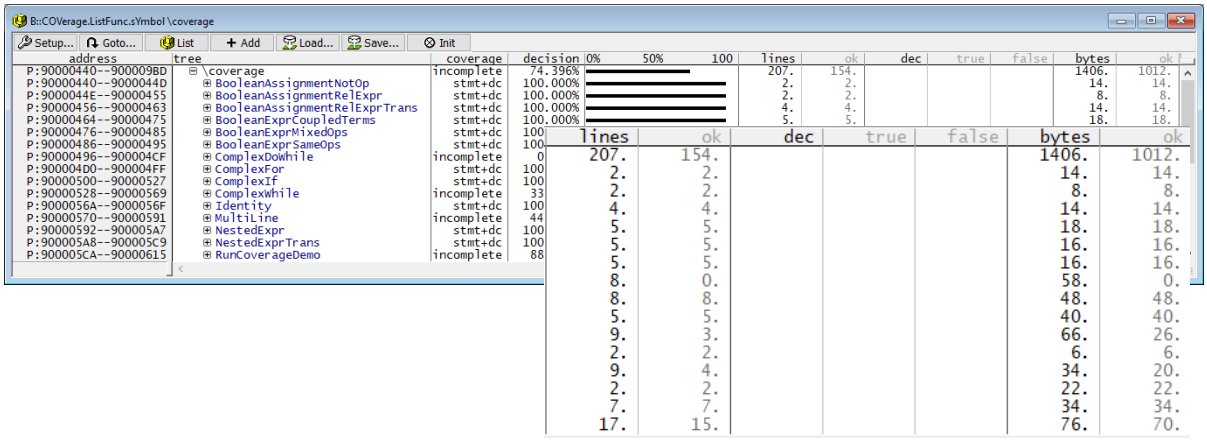




Tags for Object Code Based (ocb) Decision Coverage

- **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:



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

Example Script

```
// Demo script "~~/demo/coverage/mcdc/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
```

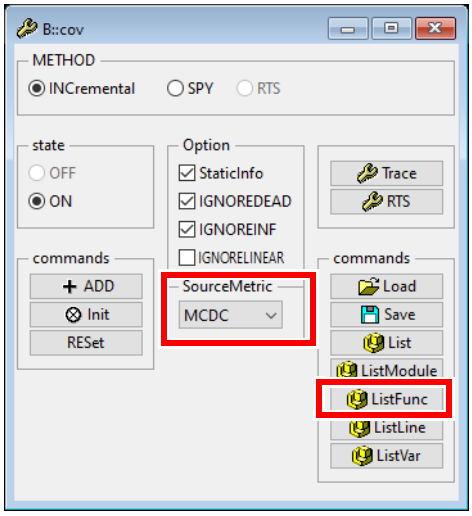
For metrics such as statement coverage, full decision coverage, condition coverage, MC/DC, as well as call and function coverage, the results of various code coverage measurements can only be merged and prepared for evaluation in a web browser outside of TRACE32 using the Lauterbach command line tool, t32covtool. However, you can always evaluate the results of a single code coverage measurement directly in TRACE32 if needed. In this context, we refer to this as an **intermediate result obtained after the completion of a single measurement**.

If you use continuous code coverage, you can display intermediate results during trace recording. In this case, we refer to these as **intermediate results obtained during a single measurement**.

To inspect intermediate results, proceed as follows:

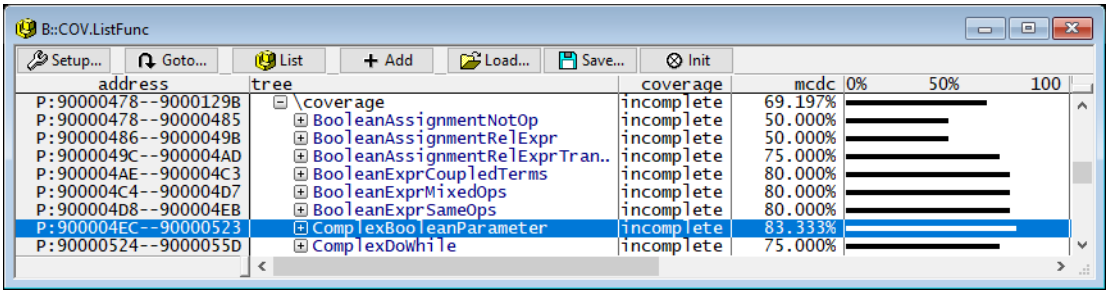
1. **Select the desired code coverage metric.**

   The TRACE Code Coverage System includes results for all available metrics. You can select the metric you are interested in either from the **COVerage configuration window** or by using the command **COVerage.Option.SourceMetric** *<metric>*. For example, to use the MC/DC metric, the command would be **COVerage.Option.SourceMetric MCDC**.



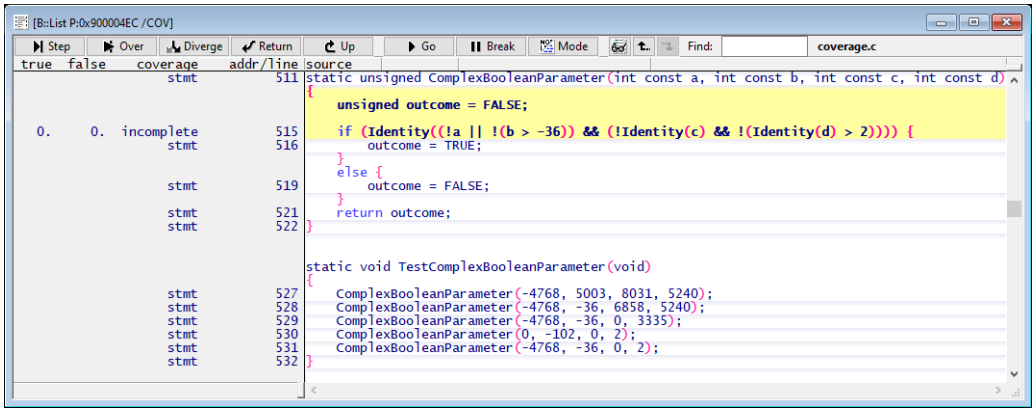
2. **Get an overview of the code coverage for all function.**

   Use the **ListFunc button** in the **COVerage configuration window** or the command **COVerage.ListFunc** to get your overview.



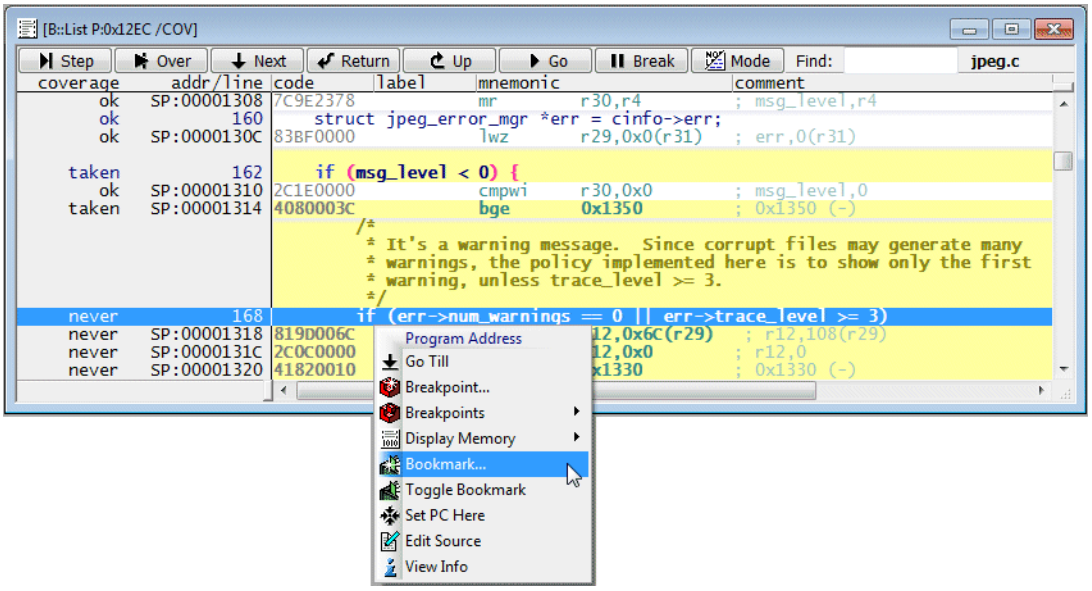
3. Drill down into the details of a specific function.

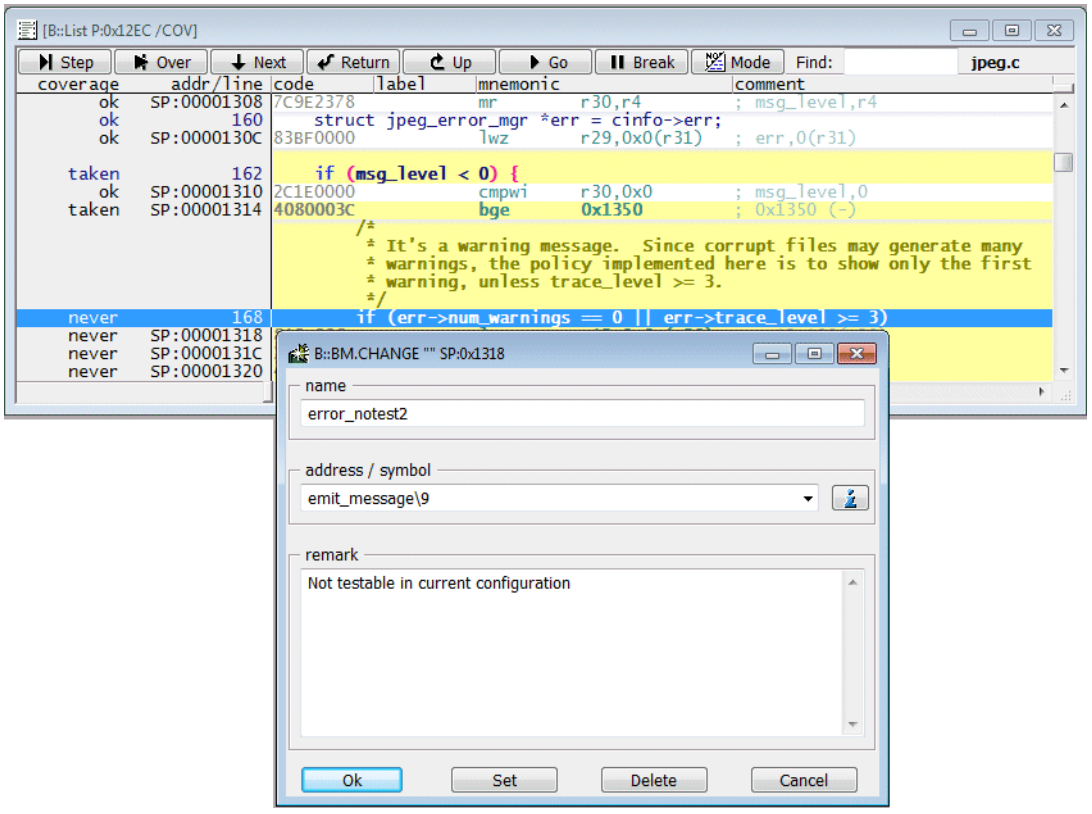
Double-click the function you are interested in, or use the command **List.HII** *<func_name>* **/COVerage** to view the details.



Comment Your Results

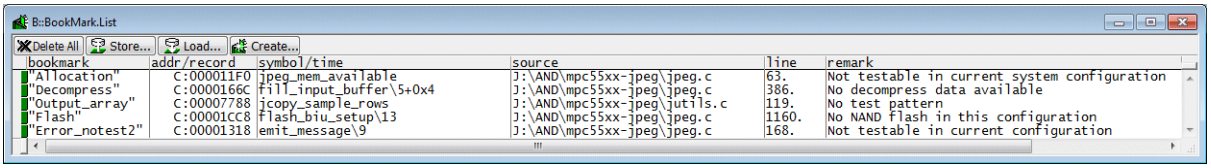
Address-based bookmarks can be used to comment not covered code ranges, which are fine but not testable in the current system configuration.





List all bookmarks:

BookMark.List



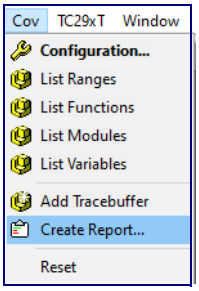
The current bookmarks can be saved to a file and reloaded later on.

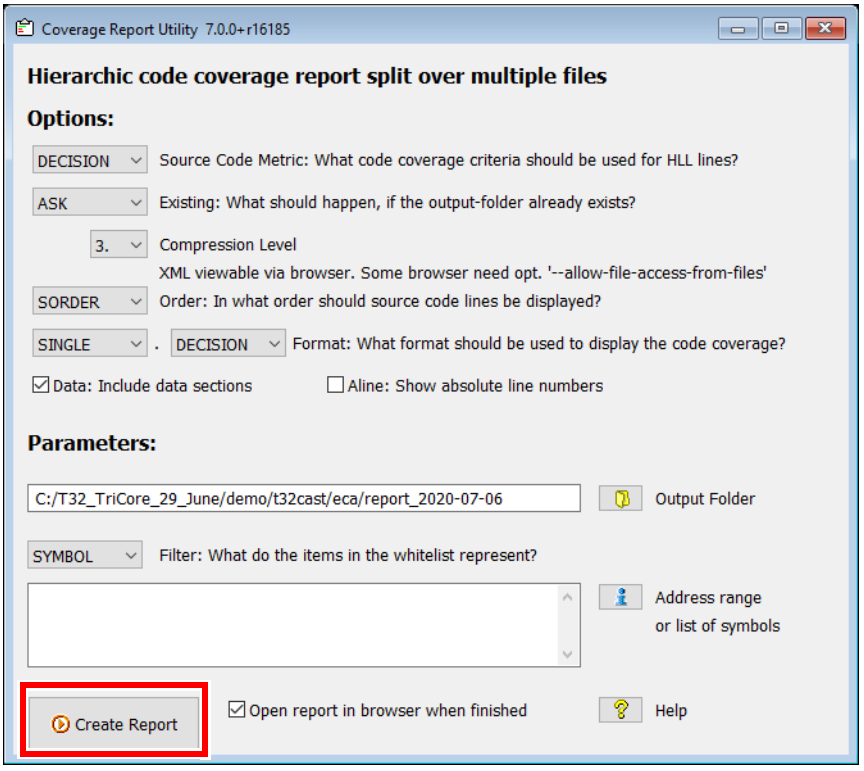
STOre *<file>* **BookMark**

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**.





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: Merge Multiple Object Code Based Measurements

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 merge code coverage measurements 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
```

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
```

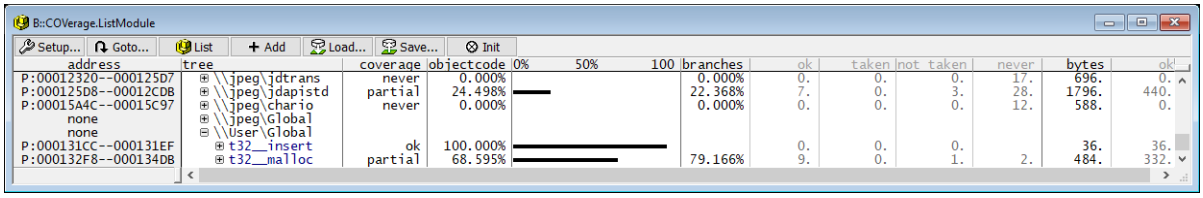
Object Code Coverage

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.



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

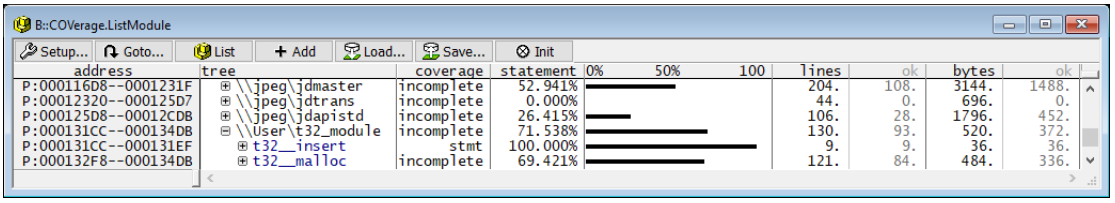
Code that is not part of a source code function is discarded for coverage. If you want to include this code you have to assign a function to it:

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)
```



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

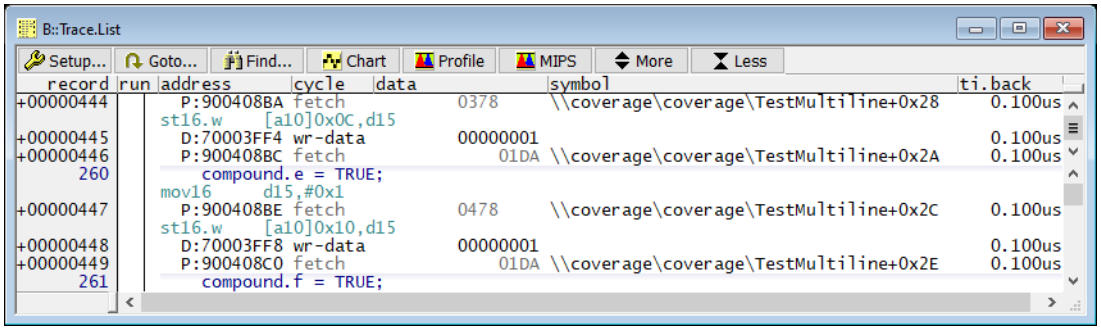
Metric	Tag	Description
all source code metrics	incomplete	At least one assembler line within the function is tagged with never, taken or not taken.
Statement	stmt	All assembler lines are tagged with ok.

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.

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.



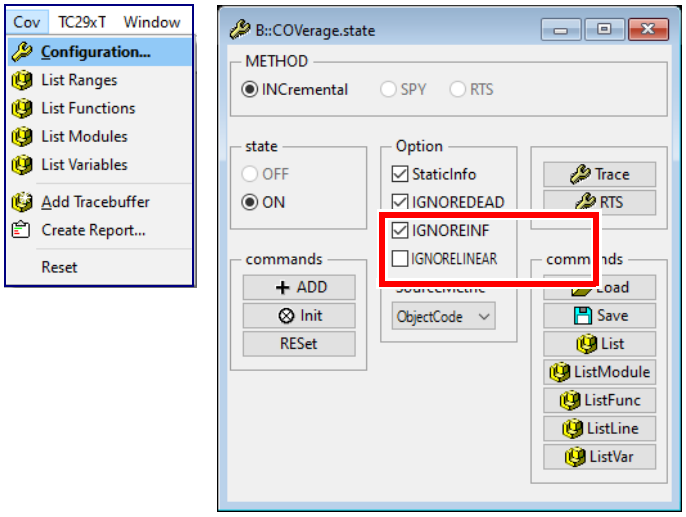
If you want to use an onchip trace or an offchip trace port for data tracing, please refer to the following documents for setup details:

- Arm: **"Training Cortex-M Tracing"** (training_cortexm_etm.pdf)
- MPC5xxx/SPC5xxx, QorIQ and RH850: **"Training MPC5xxx/SPC5xx Nexus Tracing"** (training_nexus_mpc5500.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 Measurement in Leash Mode"**, page 83).

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

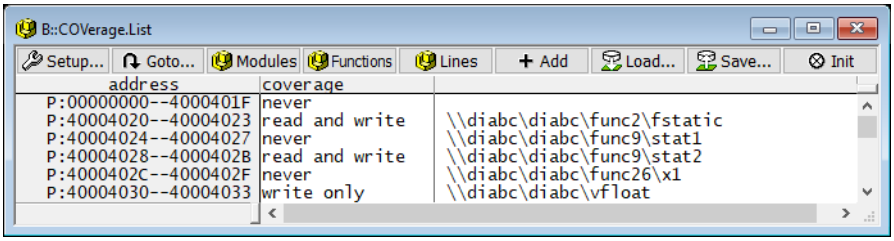
COVerage.ListVar

The following command shows the tagging per address.

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

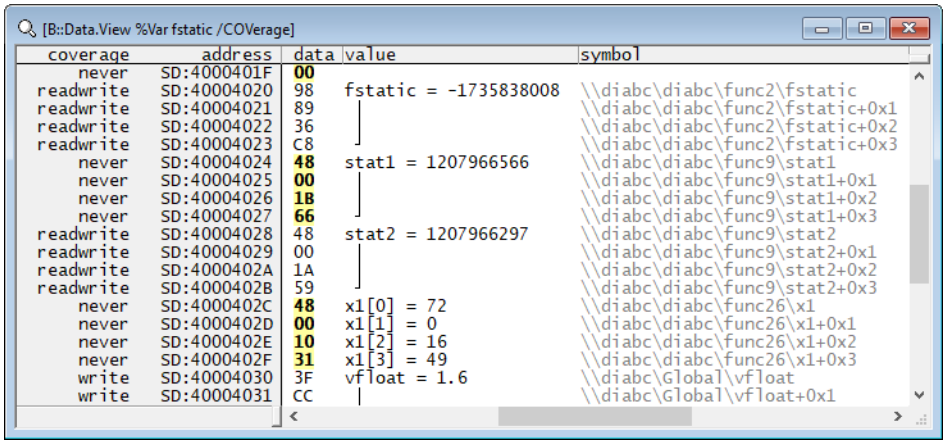
This TRACE32 command shows the coverage tagging on address range level:

```
COVerage.List
```



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

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

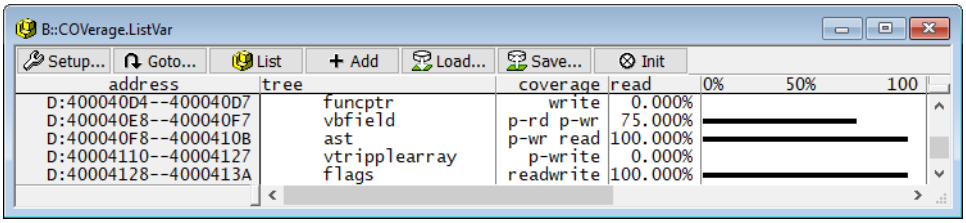


The data addresses are tagged as follow:

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
```



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

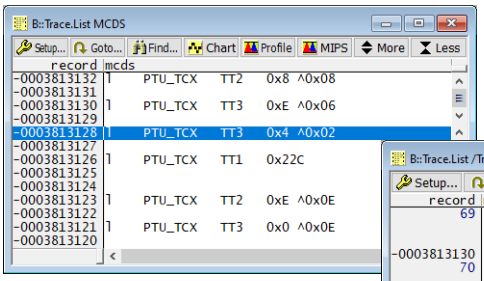
readwrite	Read and write accesses were performed for all addresses within the address range of the variable.
read	Only read accesses were performed for all addresses within the address range of the variable.
write	Only write accesses were performed for all addresses within the address range of the variable.
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.

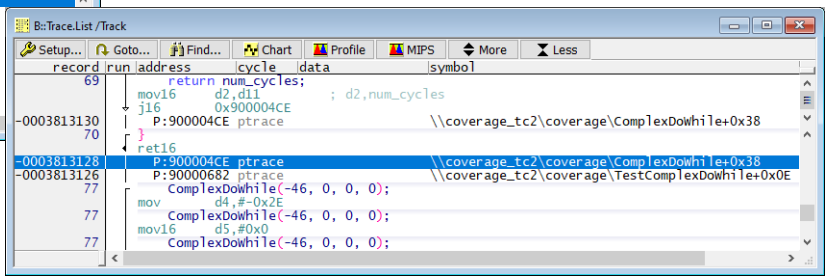
A complete list of all data coverage tags can be found in **"Appendix H: Data Coverage in Detail"**, page 149.

Appendix E: Trace Decoding in Detail

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



Raw trace data



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.

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

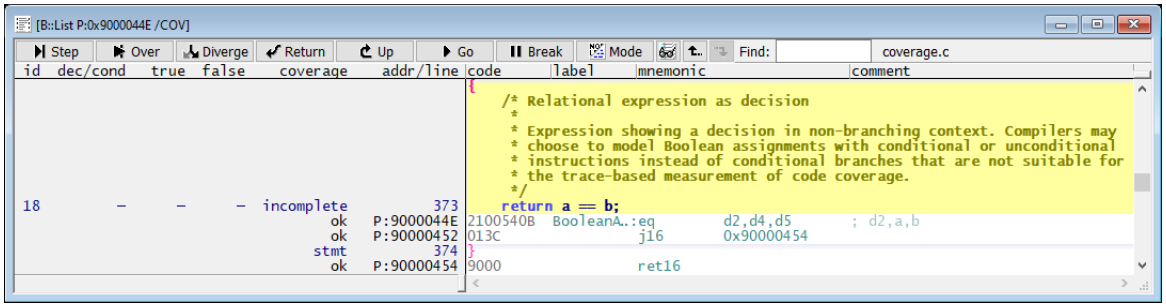
The following coding guidelines are recommended for full decision and condition coverage as well as for MC/DC. If you follow these coding guidelines you avoid false negative results. False negative means that a decision/conditions is tagged as incomplete although coverage has already been achieved.

Nevertheless, it is possible that the compiler itself generates such constructs at high optimization levels.

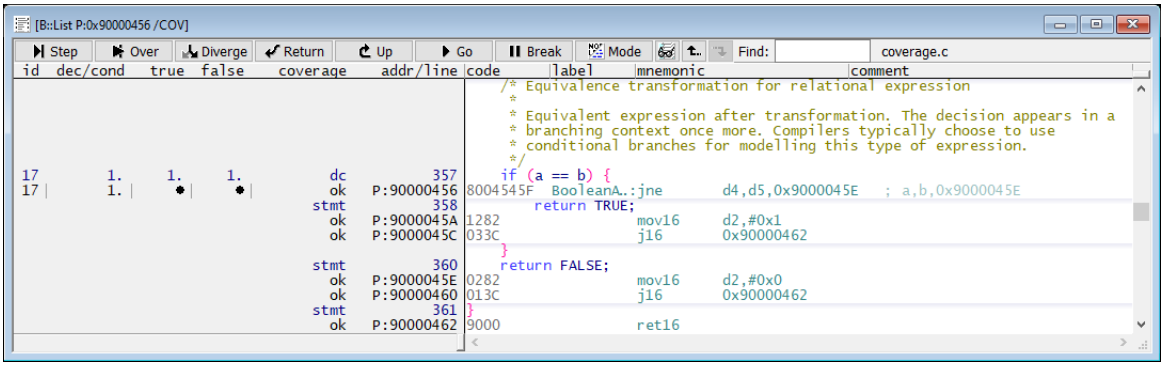
Avoid Simple Decisions in Assignment Context

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`.



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.



A few examples:

```
; source code not suitable for
; trace-based code coverage

return a == b;
```

```
; source code suitable for
; trace-based code coverage

if (a == b) {
     return TRUE;
}
return FALSE;
```

```
; source code not suitable for
; trace-based code coverage

identity(a != b);
```

```
; source code suitable for
; trace-based code coverage

tmp = FALSE;
if (a != b) {
     tmp = TRUE;
}
identity(tmp);
```

```
; source code not suitable for
; trace-based code coverage

return (a >= b) ? a : b;
```

```
; source code suitable for
; trace-based code coverage

if (a >= b) {
     return a;
}
return b;
```

Avoid Nesting of Decisions

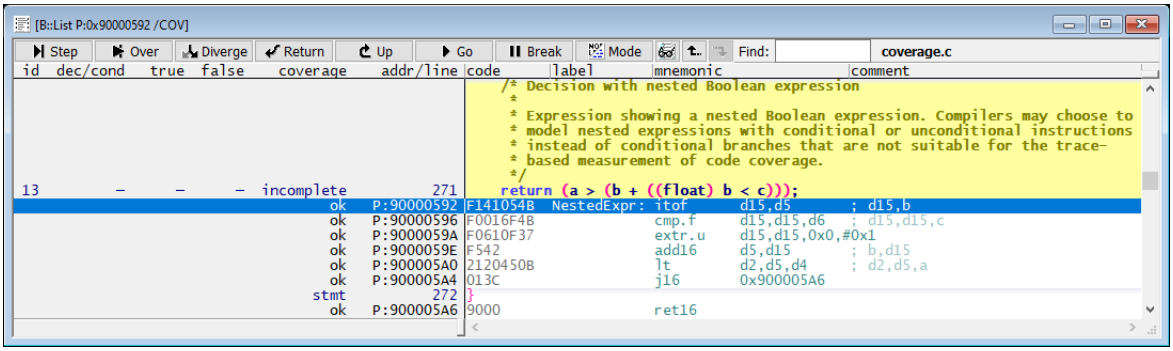
It is very likely that not all conditions are represented by a conditional branch/instruction at object code level.

This is illustrated by the following example:

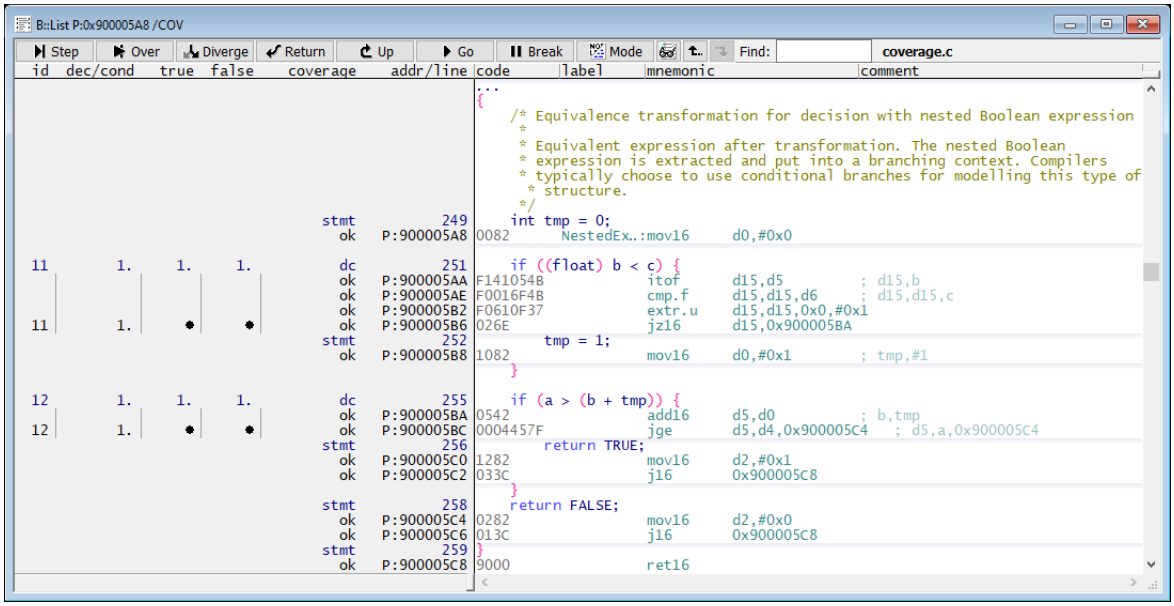
```
; source code not suitable for
; trace-based code coverage

return a > (b + (b && c));
```

```
; source code suitable for
; trace-based code coverage

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.



If the code is written in a way that suits for trace-based code coverage, all necessary conditional branches/instructions were generated.



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.

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.