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 06-Jun-2024

History

04-Jun-24	Chapter 'Build Process for Statement Coverage' added.
29-May-24	Subchapter 'Evaluation of Switch Case Statements' added to chapter 'MC/DC, Condition and Decision Coverage'.
26-Jan-24	The manual has been completely revised to integrate the new code coverage modes targeted and full instrumentation.
07-Sep-23	EN50128 (railway) added to the chapter '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-20	Initial version of the manual.

Intended Audience

Developers who want to:

- Collect code coverage data
- Perform code coverage on collected trace data
- Generate reports based upon this data

Although this is a generic manual, the screenshots were always made with a TriCore™ AURIX™ TC297T, if nothing else is mentioned. Deviations from screen displays are likely in your target environment.

The manual is written in such a way that it is sufficient to only read the relevant chapters. If you read the manual completely, this may lead to redundancies.

Supported Code Coverage Metrics

TRACE32 supports the following code coverage metrics:

- **Code coverage metric statement coverage**

  Statement coverage ensures that every statement in the program has been invoked at least once. Statement in this context means block of source code lines.

- **Code coverage metric 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.

- **Code coverage metric condition coverage**

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

- **Code coverage metric 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.

- **Code coverage metric function coverage**

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

- **Code coverage metric call coverage**

  Every function call has been executed at least once.

- **Code coverage metric 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**.

Trace-Based Code Coverage

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. Testing must be conducted with an executable that has a low level of compiler optimization.

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

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

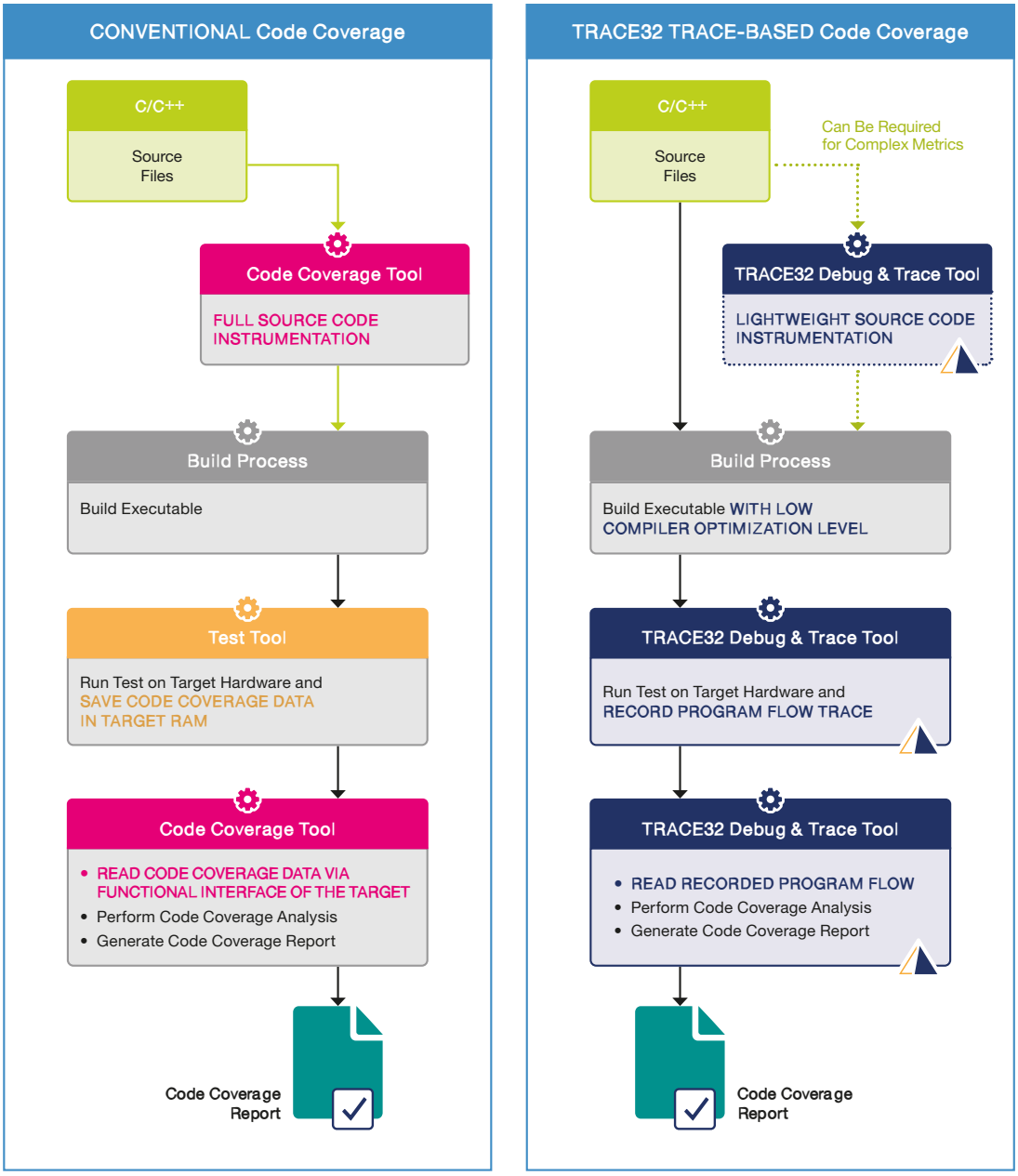


Figure: Workflow comparison, conventional code coverage vs. TRACE32 trace-based code coverage.

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.

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.

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

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

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

- **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 is always advisable to contact your Lauterbach sales office.

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

There is also the option of performing the code coverage analysis with a TRACE32 Instruction Set Simulator (**Trace.METHOD Analyzer**). 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 an Instruction Set Simulator for the core architecture you are using, you can also use the TRACE32 Advanced Register Trace (**Trace.METHOD ART**). This is a single-step trace, which makes program execution very slow. This method is therefore only suitable for unit testing.

TRACE32 Debuggers for virtual targets (**Trace.METHOD Analyzer**) should, because of their limitations, only be used for code coverage if needed. For details refer to **"Code Coverage with Virtual Targets"**, page 75.

Test Variants

TRACE32 offers two variants for code coverage analysis:

Incremental Code Coverage

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

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

Live Code Coverage

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

Live code coverage requires simpler scripts and is naturally faster due to the simultaneity of the steps. However, it only works up to a certain bandwidth.

Incremental code coverage requires more complex scripts and is slower. However, it has the advantage that it always works and that the two steps can be carried out by different teams.

Merge Results and Generate Report

Typically, code coverage is not measured in a single test run, but is approached gradually. This creates the need to combine multiple results into one final report. TRACE32 offers the possibility to merge results and to create an HTML report for all supported code coverage metrics.

For these metrics, analyzing code coverage is more challenging, which is why TRACE32 supports these metrics with multiple code coverage modes. Here is the background information on this.

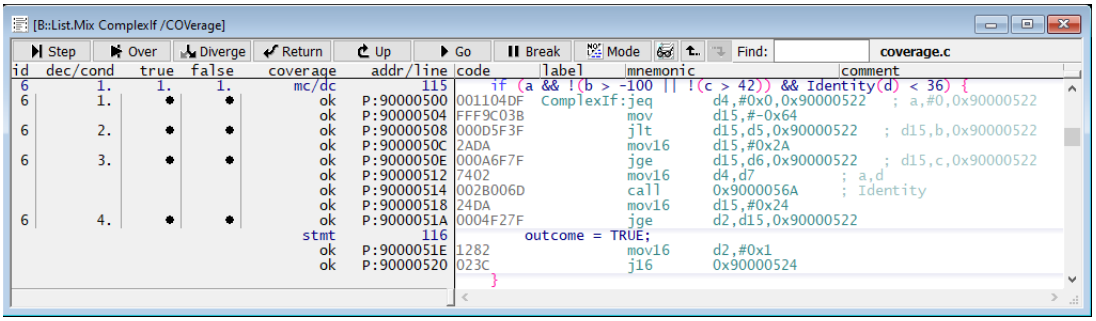
Multiple Code Coverage Modes

Preconditions for a Trace-Based Code Coverage

Four criteria must be met for MC/DC, condition or decision coverage analysis based on the recorded program flow:

1. TRACE32 has to know the structure and the position of the conditions/decisions within the source code. Since the conditions/decisions details are not included in the debug information generated by the compiler, Lauterbach offers its own Clang-based command line tool named t32cast for this purpose. t32cast analyzes the C/C++ sources and generates an extended code analysis (.eca) file for each source file, that provides the conditions/decisions details.
2. Decisions are composed of one or more (atomic) conditions. And each condition in the source code must be represented by a conditional branch or by a conditional instruction at object code level.
3. An exact mapping of the conditions/decisions in the source code to the conditional branches/instructions in the object code is required.
4. Conditional branches/instructions in the recorded program flow trace must allow to observe whether a source code condition was evaluated true or false.

The figure below illustrates what has been described using MC/DC analysis as an example.



Practice has shown that criteria 2, 3 and 4 are not always fulfilled in every test scenario. When this is the case, Lauterbach speaks of observabiltiy gaps.

Occurring Observability Gaps

Observabiltiy gap means that TRACE32 cannot monitor whether a condition has been evaluated as true or false at a certain point in the program flow trace. In this case, no code coverage result can be displayed for the related decision. The code coverage is incomplete if these gaps are not closed by other means.

Many observability gaps can be avoided from the first place by writing code coverage friendly code (please refer to **"Appendix F: Coding Guidelines"**, page 148 for details) and through a moderate compiler optimization level. Moderate optimization also has the advantage of making the code coverage analysis results clear and intuitive for the user to read.

Depending on the number of observability gaps, the following code coverage modes are available:

- **No gaps**

  The four criteria are fully met. TRACE32 only requires the recorded program flow for the code coverage analysis. Lauterbach has named this *Trace-Based Code Coverage/No Instrumentation*.

- **Moderate number of gaps**

  With a moderate number of observability gaps, Lauterbach recommends inspecting them first and then deciding whether the gaps need to be closed.

  To close individual gaps, TRACE32 has the following code coverage mode:

  - *Trace-Based Code Coverage/Targeted Instrumentation*

- **Large number of gaps**

  A large number of gaps can have different causes: High compiler optimization level, an exotic core architecture, not yet supported core/compiler pairing. A detailed overview of the possible causes of observability gaps can be found in chapter **"Causes for Observability Gaps: An Overview"**, page 18.

  In the case of a high compiler optimization level, the following consideration must be made:

  - If you want to keep high compiler optimization level, Lauterbach recommends *Trace-Based Code Coverage/Full Instrumentation* which results in many instrumentation sites. This makes the progam code larger and has an impact on the program runtime.

    Technically, however, full instrumentation is simple, it gets by with two hook functions. This makes further compiler optimizations possible.

  - You can reduce the compiler optimization level. This makes the program code slightly larger and therefore requires a little more program runtime. But the number of observability gaps should decrease enough to be able to use *Trace-Based Code Coverage/Targeted Instrumentation* with fewer instrumentation sites.

    Please note that one hook function pair per observation gap is required for targeted instrumentation within each function. This means that multiple hook functions are required. Of course, this requires a little more memory for the instrumentation.

  It is possible that the program code size for full instrumentation and targeted instrumentation is approximately the same, in which case both instrumentations are equivalent.

TRACE32 uses only body-less hook functions for instrumentation, whose calls are visible in the recorded program flow. These are used to monitor whether an instrumented source code condition has been evaluated as true or false

This form of instrumentation does not require any data memory.

The following code coverage modes result from what was described in the previous chapter.

Code Coverage Mode No Instrumentation

Since instrumentation is not used, code size and runtime remain the identical. The build process does not need to be touched.

Code Coverage Mode Full Instrumentation

All decisions in the user application are instrumented so that TRACE32 can fully monitor them. This results in a high number of instrumentation sites. Body-less instrumentation hook functions result in a moderately larger code and a modest impact on runtime behavior.

Trace-Based Code Coverage/Full Instrumentation, however, requires an adaptation of the build process.It is very robust, and therefore serves as fall-back.

Code Coverage Mode Targeted Instrumentation

Only the decisions for which an observability gap has been detected are instrumented so that TRACE32 can monitor them and thus close the gaps. This results in a small number of instrumentation sites. Body-less instrumentation hook functions result in a slightly larger code and a small impact on runtime behavior.

Trace-Based Code Coverage/Targeted Instrumentation, however, requires a more complex build process.

A Comparison of the Different Code Coverage Modes

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

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

For practical performance, we have decided to pool *Trace-Based Code Coverage/Targeted Instrumentation* and *Trace-Based Code Coverage/No Instrumentation* in this manual. This is based on the following considerations:

- Source code for which no observability gaps were initially detected can lead to observability gaps by adding new lines of code.
- Source code for which a few observability gaps were initially detected may no longer contain any observability gaps after lines of code are deleted or modified.

Causes for Observability Gaps: An Overview

Finally, to close the chapter for those who are interested, here is an overview of the causes of 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 is not to be met. Observabiltiy gaps that have occurred need to be addressed.

Macros

A macro that is used in a decision/condition can in itself contain decisions/conditions. The compiler expands all macros before compilation and handles the expanded statement as a single source block. During this step the source code locations of the decisions/conditions inside the macro are lost. In this case, criterion 3 is violated. A mapping of the inside-macro-decisions to the conditional branches / instructions is no longer possible. Observability gaps that have occurred need to be addressed.

Highly-optimized code

Highly-optimized code is not recommended for trace-based code coverage analysis. For one, individual conditions may not be represented by conditional branches/instructions at the object code level. Criterion 2 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 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 is violated.

Instruction set complexity

The challenges described in 1-4 are essentially the ones faced by cores with general-purpose RISC architecture. However, complex SoCs also contain coprocessors and special-purpose cores for which an instruction trace is generated. Examples are DSPs, configurable cores with user-defined instructions, timer IP and many more. Here, TRACE32 must always be specially adapted to the instruction set. In this respect, it is always advisable to check with Lauterbach in good time.

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;
}
```

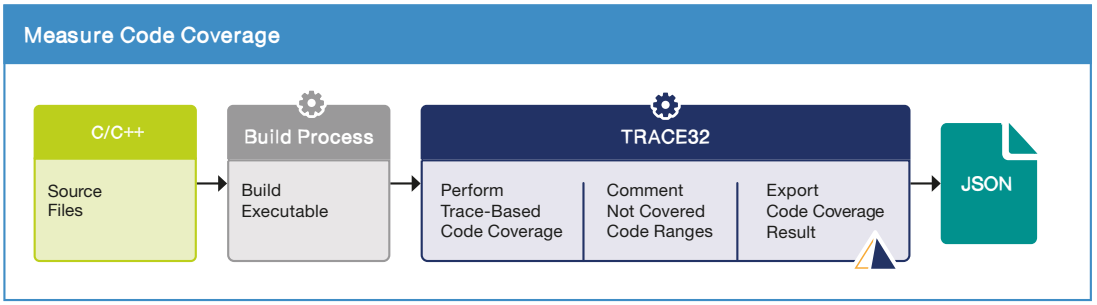
Please note: In contrast to the original switch case statement, the converted if-then expression achieves complete code coverage only when color had both the values YELLOW and GREEN.

Code Coverage Workflow

General Workflow

Measure Code Coverage

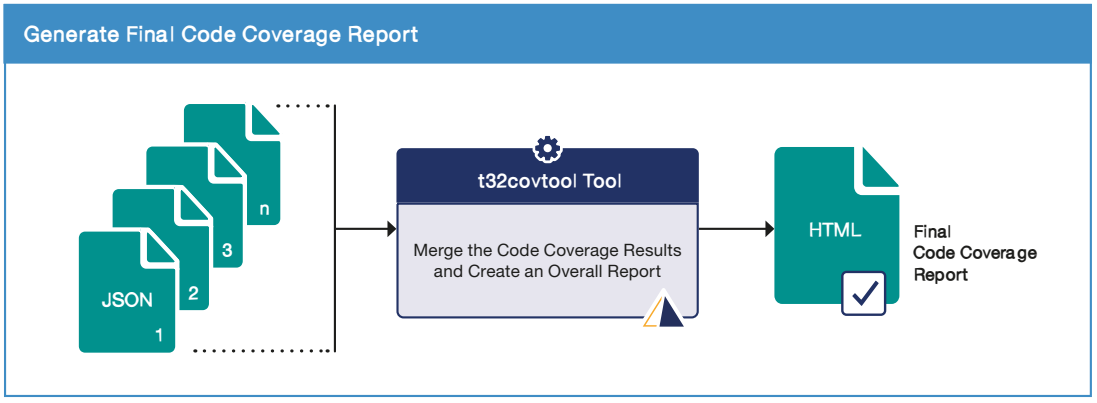
The basic workflow for a code coverage test pass with TRACE32 is as follows:



TRACE32 provides a reporting tool for a detailed report on a single code coverage measurement. See **"Appendix A: TRACE32 Coverage Report Utility"**, page 131.

Merge Measurement Results

Typically, code coverage is not measured in a single pass, but is approached gradually. This creates the need to combine multiple exports into one final report. Lauterbach provides the **t32covtool** utility for this purpose.



Object Code Coverage Workflow

If you want to perform an object code coverage analysis, you must carry out the following steps:

1. Build the executable.

Two steps are necessary for the object code coverage itself:

2. Load all files needed into TRACE32, see **"Preparation for Object Code Coverage"**.
3. Choose between the two test variants.

   Live code coverage (RTS, SPY): The code coverage analysis is already performed while the execution of the program is running.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **"Trace Data Collection Overview"**, page 43. Details on the individual test variants can be found in **"Trace Data Collection"**, page 57.

   Details on the object code coverage evaluation itself can be found in **"Object Code Coverage Evaluation"**, page 82.
4. Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 127.
5. Generate a code coverage report, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 131. If you want to merge the results of several test passes before generating a report, see **"Appendix B: Assemble Multiple Test Runs at Address Level"**, page 133.

If you want to perform a statement coverage analysis, you must carry out the following steps.

1. Build the executable. Please pay attention to **“Build Process Statement Coverage”**, page 30.

Two steps are necessary for the statement coverage itself:

2. Load all files needed for statement coverage into TRACE32, see **“Preparation for Statement Coverage”**, page 51.

3. Choose between the two test variants.

   Live code coverage (RTS, SPY): The code coverage analysis is already performed while the execution of the program is running.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **“Trace Data Collection Overview”**, page 43. Details on the individual test variants can be found in **“Trace Data Collection”**, page 57.

   Details on the statement coverage evaluation can be found in **“Statement Coverage Evaluation”**, page 87.

4. Add comments to the uncovered code ranges, see **“Comment Your Results”**, page 127.

5. To generate a code coverage report, see **“Appendix A: TRACE32 Coverage Report Utility”**, page 131. If you want to merge the results of several test passes before generating a report, see **“TRACE32 Merge and Report Tool”**, page 129.

Before you start with the decision coverage analysis, you should have read the chapter **"MC/DC, Condition and Decision Coverage"**, page 15.

If you want to perform a decision coverage analysis, you must carry out the following steps.

1. **Decide on the Appropriate Code Coverage Mode**:

   - *Targeted Instrumentation/No Instrumentation* or

   - *Full Instrumentation*

2. Generate all files needed for decision coverage, see **"Build Process Decision Coverage"**. Please pay attention to **"Recommendations for the Build Toolchain"**, page 30.

Two steps are necessary for the decision coverage itself:

3. Load all files needed for decision coverage into TRACE32, see **"Preparation for Decision Coverage"**. Read the sub-chapter on the code coverage mode that you decided to use.

4. Choose between the two test variants.

   Live code coverage (SPY): The code coverage analysis is already performed while the execution of the program is running. RTS mode cannot be used for decision coverage at present.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **"Trace Data Collection Overview"**, page 43. Details on the individual test variants can be found in **"Trace Data Collection"**, page 57.

   Details on the decision coverage evaluation can be found in **"Full Decision Coverage Evaluation"**, page 91.

5. Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 127.

6. To generate a code coverage report, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 131. If you want to merge the results of several test passes before generating a report, see **"TRACE32 Merge and Report Tool"**, page 129.

Object Code Based (ocb) Decision Coverage Workflow

If you want to perform object code based decision coverage analysis, you must carry out the following steps.

1. Build the executable. Please pay attention to **Build Process ocb Decision Coverage**.

Two steps are necessary for the ocb decision coverage itself:

2. Load all files needed for ocb decision coverage into TRACE32, see **"Preparation ocb Decision Coverage"**.
3. TRACE32 basically offers two variants of code coverage analysis:

   Live code coverage (RTS, SPY): The code coverage analysis is already performed while the execution of the program is running.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **"Trace Data Collection Overview"**, page 43. Details on the individual test variants can be found in **"Trace Data Collection"**, page 57.
4. Details on the ocb decision coverage evaluation can be found in **"Object Code Based (ocb) Decision Coverage Evaluation"**, page 97.
5. Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 127.
6. Generate a code coverage report, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 131. If you want to merge the results of several test passes before generating a report, see **"Appendix B: Assemble Multiple Test Runs at Address Level"**, page 133.

Before you start with the condition coverage analysis, you should have read the chapter **"MC/DC, Condition and Decision Coverage"**, page 15.

If you want to perform a condition coverage analysis, you must carry out the following steps.

1. **Decide on the Appropriate Code Coverage Mode**:

   - *Targeted Instrumentation/No Instrumentation* or

   - *Full Instrumentation*

2. Generate all files needed for condition coverage, see **"Build Process Condition Coverage"**. Please pay attention to **"Recommendations for the Build Toolchain"**, page 30.

Two steps are necessary for the condition coverage itself:

3. Load all files needed for the condition coverage into TRACE32, see **"Preparation for Condition Coverage"**. Read the sub-chapter on the code coverage mode that you decided to use.

4. TRACE32 basically offers two variants of code coverage analysis:

   Live code coverage (SPY): The code coverage analysis is already performed while the execution of the program is running. RTS mode cannot be used for condition coverage at present.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **"Trace Data Collection Overview"**, page 43. Details on the individual test variants can be found in **"Trace Data Collection"**, page 57.

   Details on the condition coverage evaluation can be found in **"Condition Coverage Evaluation"**, page 104.

5. Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 127.

6. To generate a code coverage report, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 131. If you want to merge the results of several test passes before generating a report, see **"TRACE32 Merge and Report Tool"**, page 129.

Before you start with the MC/DC analysis, you should have read the chapter **"MC/DC, Condition and Decision Coverage"**, page 15.

If you want to perform a MC/DC analysis, you must carry out the following steps:

1. **Decide on the Appropriate Code Coverage Mode**:

   - *Targeted Instrumentation/No Instrumentation* or

   - *Full Instrumentation*

2. Generate all files needed for MC/DC, see **"Build Process MC/DC"**. Please pay attention to **"Recommendations for the Build Toolchain"**, page 30.

Two steps are necessary for MC/DC itself:

3. Load all files needed for the MC/DC into TRACE32, see **"Preparation for MC/DC"**. Read the sub-chapter on the code coverage mode that you decided to use.

4. TRACE32 basically offers two variants of code coverage analysis:

   Live code coverage (SPY): The code coverage analysis is already performed while the execution of the program is running. RTS mode cannot be used for MC/DC at present.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **"Trace Data Collection Overview"**, page 43. Details on the individual test variants can be found in **"Trace Data Collection"**, page 57.

   Details on MC/DC evaluation can be found in **"Modified Condition/Decision Coverage (MC/DC) Evaluation"**, page 110.

5. Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 127.

6. To generate a code coverage report, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 131. If you want to merge the results of several test passes before generating a report, see **"TRACE32 Merge and Report Tool"**, page 129.

If you want to perform a function coverage analysis, you must carry out the following steps:

1. Generate all files needed for function coverage, see **"Build Process Function Coverage"**. Please pay attention to **"Recommendations for the Build Toolchain"**, page 30.

Two steps are necessary for the function coverage itself:

2. Load all files needed for the function coverage into TRACE32, see **"Preparation Function,Coverage"**, page 45.
3. TRACE32 basically offers two variants of code coverage analysis:

   Live code coverage (RTS, SPY): The code coverage analysis is already performed while the execution of the program is running.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **"Trace Data Collection Overview"**, page 43. Details on the individual test variants can be found in **"Trace Data Collection"**, page 57.

   Details on the function coverage evaluation can be found in **"Function Coverage Evaluation"**, page 116.
4. Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 127.
5. To generate a code coverage report, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 131. If you want to merge the results of several test passes before generating a report, see **"TRACE32 Merge and Report Tool"**, page 129.

If you want to perform a call coverage analysis, you must carry out the following steps.

1. Generate all files needed for call coverage, see **"Build Process Call Coverage"**, page 32.

Two steps are necessary for the call coverage itself:

2. Load all files needed for the call coverage into TRACE32, see **"Preparation for Call Coverage"**, page 52.
3. TRACE32 basically offers two variants of code coverage analysis:

   Live code coverage (RTS, SPY): The code coverage analysis is already performed while the execution of the program is running.

   Incremental code coverage: First start and stop the program execution to collect trace data and then perform the code coverage analysis based on the collected data. Repeat these steps until sufficient data are collected.

   Decision-making aid and further tips can be found in **"Trace Data Collection Overview"**, page 43. Details on the individual test variants can be found in **"Trace Data Collection"**, page 57.

   Details on the call coverage evaluation can be found in **"Call Coverage Evaluation"**, page 120.
4. Add comments to the uncovered code ranges, see **"Comment Your Results"**, page 127.
5. To generate a code coverage report, see **"Appendix A: TRACE32 Coverage Report Utility"**, page 131. If you want to merge the results of several test passes before generating a report, see **"TRACE32 Merge and Report Tool"**, page 129.

Recommendations for the Build Toolchain

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): **`-O0 -fno-jump-tables`**
- Wind River Diab Compiler: **`-Xoptimized-debug-off -Xdebug -source-line-barriers-on -Xswitch-table-off`**

Build Process Statement Coverage

Apart from the **"Recommendations for the Build Toolchain"**, page 30, there are no additional recommendations for the build process here. However, TRACE32 currently does not consider two special cases.

Let's review the definition of statement coverage: "“The statement coverage ensures that each statement in the program has been called at least once. In this context, an instruction is a block of source code lines.” There are two cases that must be taken into account:

- **The compiler does not generate object code for a source code line**

  Optimizations may cause the compiler to omit object code for certain source code lines. However, TRACE32's code coverage analysis depends on the object code, as only this is captured in the program flow trace recording. Source code lines are tagged for statement coverage based on a suitable mapping between the object code and the source code.

  Here is a small source code example where the compiler could optimize by generating a single conditional branch for the two source code lines:

```
if (terminate == TRUE)
     break;
```

  After loading the program, TRACE32 does not display line numbers for source code lines without corresponding object code. These lines are ignored in the statement coverage analysis, which may lead to inaccuracies. Therefore, we recommend verifying the results by manually inspecting the source code.

- **The compiler uses conditional instructions to handle simple conditions**

```
if (a == 5)
     b =7 ;

            CMP R3, #5              CMP   R3, #5
            BNE not_equal           MOVEQ R4, #7
            MOV R4, #7
not_egual:
```

In this case, it is advisable to first verify whether the trace protocol of the core under debug supports conditional instructions, specifically indicating if the condition code check passed or failed. You can use the **COVerage.INFO** command or the **CPU.Feature**(CONDTRACE) function to do this.

- If the trace protocol does not support conditional instructions, statement coverage cannot be performed for the affected source code lines.

- Even if the trace protocol supports conditional instructions, you must ensure that object code is generated for every source code line. Otherwise, you will encounter the problem described in the previous section under 'The compiler does not generate object code for a source code line.

Build Process Function and ocb Decision Coverage

The following recommendations apply here for the build process:

- Function Coverage

  It is recommended to disable function inlining so that the results are clear and intuitively readable.

- ocb Decision Coverage

  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 148.

Apart from that, the executable can be generated as usual.

Build Process Call Coverage

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

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

All input/outputs of the build process that are required for the call coverage analysis are marked in figure “Build Process Call Coverage” with an arrow pointing downwards.

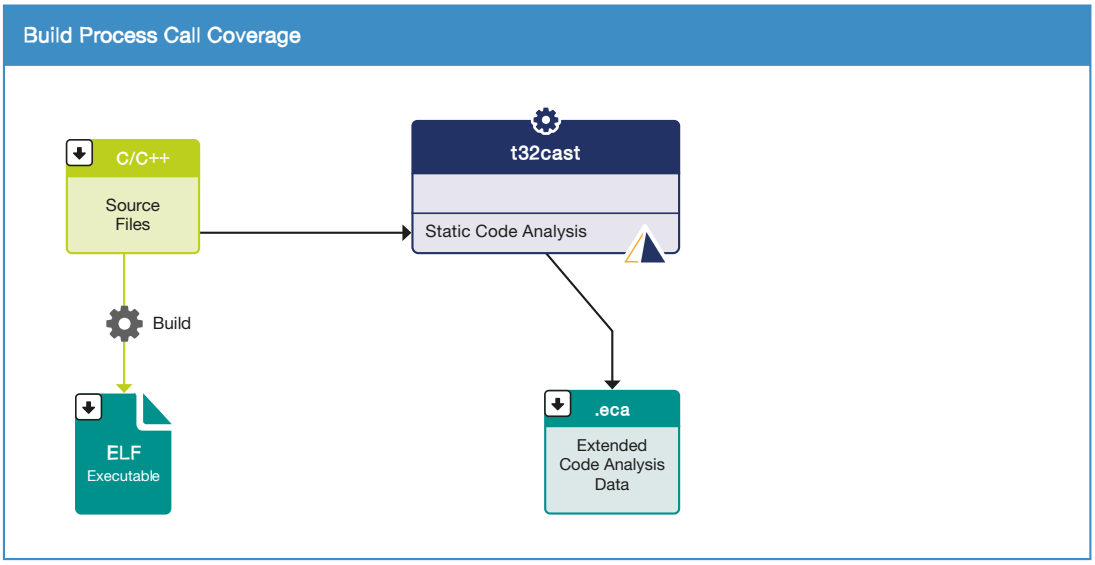
For call coverage you must deactivate the inlining of functions and reduce the optimization.

To measure call coverage TRACE32 needs to identify the locations of function calls. Since this information is not contained in the debug information generated by the compiler, Lauterbach offers its own Clang-based command line tool called t32cast for this purpose. t32cast analyzes the C/C++ sources and generates an extended code analysis file (.eca) for each source file, which contains the required location information. To generate these files, t32cast offers the following command:

```
t32cast eca -o <output-file> <input-file>
```

More details can be found in **“Command Line Parameters of t32cast”** in Application Note for t32cast, page 7 (app_t32cast.pdf).

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



Decision Making

As described in **"MC/DC, Condition and Decision Coverage"**, page 15, several code coverage modes are available for these metrics.

Before you adapt the build process for TRACE32 code coverage

- you must decide on the appropriate code coverage mode.
- you can check whether you are able to use a TRACE32 Instruction Set Simulator instead of a debugger with target during the build process.

Decide on the Appropriate Code Coverage Mode

The goal of this step is to select the appropriate mode from the TRACE32 code coverage modes. The number of observability gaps is decisive for this.

The following steps are necessary to determine the number of observability gaps:

1. **Build the executable.**

   Please pay attention to **"Recommendations for the Build Toolchain"**, page 30.

2. **Use t32cast to generate the ECA files for all C/C++ files.**

   The .eca files contain the conditions/decision details that are necessary for the detection of the observability gaps. To create an ECA file with t32cast, please use the command:

   ```
   t32cast eca -o <eca-file> <c-file>
   ```

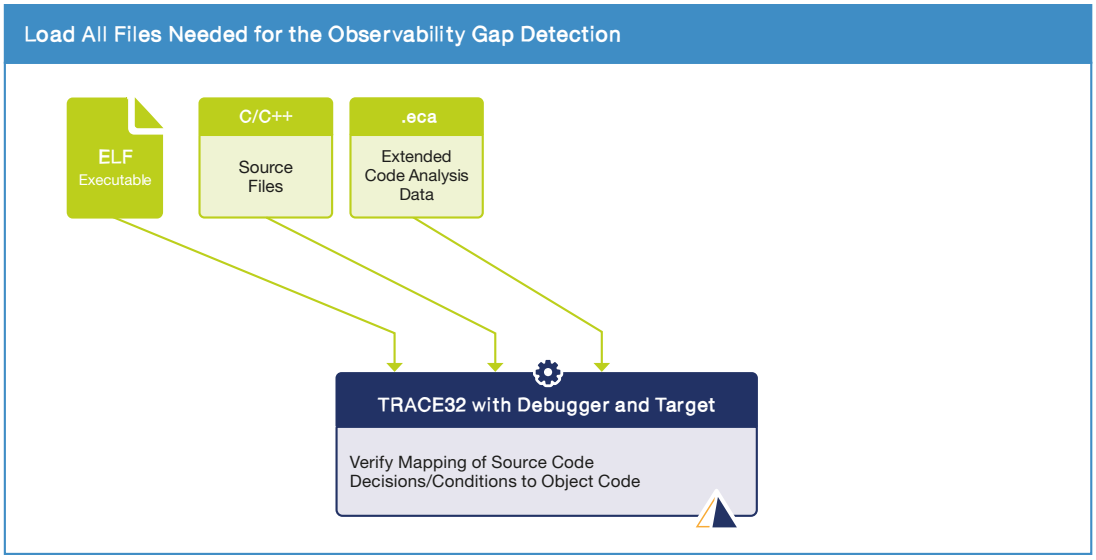
   More details can be found in **"Command Line Parameters of t32cast"** in Application Note for t32cast, page 7 (app_t32cast.pdf).

3. 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
```

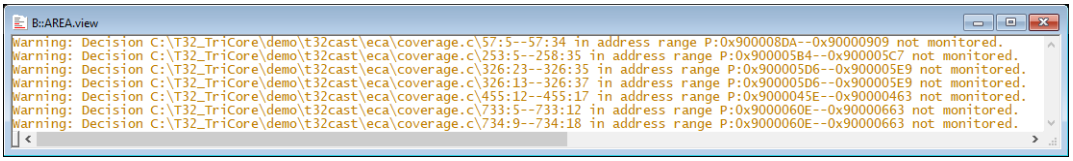
4. Perform the observability gaps detection.

Configure and perform the mapping of the decisions/conditions to the object code. An observabiltiy gap is detected, if a decision/condition is not mapped to a conditional branch/instruction.

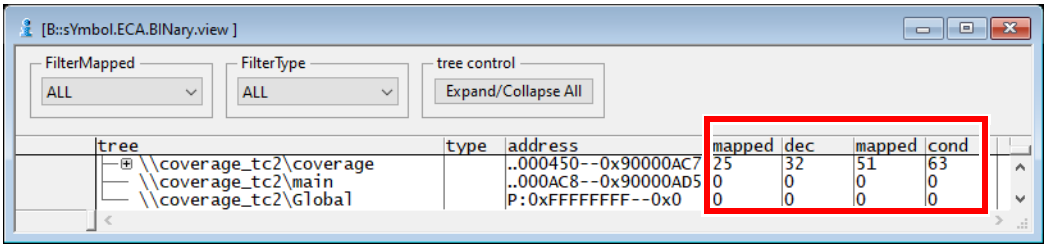
```
; clear message AREA
AREA.CLEAR

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

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

There are two ways to inspect the observabiltiy gaps:

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




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



The following function returns the number of recognized observabiltiy gaps.

sYmbol.ECA.BINary.GAPNUMBER()

The result can be no, few or many observabiltiy gaps. Please be aware that fewer enabled optimization switches should result in a lower amount of observability gaps.

Depending on the result, you have to choose your code coverage mode. Decision-making aid can be found in **"MC/DC, Condition and Decision Coverage"**, page 15.

Decide on the Use of TRACE32 Instruction Set Simulator

In many cases it will be possible to use a TRACE32 Instruction Set Simulator (ISS) instead of the TRACE32 debugger with target during the build process. The advantage would be that you do not have to allocate a debugger/target configuration for the build process. In addition, no license is required to use the ISS in this usage scenario.

The prerequisite for this is that the ISS detects the same observabiltiy gaps as the debugger/target configuration. You should check this before you make this decision.

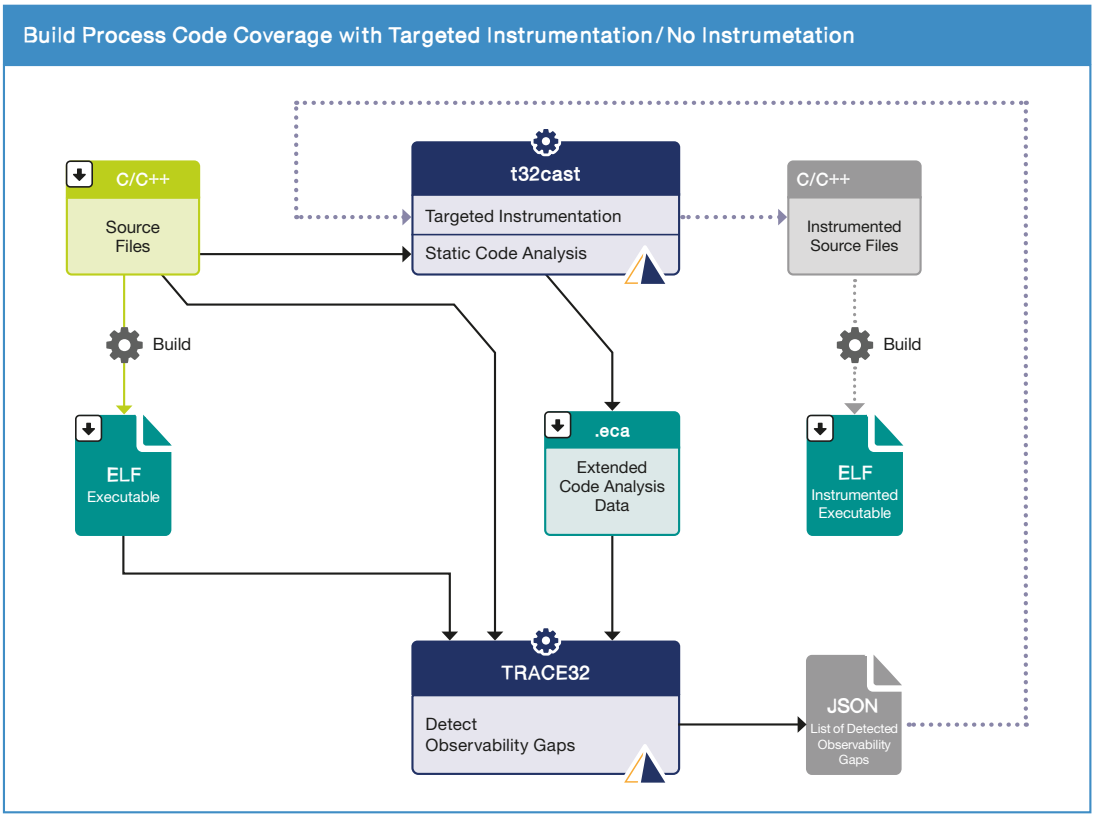
Use the command **sYmbol.ECA.BINary.EXPORT.GAPS** to export the observabiltiy gaps to a JSON file.

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

Then perform the same test with a TRACE32 Instruction Set Simulator and export the detected observabiltiy 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.



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

- 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.

All input/outputs of the build process that might be required for code coverage are marked in figure “Build Process Code Coverage with Targeted Instrumentation/No Instrumentation” with an arrow pointing downwards.

If you want to perform code coverage with targeted instrumentation/no instrumentation, you need to extend your build process as follows:

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

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

   ```
   t32cast eca --export-cfg -o <eca-file> <c-file>
   ```

   More details can be found in **“Command Line Parameters of t32cast”** in Application Note for t32cast, page 7 (app_t32cast.pdf).

2. Add TRACE32 to perform the observability gap check.

TRACE32 can be called from the Make file with a script that performs the check automatically. Please refer to **"Command Line Arguments for Starting TRACE32"** in TRACE32 Installation Guide, page 54 (installation.pdf) for details. This could look like the following:

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

You should have checked in step **"Decide on the Use of TRACE32 Instruction Set Simulator"**, page 36, if you can use a TRACE32 Instruction Set Simulator instead of a debugger/target configuration.

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

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

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

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

; delete JSON file, if existing
IF FILE.EXIST(gaps.json)
(
  RM gaps.json
)

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

; perform mapping
sYmbol.ECA.BINary.PROCESS

; export observabiltiy gaps to JSON file
IF sYmbol.ECA.BINary.GAPNUMBER()>0.
(
      sYmbol.ECA.BINary.EXPORT.GAPS gaps.json
)
```

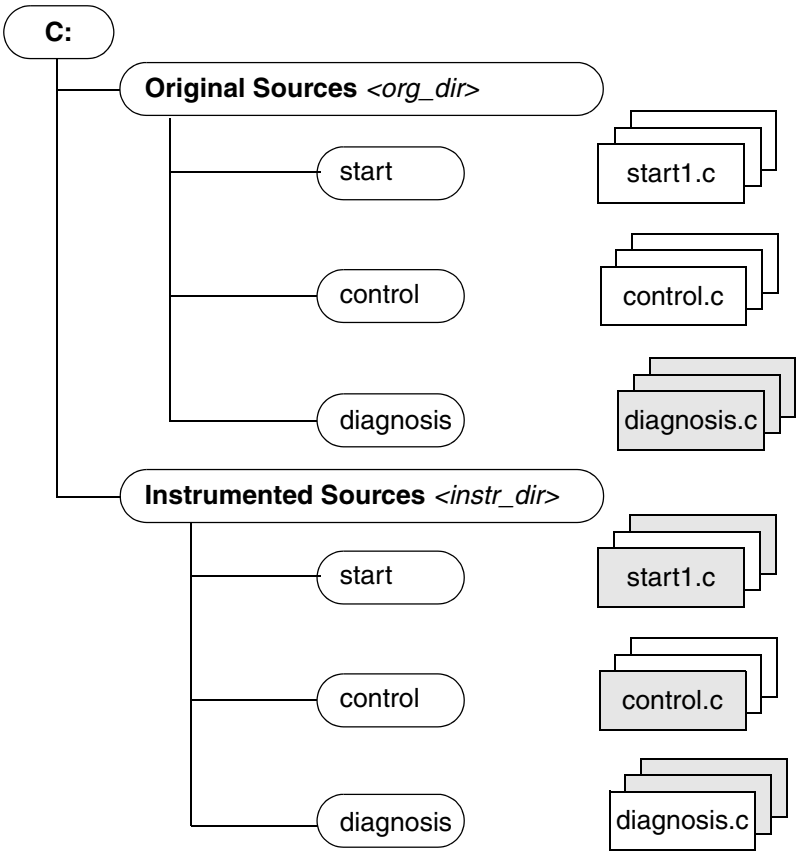
TRACE32 only generates a fresh JSON file if gaps are detected in the observation.

An existing JSON file is deleted here, as TRACE32 decides in this manual on the basis of the existence of the JSON files whether it must load the instrumented or the non-instrumented executable. However, this is only one possible approach.

3.A If TRACE32 has not generated a JSON file, build the non-instrumented executable.

3.B If TRACE32 has generated a JSON file, use t32cast for targeted instrumentation..

The result of this step should be a structure of directories (Instrumented Sources in the figure below) with the following content:



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

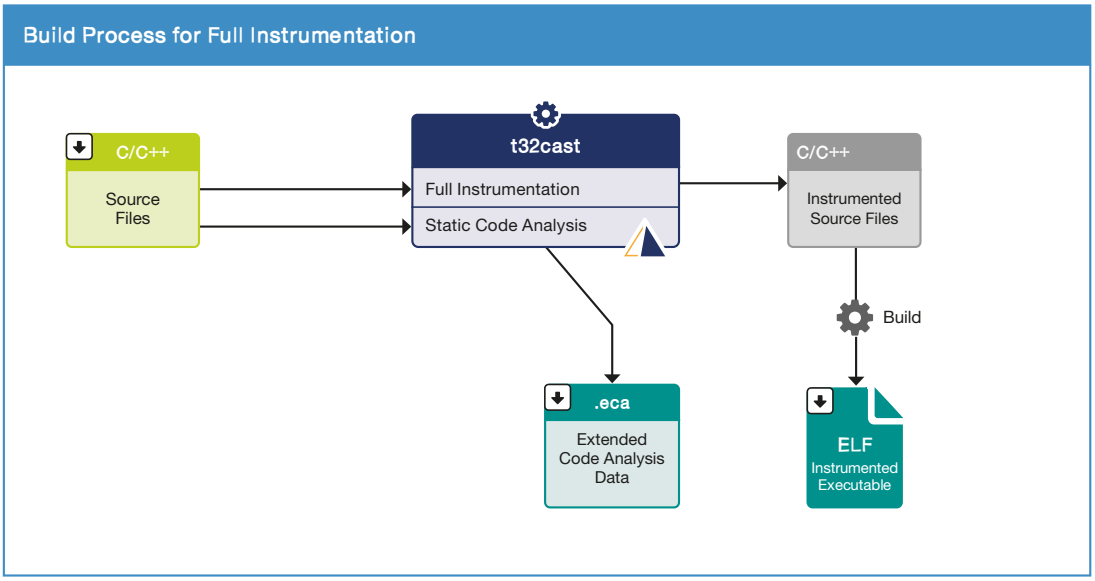
- For each source file that does not contain observabiltiy gaps, there is a copy of the original in the Instrumented Sources directory (white rectangles for not-instrumented source files in the figure above).

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=<instr_dir>
; 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 file with observabiltiy gaps as input for targeted
; instrumentation and instrument all decisions for which
; a observabiltiy gap was detected

; source files without observabiltiy gaps are simply
; copied to <instr_dir>
t32cast instrument --mode=mcdc --filter=gaps.json
-o <instr_dir\file> <org_dir\file>
```

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

4.B Build the instrumented executable.



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

All input/outputs of the build process that are required for code coverage are marked in figure “Build Process Code Coverage with Full Instrumentation” with an arrow pointing downwards.

If you want to perform code coverage with full instrumentation, 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 --export-cfg -o <eca-file> <c-file>
```

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. The result of this step should be a structure of directories that contains all instrumented source files.

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

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

Finally, an instrumented executable must then be generated.

TRACE32 Tool Configurations

The following TRACE32 tools are suitable for code coverage:

- **TRACE32 Debugger and Off-Chip Trace**
- **TRACE32 Debugger and On-Chip Trace**
- **TRACE32 Instruction Set Simulator**

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

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

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

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

TRACE32 Debuggers for virtual targets should, because of their limitations, only be used for code coverage if needed. For details refer to **"Code Coverage with Virtual Targets"**, page 76.

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

Choose the Appropriate Trace Data Collection Variant

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

If you are using a TRACE32 Advanced Register Trace (**Trace.METHOD ART**), please refer to **"ART Mode Code Coverage"**, page 78.

Collection Method	Incremental (fallback)	Incremental with Streaming	SPY	RTS
Description	Trace data is first recorded and then analyzed. Code coverage requires repeated test runs. The size of the trace memory limits the amount of data that can be recorded in a single test run.	Trace data is first recorded and then analyzed. Code coverage requires repeated test runs. Since trace data is streamed to the host computer at recording time, a larger amount of data can be recorded in each test run.	Trace data is recorded and analyzed on a timely basis. Code coverage results are rapidly visible.	Trace data is recorded and directly analyzed. Code coverage results are immediately visible.
Supported Recorder	TRACE32 Instruction Set Simulator Onchip trace Virtual targets PowerTrace µTrace or CombiProbe for Cortex-M	PowerTrace µTrace and CombiProbe for Cortex-M		PowerTrace µTrace and CombiProbe for Cortex-M
Supported Trace Protocols	all	all	all	ETM v3, PTM, ETM v4 for Arm/Cortex MCDS for Infineon Tricore Nexus for MPC5xxx/STM SPC5xx Nexus for PPC QorIQ
Supported Coverage Metrics	all	all	all	Object code coverage Statement coverage ocb decision coverage Call coverage Function coverage
Restrictions	none	Not suitable for high-bandwidth trace ports	Not suitable for high-bandwidth trace ports. Only restrictively suitable if a rich OS is used.	

Reduce the Amount of Trace Data

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

It is recommended to configure the onchip trace logic:

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

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

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

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

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

Ensure a Fault-Free Trace Recording

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

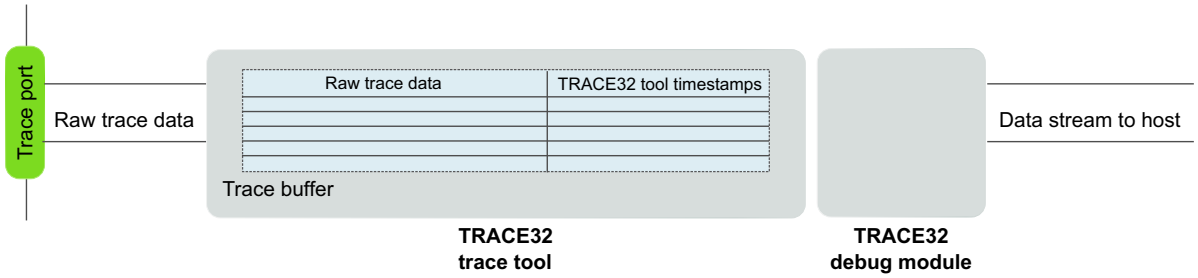
```
Go
Break
SILENT.Trace.Find FLOWERROR /ALL
IF FOUND.COUNT()!=0.
(
  PRIVATE &msg
  &msg="FLOWERRORS were found in the analyzed trace recording."
  &msg="&msg It is recommended to check"
  &msg="&msg if the trace recording works properly."
  ECHO FOUND.COUNT() "&msg"
)
ELSE
(
  ECHO "The analyzed trace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

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

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

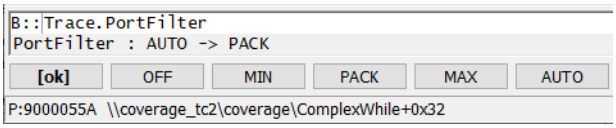
Disable Timestamps for Trace Streaming

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



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

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



SMP Multicore Systems

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

Notes on the Individual Test Variants

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

Incremental code coverage (one test run with repeated cycles)

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

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

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

Incremental code coverage (two separate test runs)

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

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

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

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

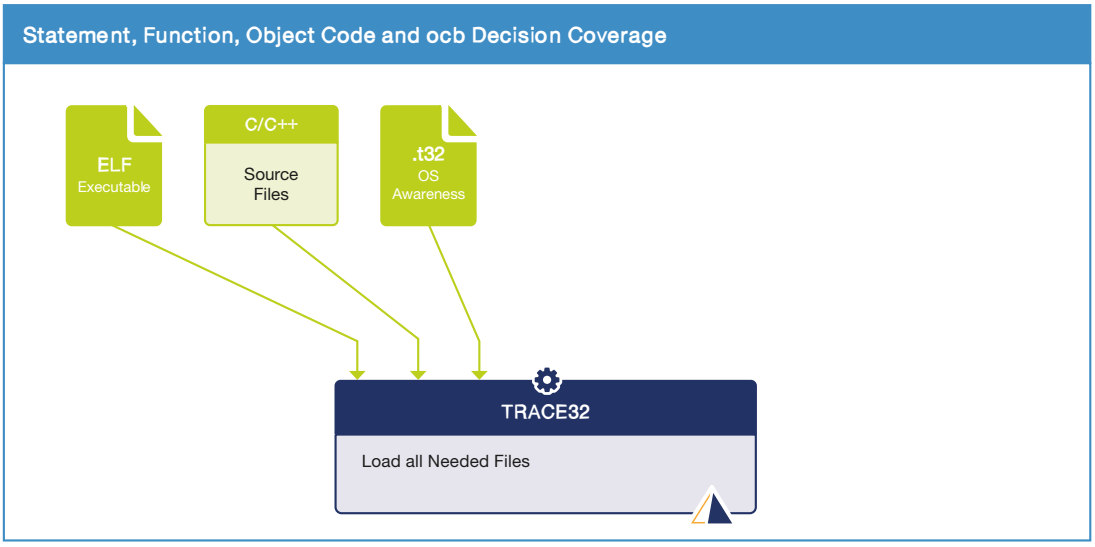
   The code coverage team must always load all files.

Live code coverage (RTS, SPY)

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

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

All trace data collection variants can be used here.



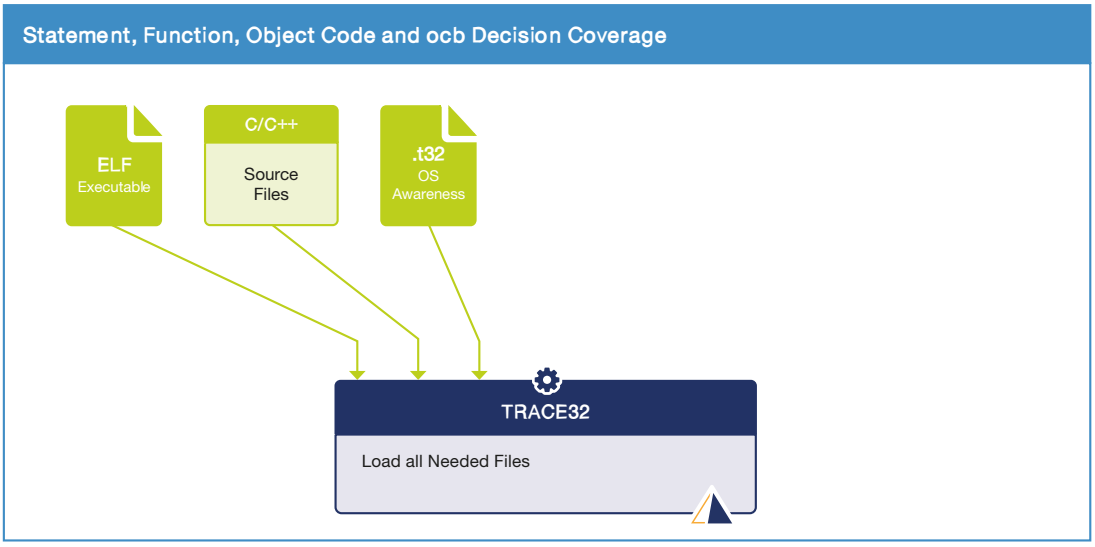
The following files need to be loaded into TRACE32:

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

The following commands can be used for this purpose:

```
; basic debug and trace setup

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

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

; load the OS Awareness
TASK.CONFIG myos.t32

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

All trace data collection variants can be used here.



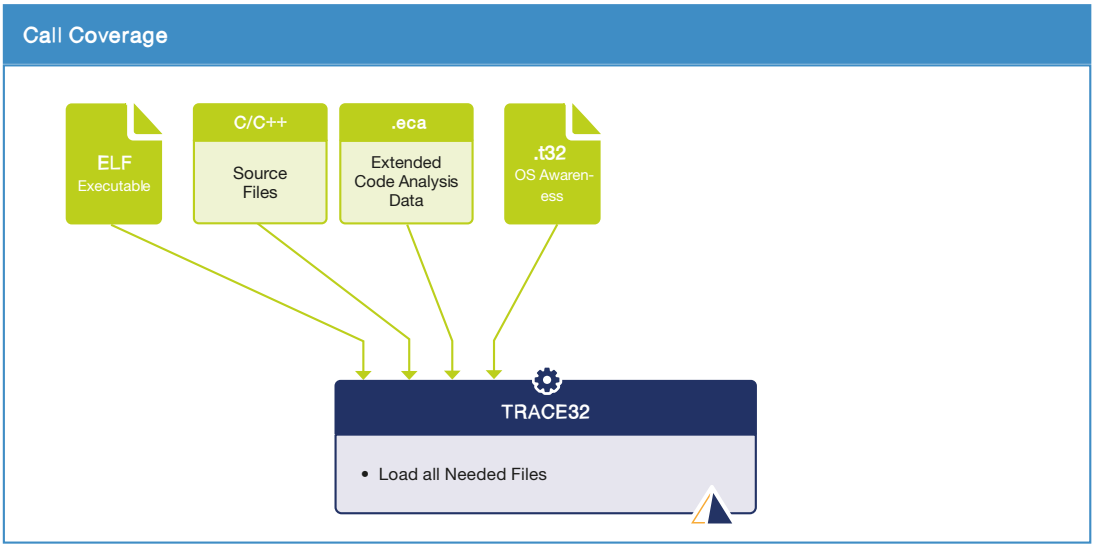
The following files need to be loaded into TRACE32:

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

The following commands can be used for this purpose:

```
; basic debug and trace setup

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

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

; load the OS Awareness
TASK.CONFIG myos.t32

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

All trace data collection variants can be used here.



The following files need to be loaded into TRACE32:

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

The following commands can be used for this purpose:

```
; basic debug and trace setup

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

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

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

; load the OS Awareness
TASK.CONFIG myos.t32

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

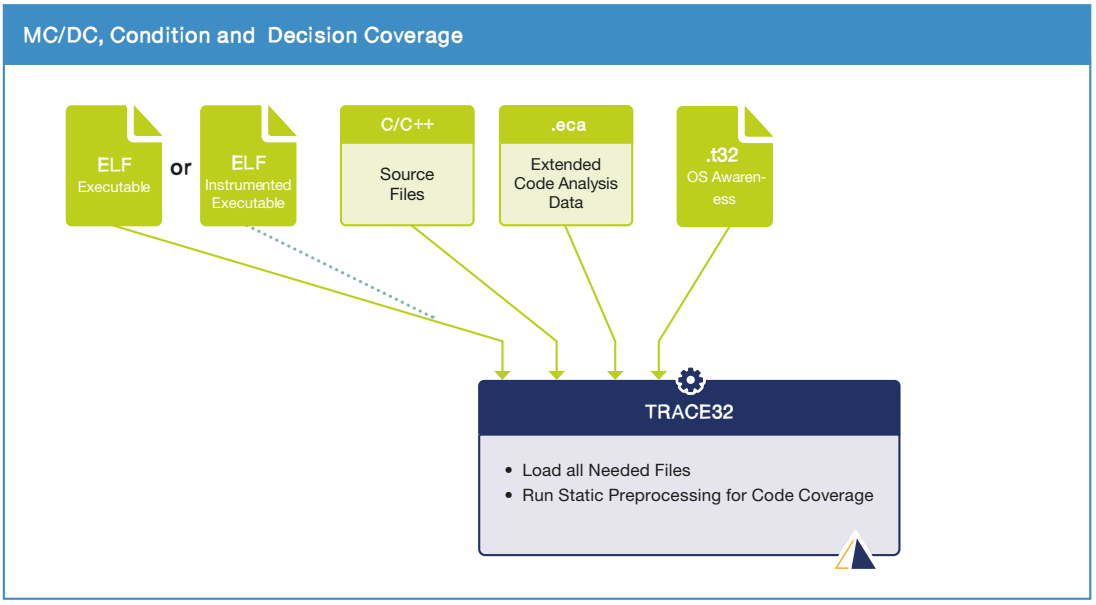
Preparation for MC/DC, Condition and Decision Coverage

All trace data acquisition modes can be used here, but not RTS mode.

The preparation is different for the individual code coverage modes:

- Targeted Instrumentation/No Instrumentation
- Full Instrumentation

Preparation for Targeted Instrumentation/No Instrumentation



The following files need to be loaded into TRACE32:

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

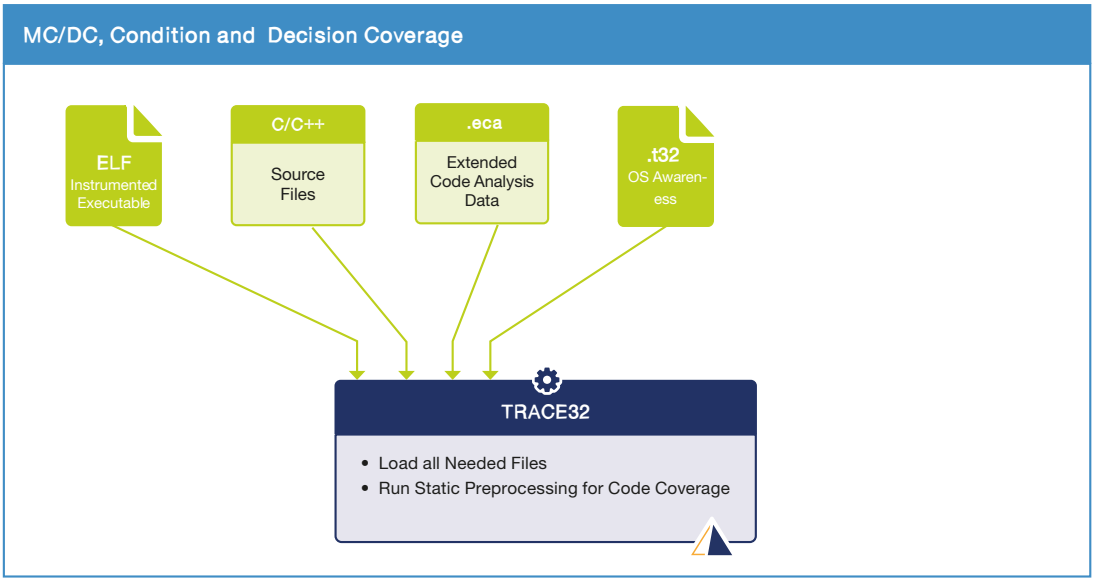
> **NOTE:** Please note that TRACE32 performs the code coverage analysis for the instrumented executable with the original, non-instrumented source code files.
>
> For this reason, the paths to the source code files included in the instrumented executable file must always be adapted accordingly. The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

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

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

The following framework can be used for this purpose:

```
; basic debug and trace setup

; load appropriate executable

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

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

; load the OS Awareness
TASK.CONFIG myos.t32

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

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

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

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

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

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



The following files need to be loaded into TRACE32:

- Instrumented executable

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

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

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

The following framework can be used for this purpose:

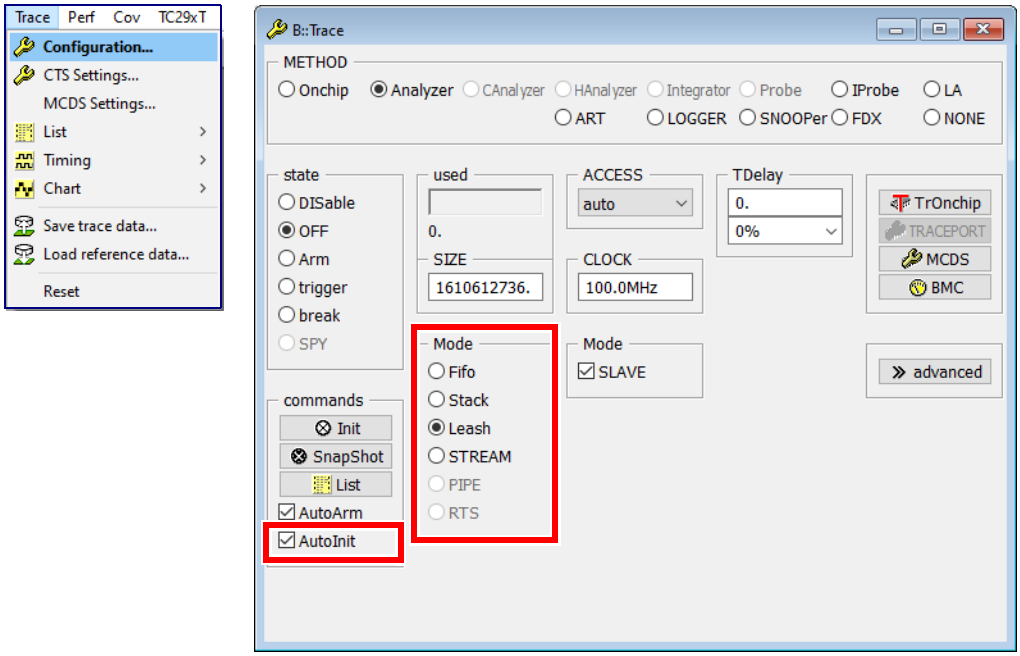
```
; basic debug and trace setup

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

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

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

; load the OS Awareness
TASK.CONFIG myos.t32

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

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

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

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

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

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

Incremental Code Coverage

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

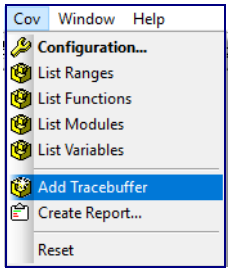
Data Collection

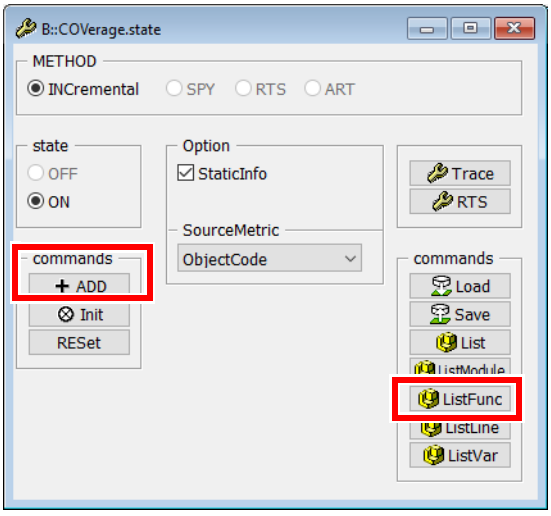
1. Set the trace to Leash Mode either via the **Trace configuration** window or via the command **Trace.Mode Leash**. This ensures that the target will halt when the trace buffer becomes nearly full, preventing loss of data. Stack or Fifo mode can also be used if Leash Mode is not supported.
2. Enable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



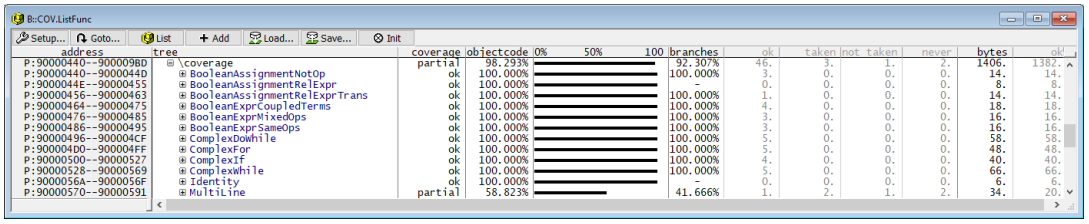
3. Start program execution and wait until it stops.

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





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



Details on the code coverage analysis itself are provided in the chapter **"Code Coverage Analysis"**, page 81.

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

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

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **"TRACE32 Merge and Report Tool"**, page 129.

Example Script

The entire process can be automated by creating a PRACTICE script. It is assumed that the preconditions listed in **"Preconditions"**, page 46 are satisfied before running the script. In the example script default settings are commented out.

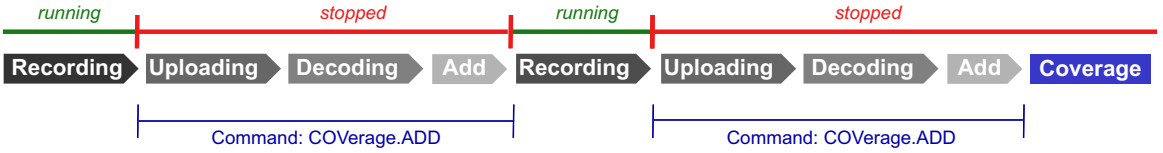
```
...
// Trace.METHOD as automatically selected by TRACE32
Trace.Mode Leash
// Trace.AutoArm ON
Trace.AutoInit ON
COVerage.RESet
// COVerage.METHOD INCremental
RePeaT 10.
(
    Go.direct
    WAIT !STATE.RUN()
    COVerage.ADD
)
COVerage.ListFunc

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

Summary

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

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

This workflow is summarized in the diagram below.



Details about the code coverage analysis itself are provided in the chapter **"Code Coverage Analysis"**, page 81.

Incremental Code Coverage in STREAM Mode

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

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

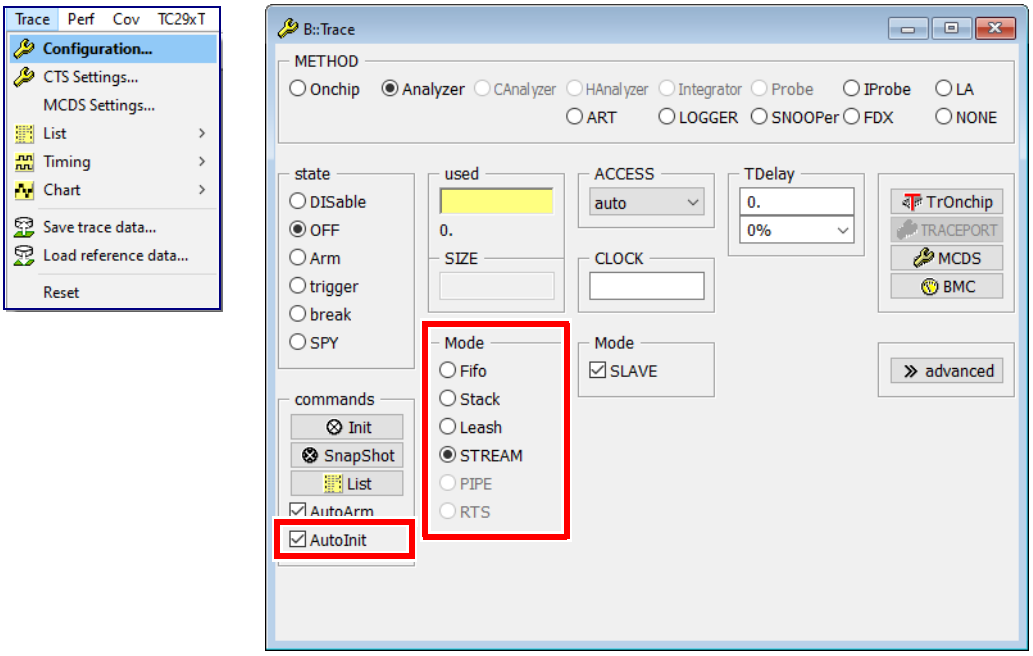
In case of large amounts of trace data, processing may take a long time. TRACE32 provides two alternative methods to avoid this situation.

The first method is RTS, which is supported for all major architectures. RTS means that trace data is processed while being recorded and the code coverage results are displayed dynamically. Please see **"RTS Mode Code Coverage"**, page 64 for additional information.

If RTS is not supported for your core architectures, then SPY Mode Code Coverage can be an alternative. Please see **"SPY Mode Code Coverage"**, page 70 for more details.

Data Collection

1. Set the trace to STREAM Mode either via the **Trace Configuration** window or via the **Trace.Mode STREAM** command.
2. Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to ensure that the trace buffer is always cleared before the trace recording is started.



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

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

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

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

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

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

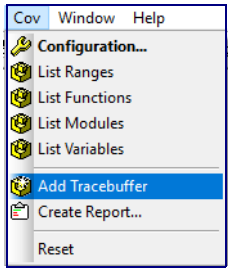
5. Since code coverage does not need any timestamp information, please use the command **Trace.PortFilter MAX** to instruct TRACE32 to stream only the raw trace data. Further background information can be found in the chapter **“Disable Timestamps for Trace Streaming”**, page 48.

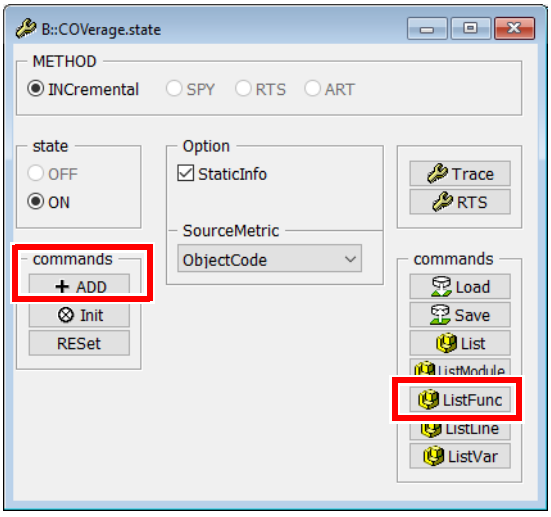
6. Start the program execution.

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

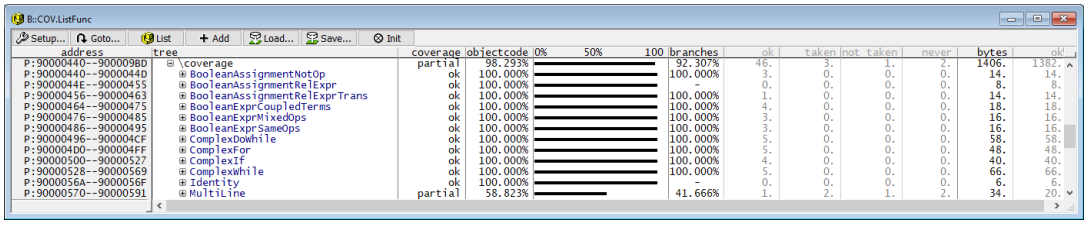
   - The user may manually stop the program execution.
   - A breakpoint may be used to stop the program execution.
   - With the help of a script, the program execution may be stopped after a specific period of time.

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





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



Details on the code coverage analysis itself are provided in the chapter **"Code Coverage Analysis"**, page 81.

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

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

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **"TRACE32 Merge and Report Tool"**, page 129.

Example Script

In this example script default settings are commented out. It is assumed that the preconditions listed in **"Preconditions"**, page 46 are satisfied before running the script.

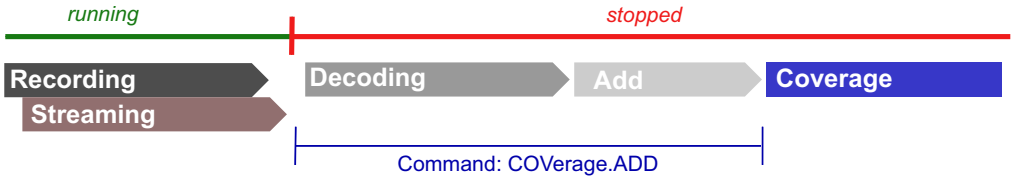
```
...
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer
// Trace.AutoArm ON
Trace.AutoInit ON

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

Trace.PortFilter MAX

COVerage.RESet
// COVerage.METHOD INCremental

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

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

Summary

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

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

This workflow is summarized in the diagram below.



Details about the code coverage analysis itself are provided in the chapter **"Code Coverage Analysis"**, page 81.

RTS Mode Code Coverage

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

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

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

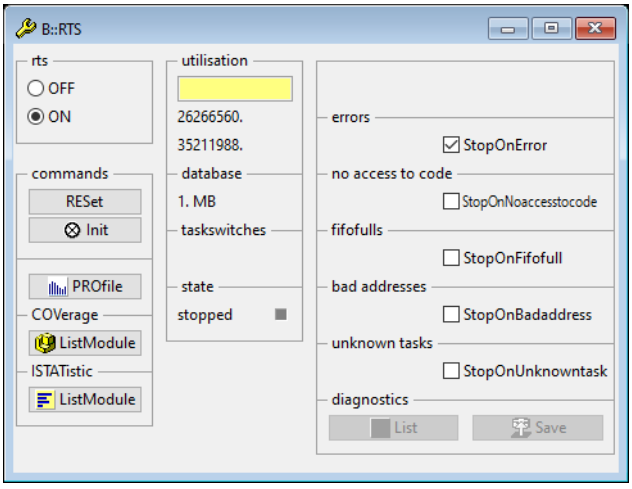
If RTS is not supported for your core architectures, then SPY mode code coverage could be an alternative. Please refer to **"SPY Mode Code Coverage"**, page 70.

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

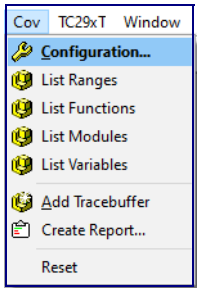
RTS mode code coverage supports only the following code coverage metrics: statement coverage, function coverage, object code coverage and ocb decision coverage.

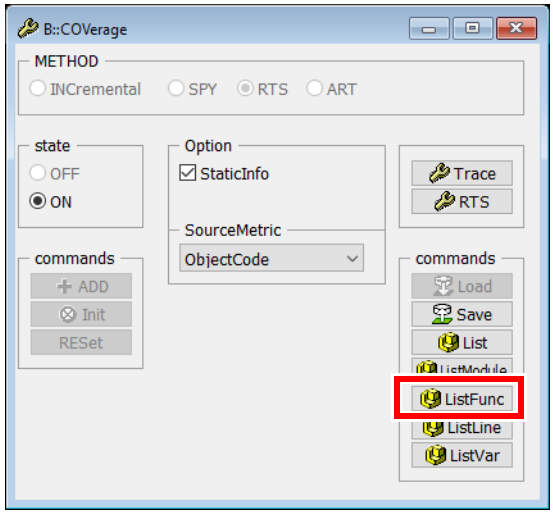
Data Collection

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

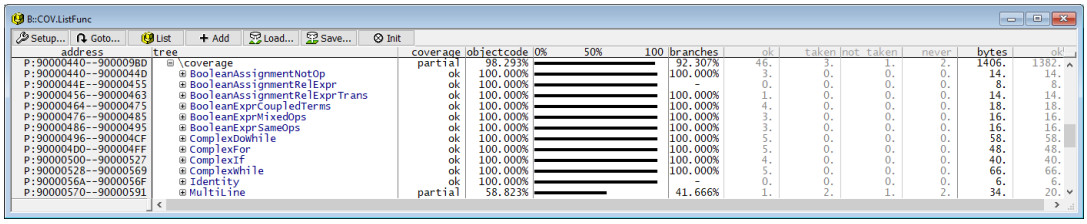


2. Open a **COVerage.ListFunc** window by using the **ListFunc** button in the **COVerage Configuration** window or by using the command **COVerage.ListFunc**. Please be aware that trace data recorded in RTS mode are only processed by TRACE32 as long as one window in TRACE32 displays code coverage information.





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



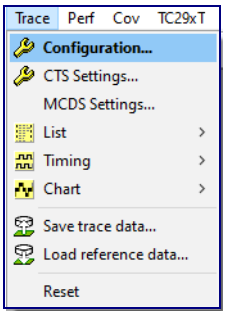
Details on the code coverage analysis itself are provided in the chapter **“Code Coverage Analysis”**, page 81.

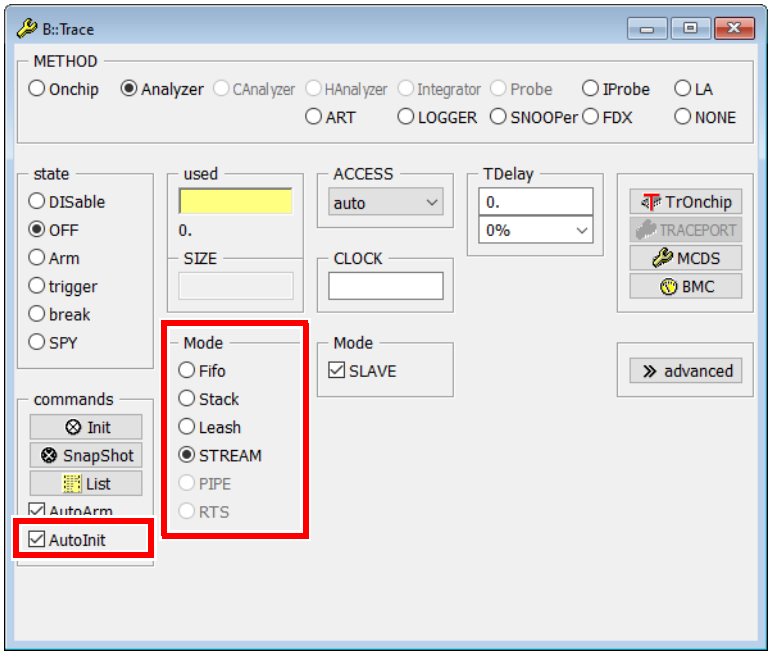
4. Stop the program exucution when your tests are completed.

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

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

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





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

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

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

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

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

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

5. Since code coverage does not need any timestamp information, please use the command **Trace.PortFilter MAX** to instruct TRACE32 to stream only the raw trace data. Further background information can be found in the chapter **"Disable Timestamps for Trace Streaming"**, page 48.

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **"TRACE32 Merge and Report Tool"**, page 129.

Example Scripts

This example script discards the trace data after it is processed; default settings are commented out. It is assumed that the preconditions listed in **"Preconditions"**, page 46 are satisfied before running the script.

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

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

COVerage.RESet
RTS.ON
COVerage.ListFunc

Go
WAIT !STATE.RUN()
...

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

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

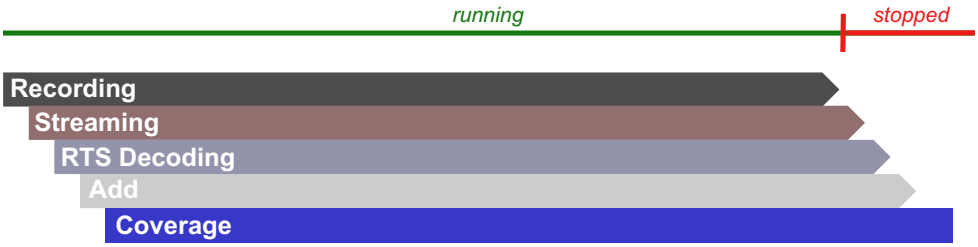
```
...
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer
// Trace.AutoArm ON
Trace.AutoInit ON

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

Trace.PortFilter MAX

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

COVerage.RESet
RTS.ON
COVerage.ListFunc

Go
WAIT !STATE.RUN()
Trace.List

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

Summary

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

The following steps are performed concurrently with trace data collection:

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



Details about the code coverage analysis itself are provided in the chapter **“Code Coverage Analysis”**, page 81.

SPY Mode Code Coverage

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

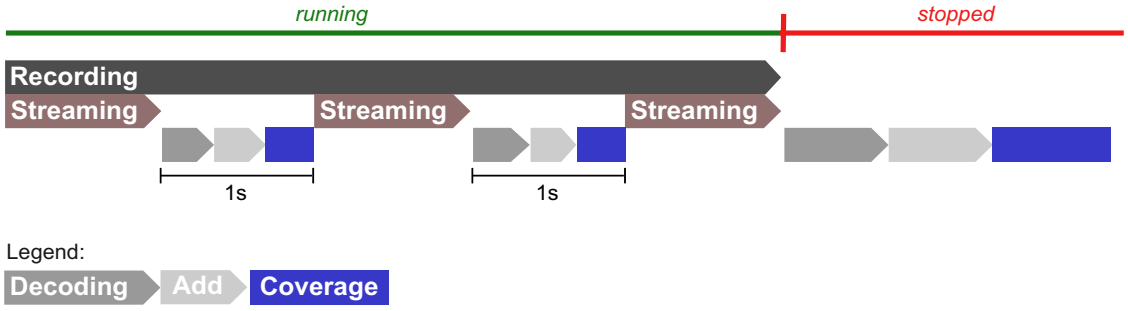
- TRACE32 trace hardware tool such as PowerTrace is required
- Streaming of the trace data to a file on the host file system is working without issues

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

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

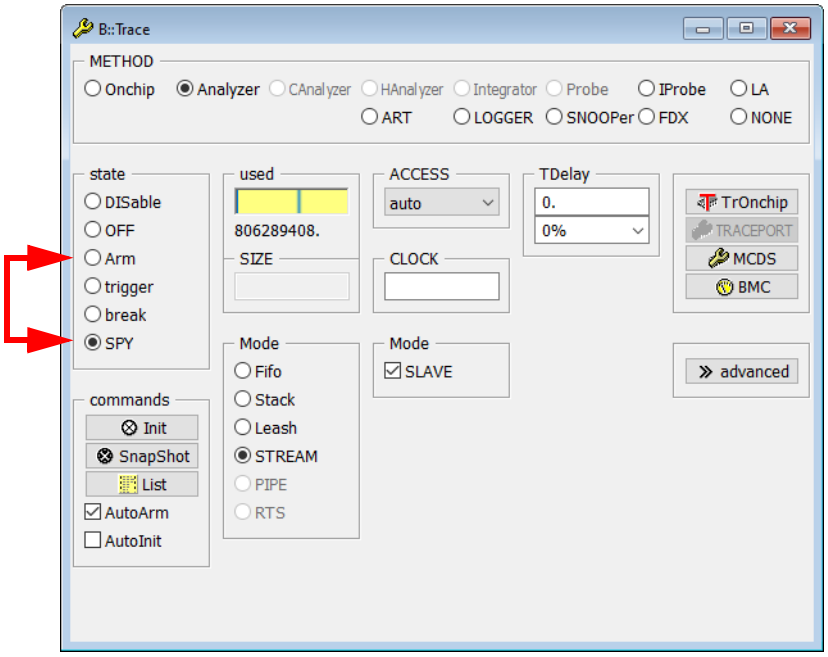
Operation States

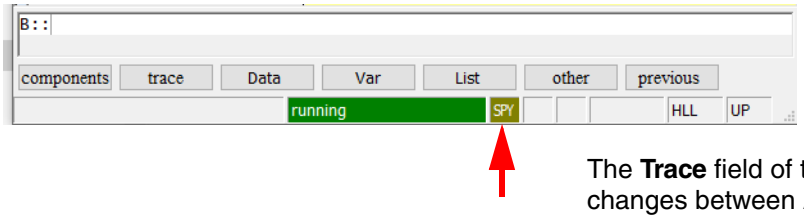
For SPY mode code coverage, trace streaming is periodically suspended in order to decode the raw trace data and to process it for code coverage. Please be aware that TRACE32 does not suspend trace streaming if the trace memory of the TRACE32 trace tool, that operates as a large FIFO, is filled more the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



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

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

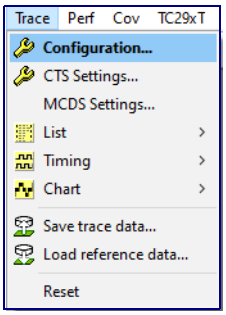


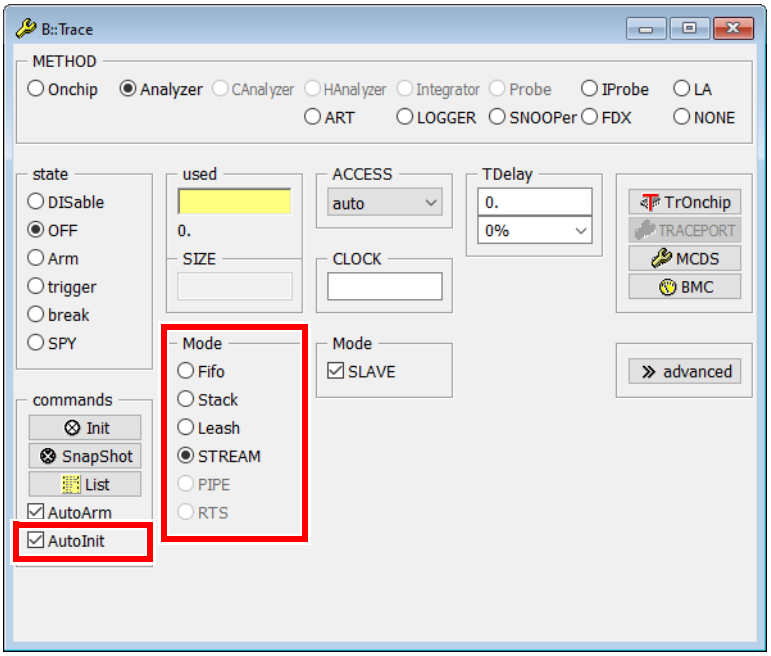


The **Trace** field of the TRACE32 state line changes between Arm and SPY

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

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





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

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

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

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

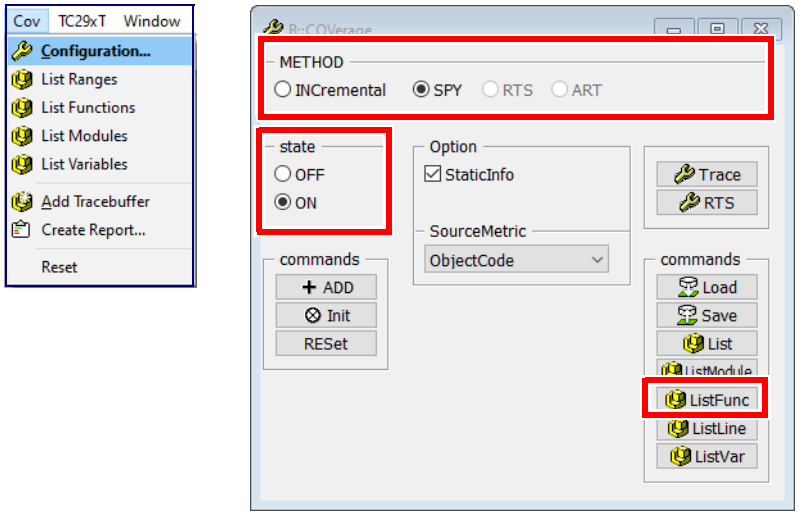
```
; limit the size of the streaming file to 5 GBytes
Trace.STREAMFileLimit 5000000000.
```

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

5. Since code coverage does not need any timestamp information, please use the command **Trace.PortFilter MAX** to instruct TRACE32 to stream only the raw trace data. Further background information can be found in the chapter **"Disable Timestamps for Trace Streaming"**, page 48.

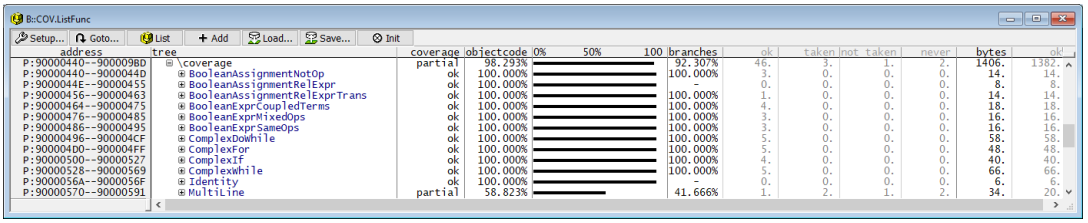
6. Set the coverage method to SPY by using the command **COVerage.METHOD SPY** or by selecting **SPY** in the **COVerage configuration** window.

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



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

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



Details on the code coverage analysis itself are provided in the chapter **"Code Coverage Analysis"**, page 81.

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

You can use the **COVerage.EXPORT.JSONE** command to export the result of the test run. With the Lauterbach command line tool **t32covtool**, you can accumulate coverage data that was collected at different times, with different builds and different target configurations. For details refer to **"TRACE32 Merge and Report Tool"**, page 129.

Example Script

In the script the default settings are commented out. It is assumed that the preconditions listed in **"Preconditions"**, page 46 are satisfied before running the script.

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

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

Trace.PortFilter MAX

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

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

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

Summary

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

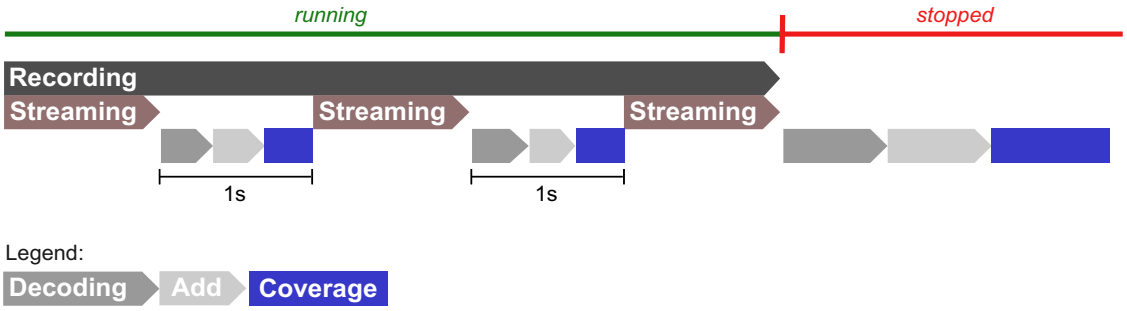
The following steps are involved:

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

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

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

  - to update code **coverage** results



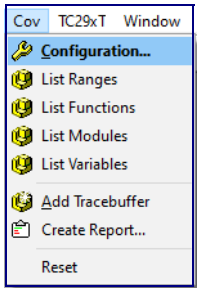
Details about the code coverage analysis itself are provided in the chapter **“Code Coverage Analysis”**, page 81.

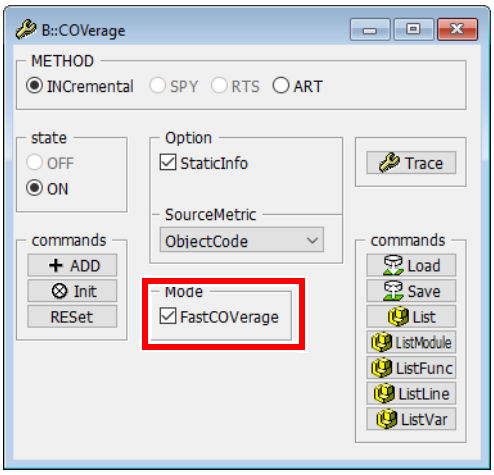
Tracing the program execution on a virtual target slows down its performance. To minimize this impact, Lauterbach works closely together with manufacturers such as Synopsys. The basic idea is that some parts of the code coverage processing are offloaded to the virtual target. This information is uploaded 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 Evaluation”**, page 82)
- Statement coverage (see **“Statement Coverage Evaluation”**, page 87)
- Decision coverage (ocb) (see **“Object Code Based (ocb) Decision Coverage Evaluation”**, page 97)
- Function coverage (see **“Function Coverage Evaluation”**, page 116)

An example script might look like this:

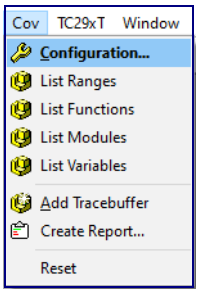
```
COVerage.RESet
COVerage.METHOD INCremental
COVerage.Mode FastCOVerage ON

Go

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

Break

COVerage.Add

COVerage.ListFunc
```

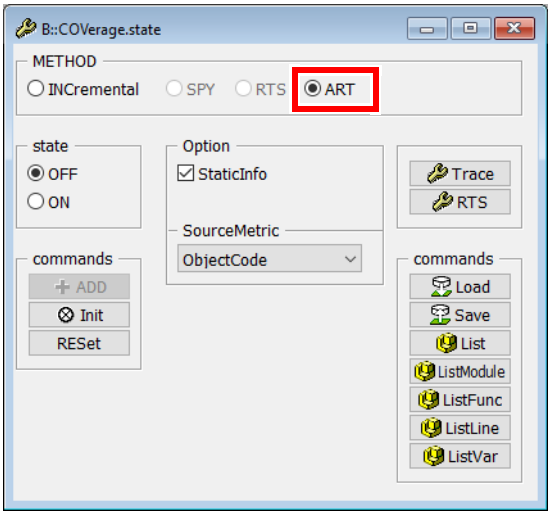
Details about the code coverage analysis itself are provided in the chapter **“Code Coverage Analysis”**, page 81.

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:

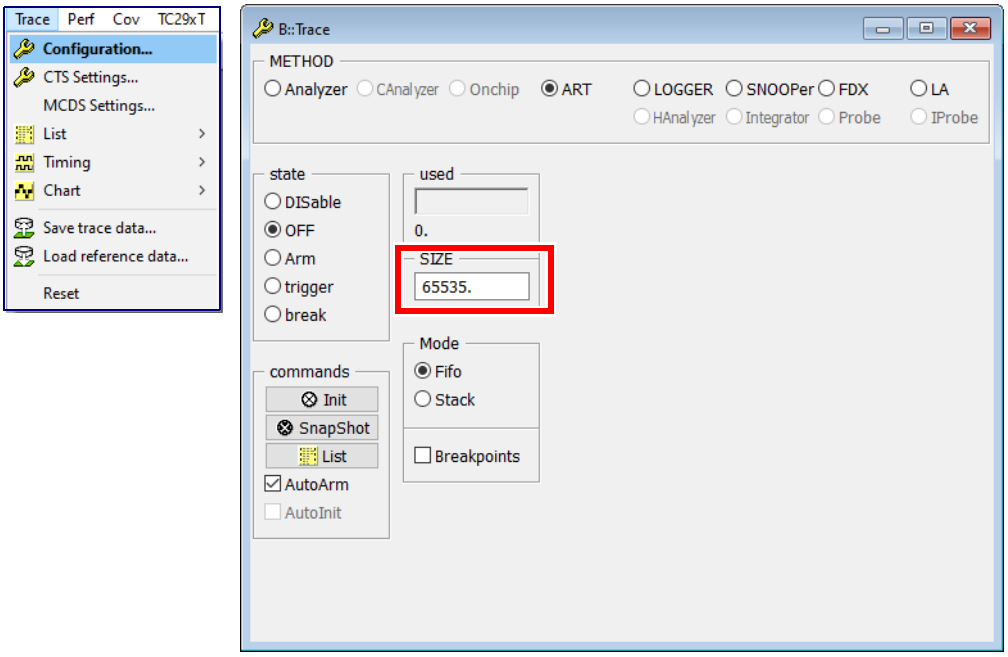
- Object code coverage (see **"Object Code Coverage Evaluation"**, page 82)
- Statement coverage (see **"Statement Coverage Evaluation"**, page 87)
- Decision coverage (ocb) (see **"Object Code Based (ocb) Decision Coverage Evaluation"**, page 97)
- Function coverage (see **"Function Coverage Evaluation"**, 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.

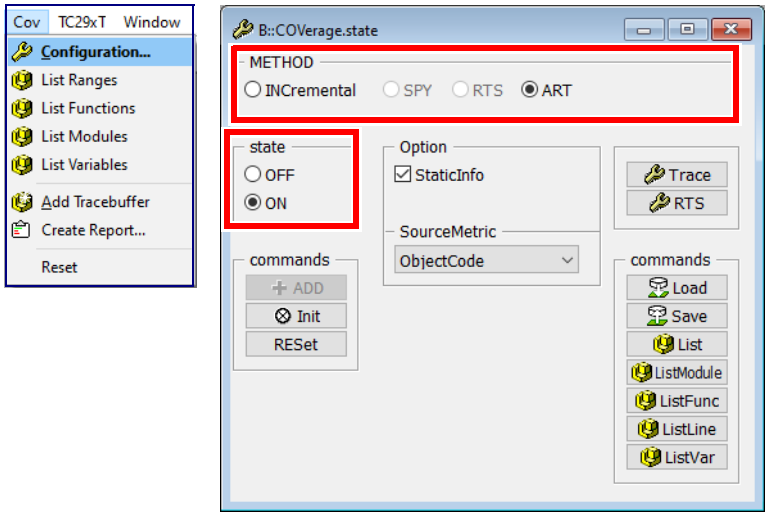
The TRACE32 Instruction Set Simulator simulates the instruction set, but does not model timing characteristics and peripherals. However, the simulator provides a bus trace so that code coverage is easy to perform. For details on how to start the TRACE32 Instruction Set Simulator refer to **"TRACE32 Instruction Set Simulator"** in TRACE32 Installation Guide, page 56 (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.

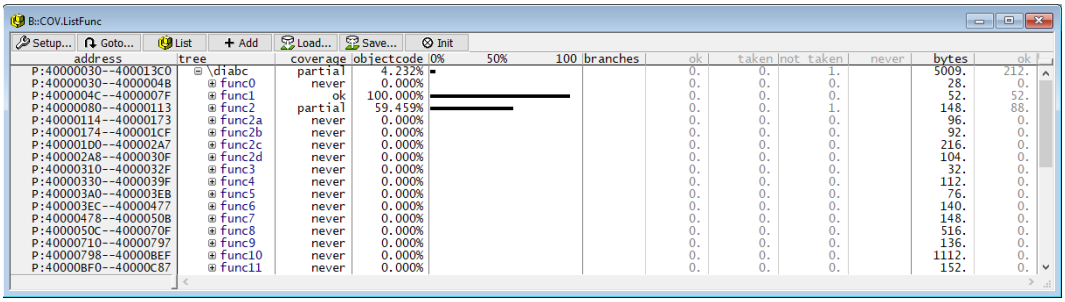
1. Select **Trace.METHOD ART** in the **Trace configuration** window.
2. Set the size of the ART buffer, using either the command **ART.SIZE** *<n>* or by entering the value in the **SIZE** field of the **Trace configuration** window.



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.



Details about the code coverage analysis itself are provided in the chapter **"Code Coverage Analysis"**, page 81.

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 Tags

- **Statement coverage**

  stmt: At least one corresponding object code instruction generated for the source code line has been executed.

  incomplete: None of the object code instructions generated for the source code line has been executed.

- **Decision coverage**

  dc: Decisions have taken all possible outcomes at least once.

  incomplete: There is at least one possible outcome missing for the decision.

- **Condition coverage**

  cc: Conditions have evaluated both, true and false.

  incomplete: Condition have not evaluated both, true and false.

- **MC/DC**

  mcdc: Each condition in decision is shown to independently affect the outcome of that decision.

  incomplete: There is at least one condition in the decision for which has not yet proven to independently affect the outcome of the decision.

- **Function coverage**

  func: At least one function's object code instructions has been executed.

  incomplete: None of the function's object code instructions has been executed.

- **Call coverage**

  call: 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: 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.

Object Code Coverage Evaluation

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**

  For Arm/Cortex cores that use the protocols Arm-ETMv1 or Arm-ETMv3, as well as Arm-ETMv4 with **ETM.COND ON**.

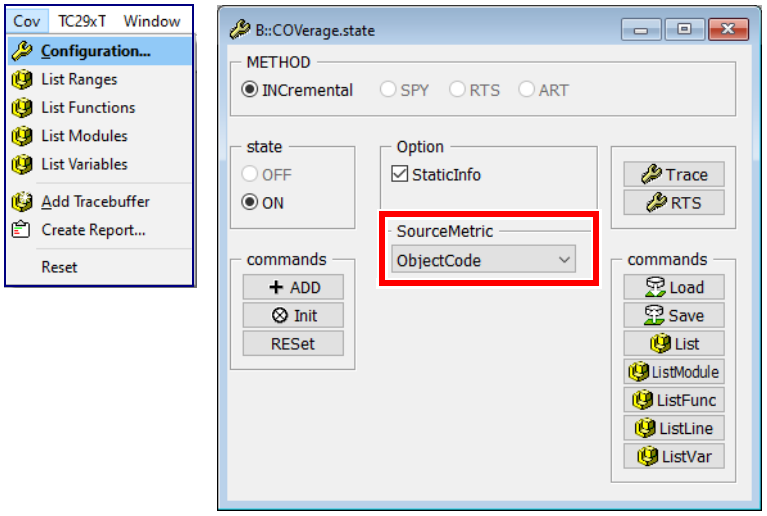
- **ok | taken | not taken | never**

  Otherwise.

For details refer to **"Appendix G: Object Code Coverage Tags in Detail"**, page 151.

Evaluation

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



The following commands show a tabular analysis:

COVerage.ListModule

COVerage.ListFunc

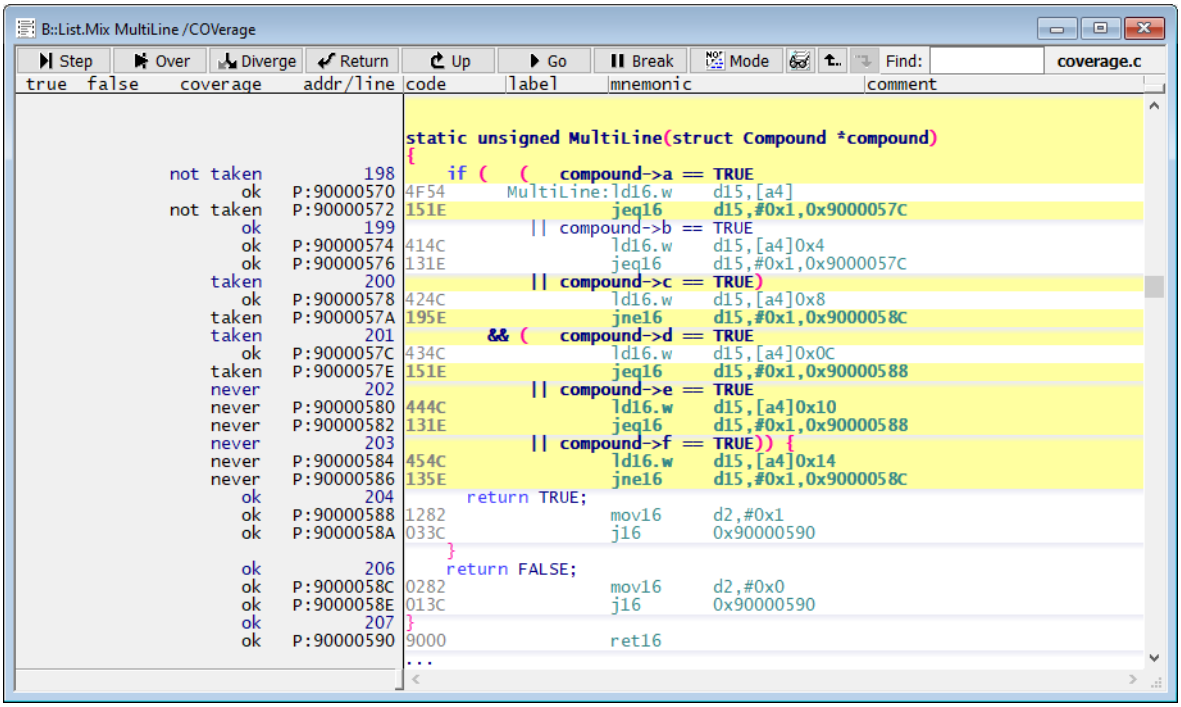
COVerage.ListLine

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

List.Mix /COVerage

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

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

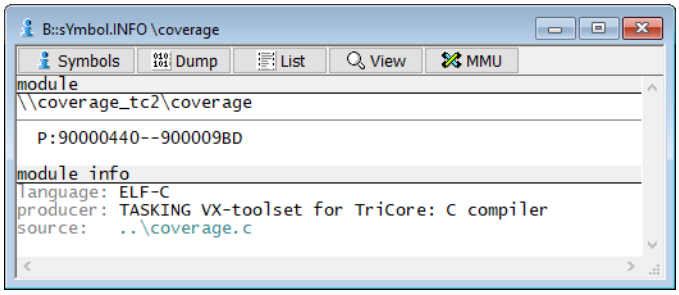


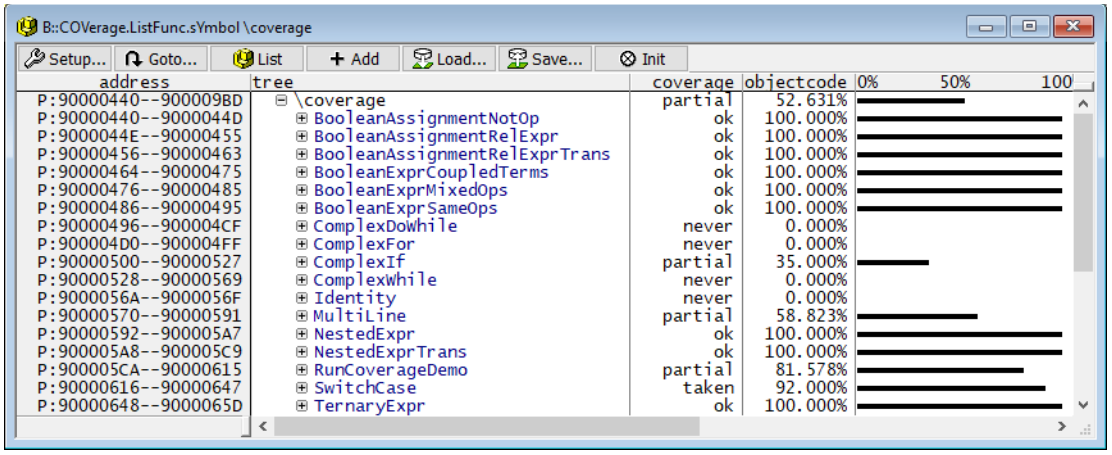
The screenshot on the previous page was taken with the Infineon TriCore™ debugger. Its instruction set contains no conditional instructions beyond conditional branches. Thus the object code is tagged as follows:

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

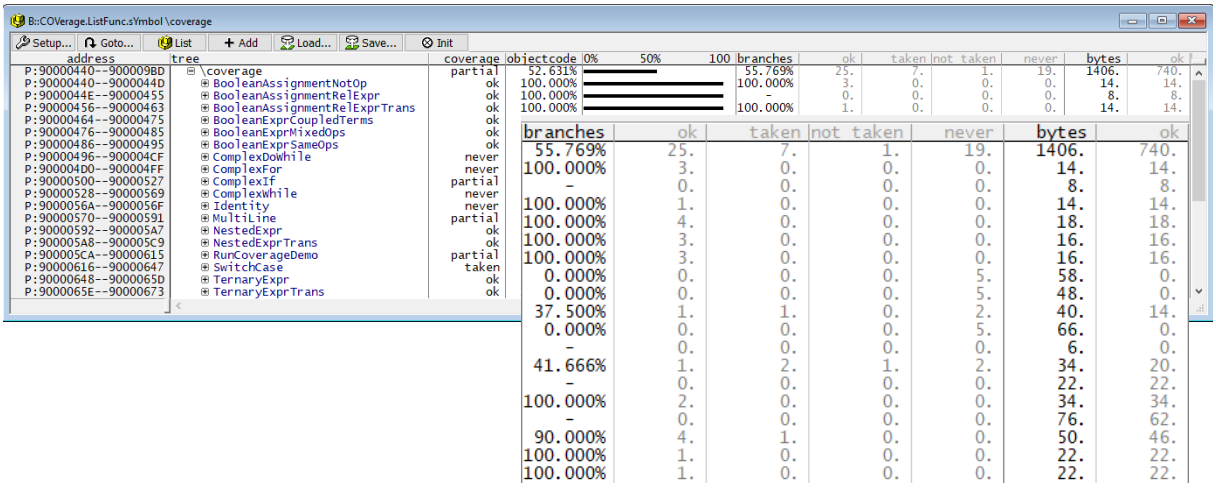
This TRACE32 command displays a tabular analysis of all functions of the module "coverage". A module usually corresponds to a source code file.

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





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/t32cast/eca/measure_mcdc.cmm"

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

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

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

Statement Coverage Evaluation

Statement coverage: Statement coverage ensures that every statement in the program has been invoked at least once. Statement in this context means block of source code lines**.**

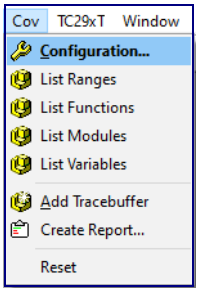
TRACE32 interpretation: A source code line achieves statement coverage when at least one corresponding object code instruction has been executed.

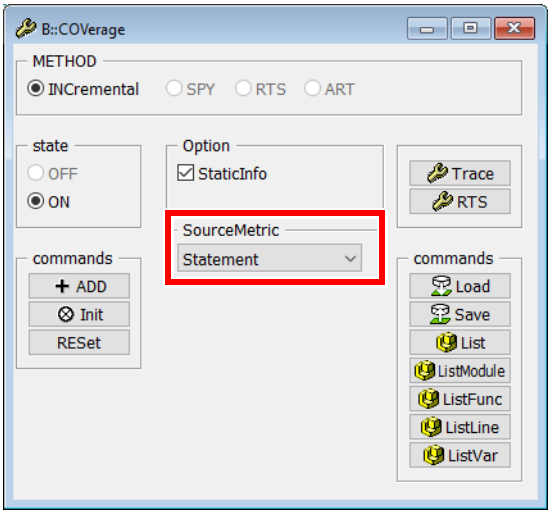
The following tagging is performed:

- **stmt** | **incomplete**

Evaluation

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





The following commands show a tabular analysis:

COVerage.ListModule

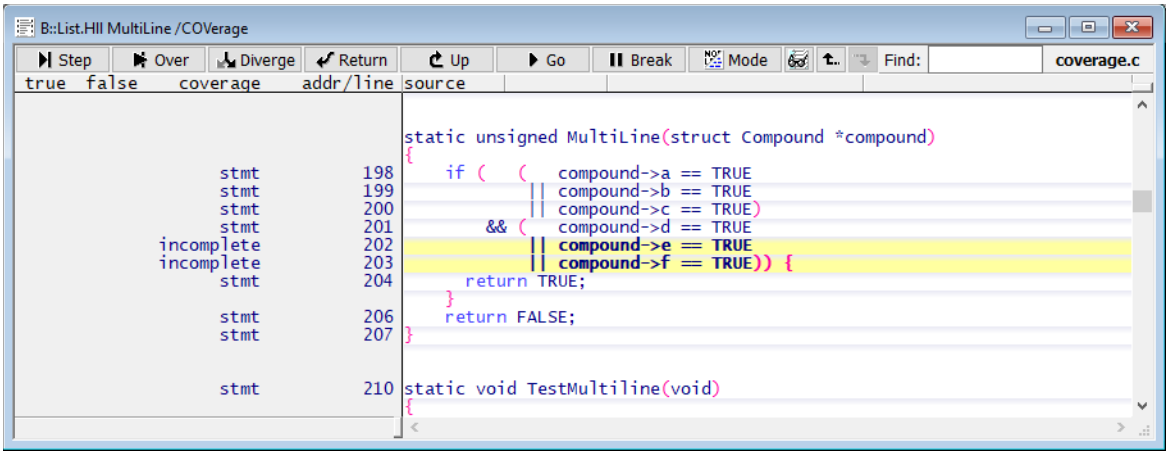
COVerage.ListFunc

The following command shows the tagging on source code level.

List.Hll /COVerage

This TRACE32 command displays the statement coverage tagging for the function *MultiLine*:

```
List.Hll MultiLine /COVerage
```



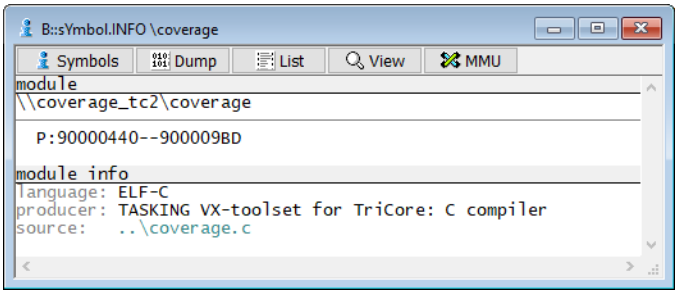
The source code lines are tagged as follows:

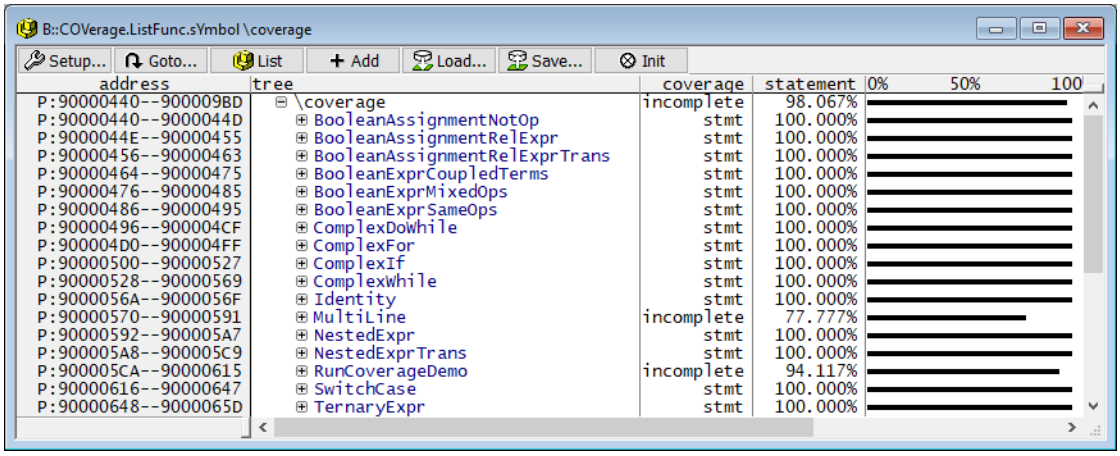
stmt	At least one corresponding object code instruction generated for the block of source code lines has been executed.
incomplete	None of the object code instructions generated for the block of source code lines has been executed.

Object code instructions show the corresponding tags for object code coverage, if statement coverage is selected.

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

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





Tags for Statement Coverage

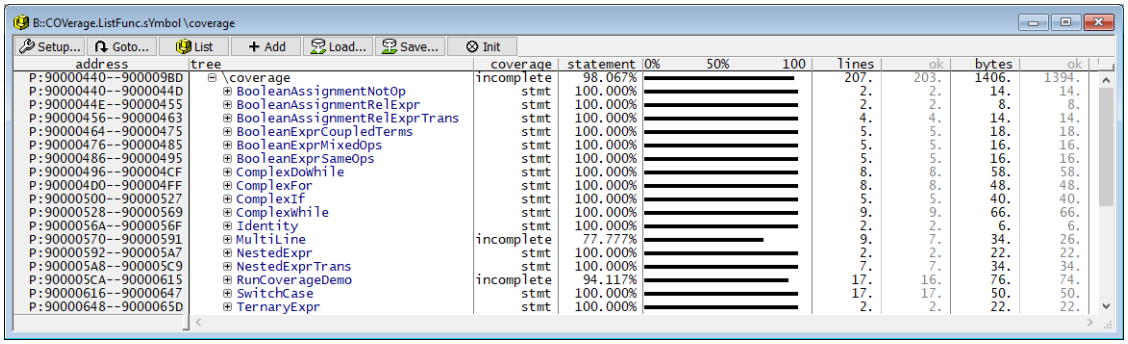
Statement coverage is achieved for a group of **HLL source code statements** as soon as one of its associated assembly instructions has been partially executed.

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

If a tag marks the coverage status of **HLL source code statements**, the following definitions apply:

- **stmt**: The measured code coverage of the HLL source code statement(s) is sufficient to achieve statement coverage.
- **incomplete**: The measured code coverage of the HLL source code statement(s) is not sufficient to achieve statement coverage.

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



Line count	
line	Number of source code line blocks
ok	Number of source code line blocks tagged with stmt

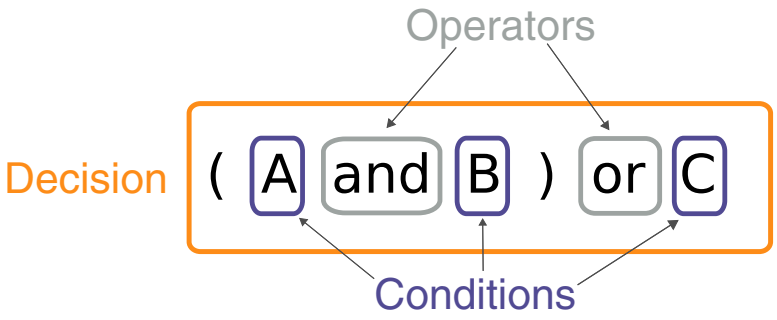
Byte count	
bytes	Number of bytes
ok	Number of bytes tagged with stmt

Example Script

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

// Select code coverage metric statement
COVerage.Option SourceMetric Statement

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

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

Full Decision Coverage Evaluation

The following diagram defines the terms used in this chapter:

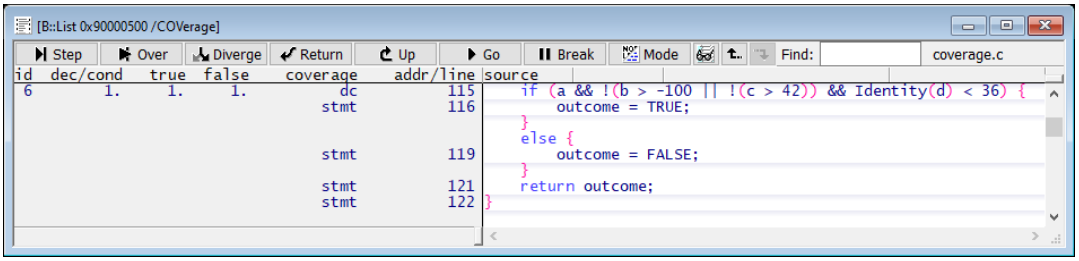


TRACE32 distinguishes between two forms of decision coverage:

- full decision coverage and
- object code coverage based decision coverage - ocb in short (for details refer to **"Object Code Based (ocb) Decision Coverage Evaluation"**, page 97)

Interpretation

TRACE32 Interpretation: A decision achieves decision coverage when all decision paths achieve statement coverage. The following screenshot illustrates this:



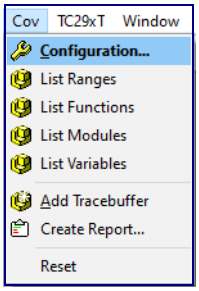
Each decision receives its own ID.

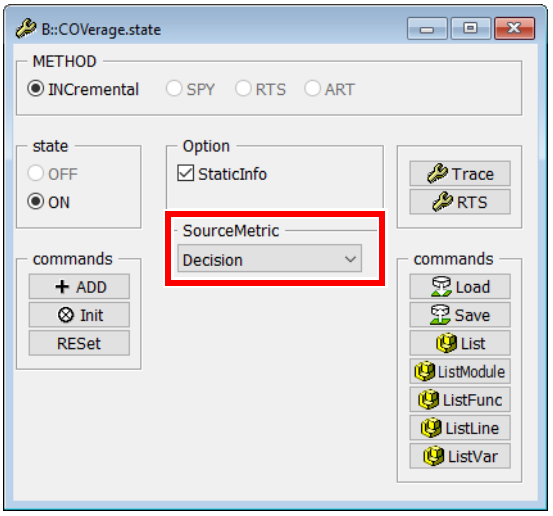
Source code lines that represent decisions are tagged as follows:

- **dc** | **incomplete**

All other source code lines use the corresponding tags for statement coverage.

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





The following commands show a tabular analysis:

COVerage.ListModule

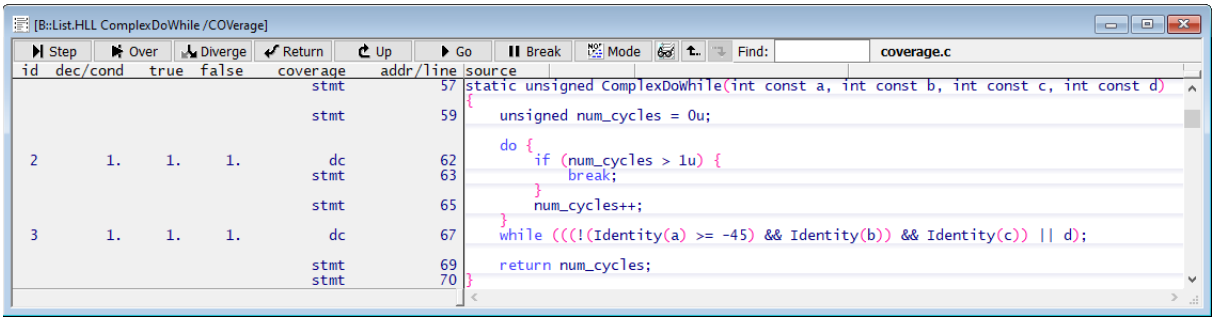
COVerage.ListFunc

The following command shows the tagging on source code level.

List.Hll /COVerage

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

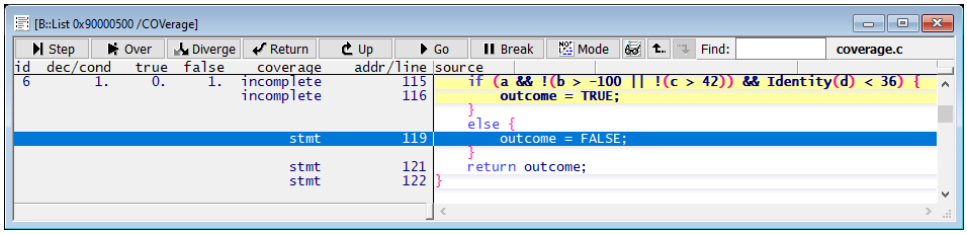
```
List.HLL ComplexDoWhile /COVerage
```



Decisions are tagged as follows:

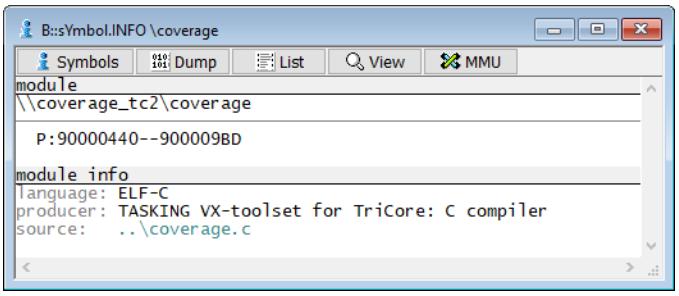
dc	Decisions have taken all possible outcomes at least once.
incomplete	There is at least one possible outcome missing for the decisions.

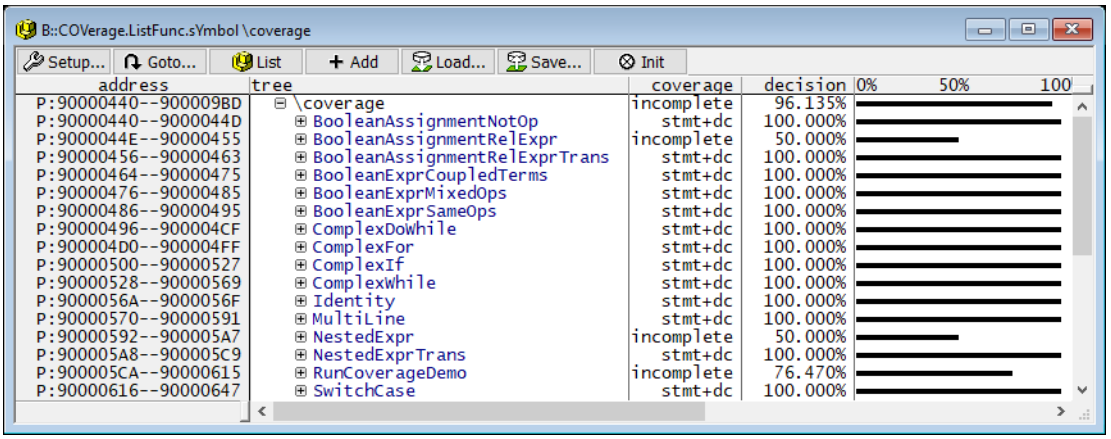
Not executed decision paths are tagged with incomplete at source code level. Already taken decision paths are tagged with stmt.



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

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





Tags for Decision Coverage

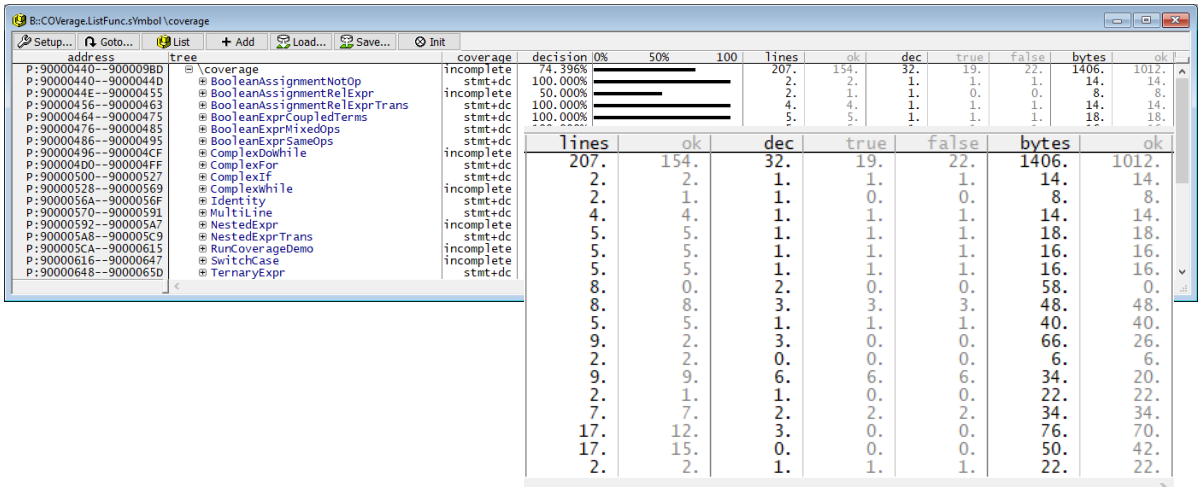
Decision coverage is achieved for a group of **HLL source code statements** as soon as all of its associated assembly instructions have been fully covered.

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

If a tag marks the coverage status of **HLL source code statements**, the following definitions apply:

- **stmt+dc**: The measured code coverage of the HLL source code statement(s) is sufficient to achieve decision coverage.
- **incomplete**: The measured code coverage of the HLL source code statement(s) is not sufficient to achieve decision coverage.

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



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

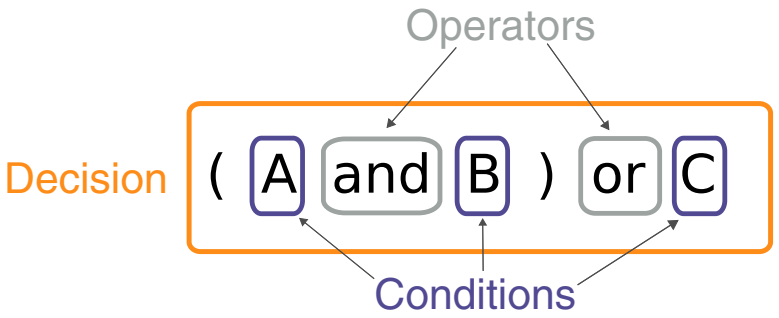
Decision count	
dec	Number of decisions within the function/module
true	Number of decisions evaluated as true
false	Number of decisions evaluated as false

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

Example Script

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

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

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

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

The following diagram defines the terms used in this chapter:



TRACE32 distinguishes between two forms of decision coverage:

- full decision coverage (for details refer to **“Full Decision Coverage Evaluation”**, page 91) and
- object code coverage based decision coverage - ocb in short

Evaluation Strategy

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

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

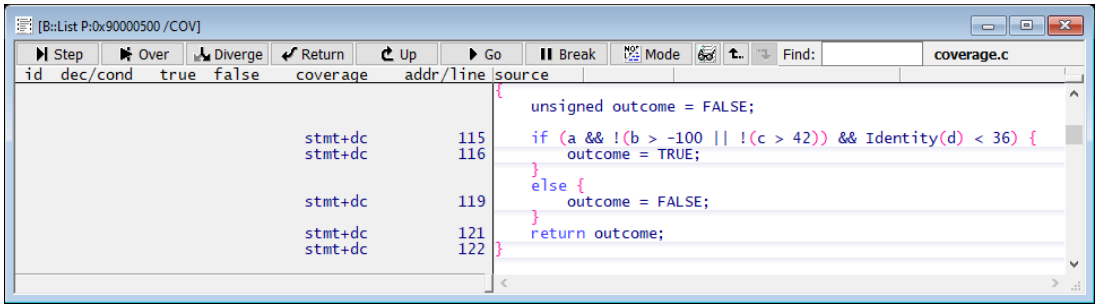
This eliminates the prerequisites necessary for full decision coverage. However, the following should be considered:

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

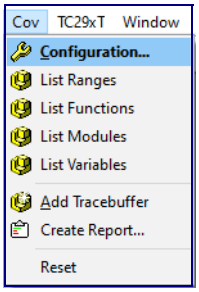
Optimized code can lead to false positive results if a condition is no longer represented by a conditional branch/instruction or the trace protocol provides no information about the state of conditional instructions. False positive means that decision coverage is indicated too early.

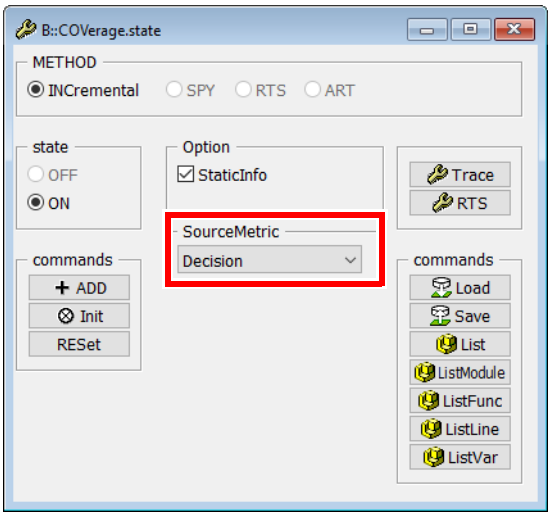
Since the source code is not analyzed for ocb decision coverage, TRACE32 does not know where decisions are located. Therefor source code lines are tagged as follows:

- **dc+stmt | incomplete**



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





The following commands show a tabular analysis:

COVerage.ListModule

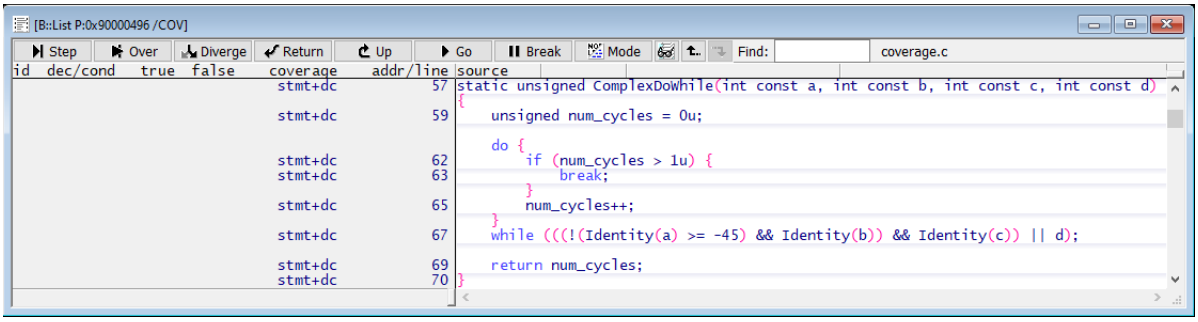
COVerage.ListFunc

The following command shows the tagging on source code level.

List.Hll /COVerage

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

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



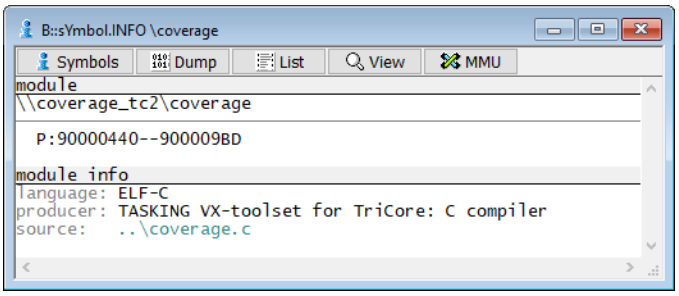
Source code lines are tagged as follows:

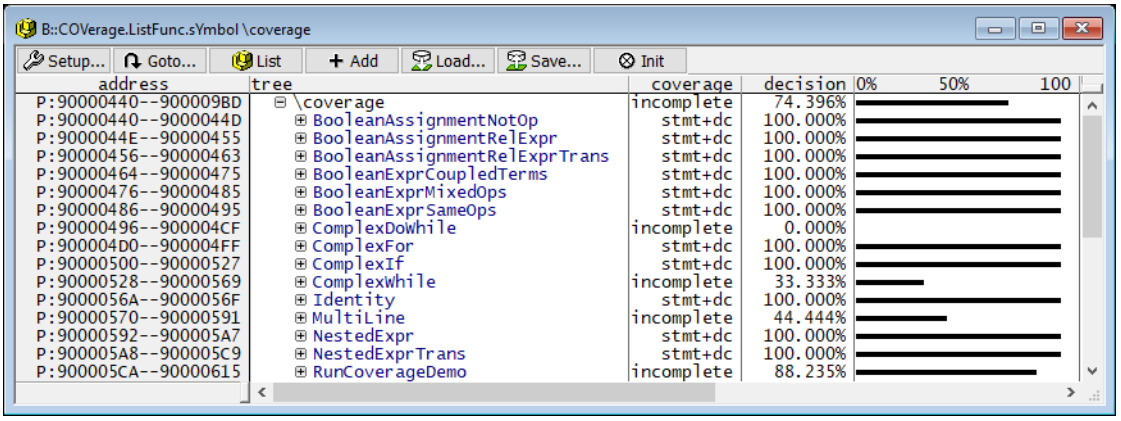
dc+stmt	The source code line achieved full object code coverage and thereby either decision or statement coverage.
incomplete	The source code line did not achieve full object code coverage and thereby no decision or statement coverage.

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

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

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

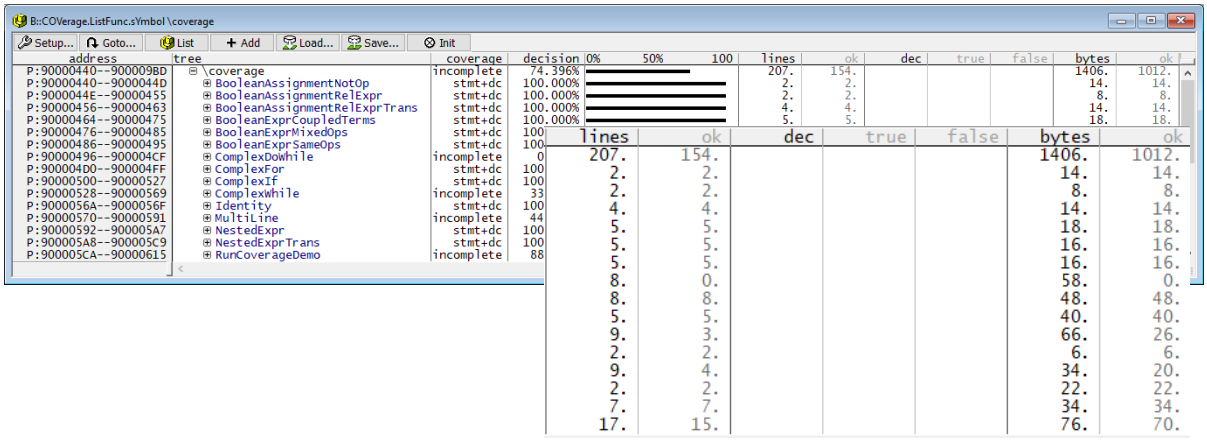




Tags for Object Code Based (ocb) Decision Coverage

- **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:



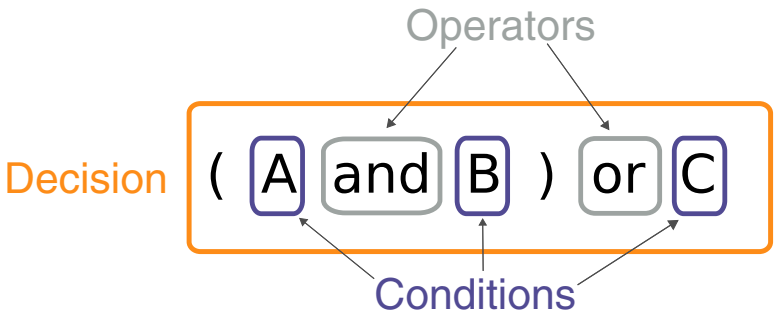
Line count	
lines	Number of source code lines within the function/module
ok	Number of source code lines tagged with stmt+dc

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

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

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

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

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

Condition Coverage Evaluation

The following diagram defines the terms used in this chapter:

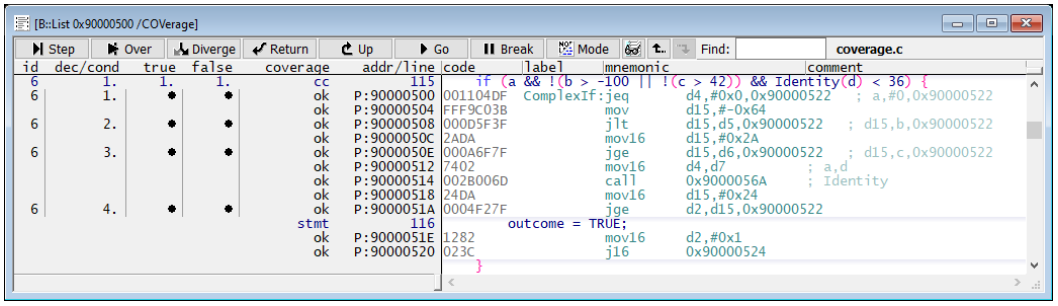


Evaluation Strategy

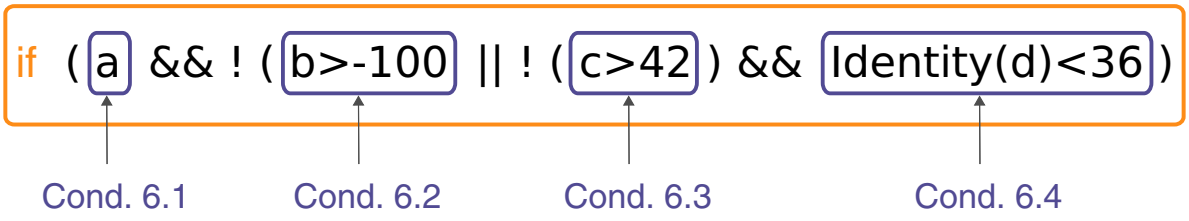
Condition coverage: All conditions in the program have evaluated both true and false.

TRACE32 Interpretation: A condition achieved condition coverage when the execution of its conditional branches/instructions results in both a true and false outcome.

Each decision receives its own ID. The atomic conditions of which the decision is composed are numbered consecutively. Each atomic condition is represented by a conditional branch/instruction.



Decision 6



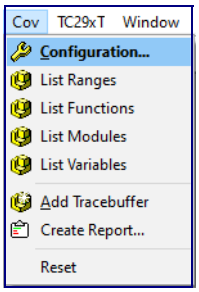
Source code lines that contain conditions are tagged as follows:

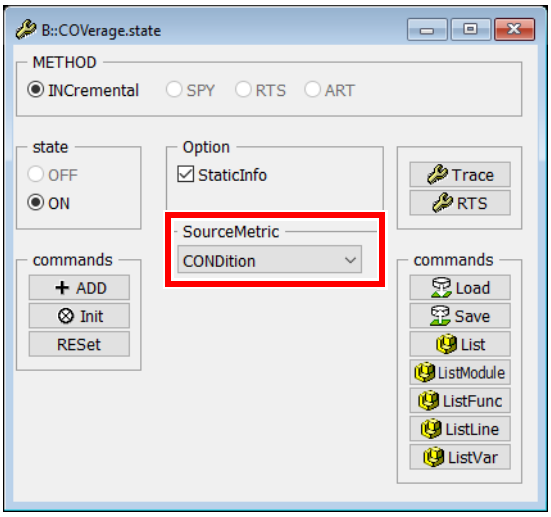
- **cc** | **incomplete**

All other source code lines use the corresponding tags for statement coverage.

Evaluation

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





The following commands show a tabular analysis:

COVerage.ListModule

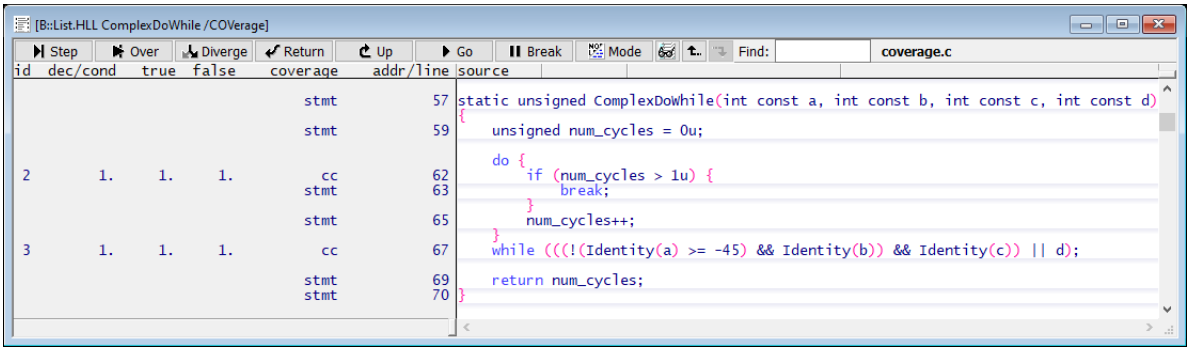
COVerage.ListFunc

The following command shows the tagging on source code level.

List.Hll /COVerage

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

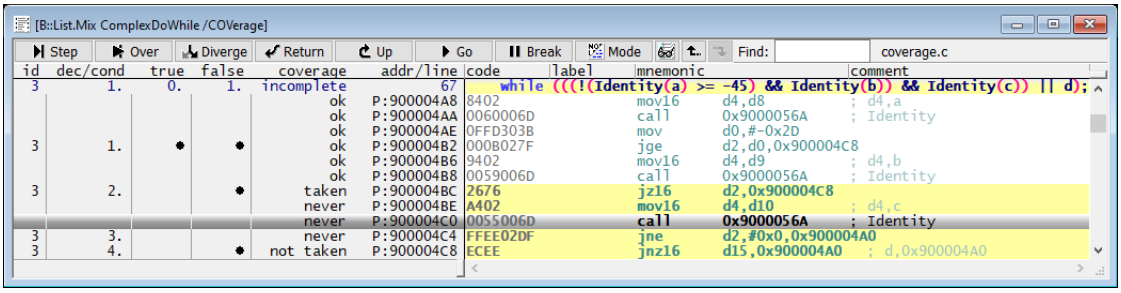
```
List.HLL ComplexDoWhile /COVerage
```



Decisions are tagged as follows:

cc	The conditions have evaluated both, true and false.
incomplete	The conditions have not evaluated both, true and false.

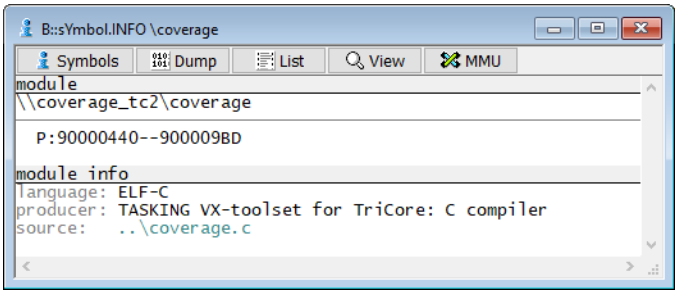
TRACE32 displays the result in **mixed mode** in such a way that it is clear which atomic conditions are still missing for a full condition coverage.

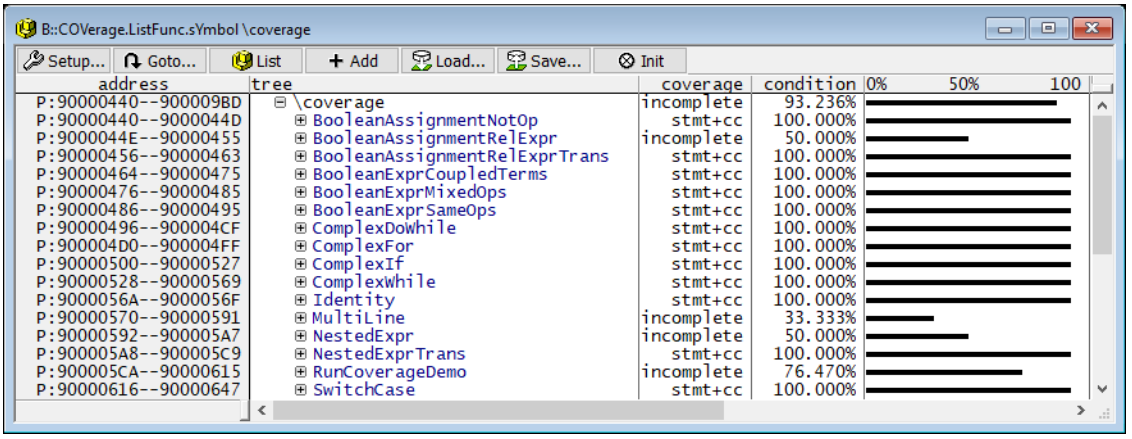


Object code instructions show the corresponding tags for object code coverage, if condition coverage is selected.

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

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

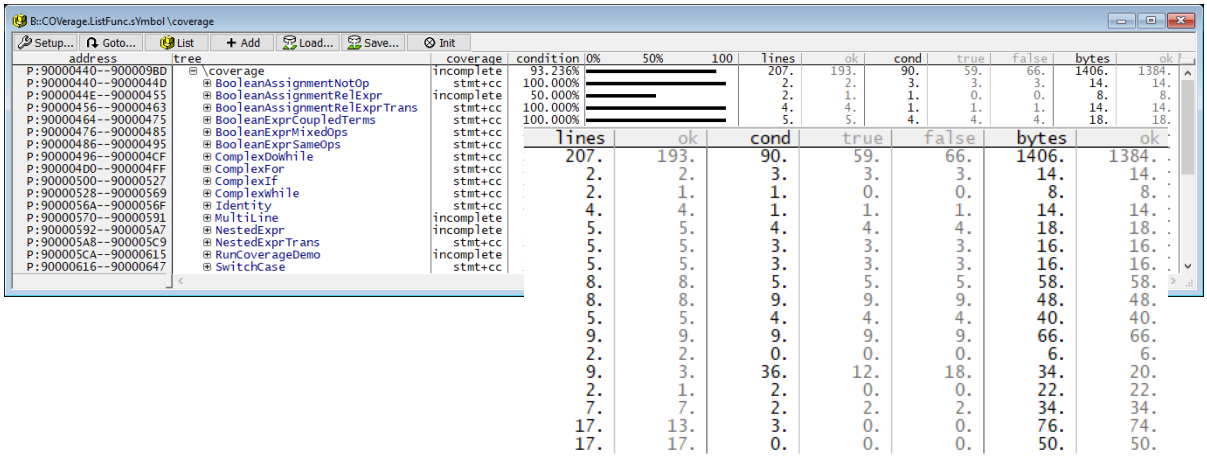




Tags for Condition Coverage

- **stmt+cc**: All source code line blocks of the function/module are tagged with cc or stmt.
- **incomplete**: At least one source code line block of the function/module is tagged with incomplete.

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



Line count	
lines	Number of source code line blocks within the function/module
ok	Number of source code line blocks tagged with cc or stmt

Condition count	
cond	Number of conditions within the function/module
true	Number of conditions evaluated as true
false	Number of conditions evaluated as false

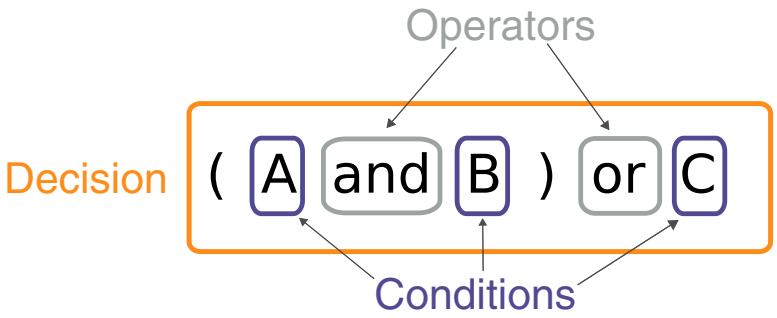
Byte count	
bytes	Number of bytes within the function/module
ok	Number of bytes tagged with cc or stmt

Example Script

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

// Select code coverage metric condition
COVerage.Option SourceMetric CONDition

// Load .eca files so that TRACE32 knows which source code lines
// represent decisions
sYmbol.ECA.LOADALL /SkipErrors

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

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

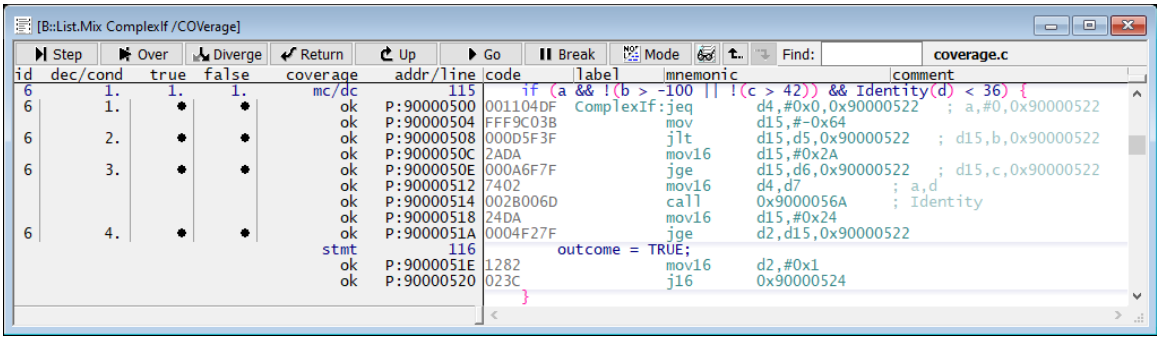
Modified Condition/Decision Coverage (MC/DC) Evaluation

The following diagram defines the terms used in this chapter:



Evaluation Strategy

Modified Condition/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. Each condition in a decision is shown to independently affect the outcome of that decision.



- Each decision receives its own ID.
- The conditions belonging to the decision are numbered consecutively.
- Each condition is represented by a conditional branch/instruction.

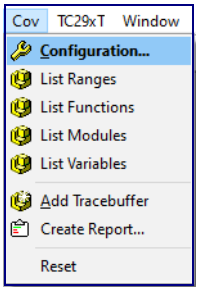
The point for true is set in the true column if the condition has been independently tested for true. The same applies to false.

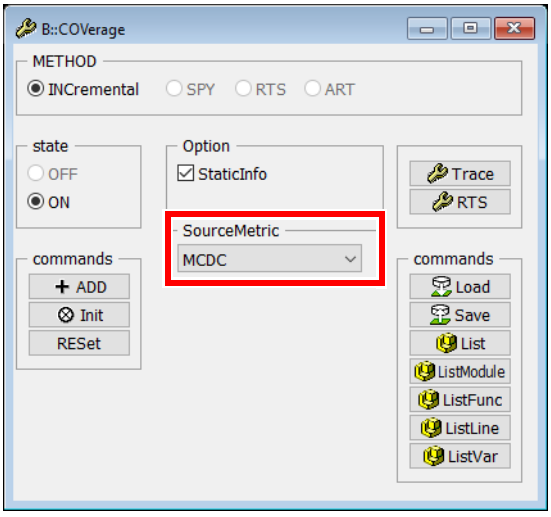
Source code lines that contain decisions are tagged as follows:

- **mc/dc** | **incomplete**

All other source code lines use the corresponding tags for statement coverage.

If you want to use the trace data stored in the code coverage system for MC/DC, select the SourceMetric **MCDC** in the **COVerage state configuration** or use the command **COVerage.Option SourceMetric** MCDC.





The following commands show a tabular analysis:

COVerage.ListModule

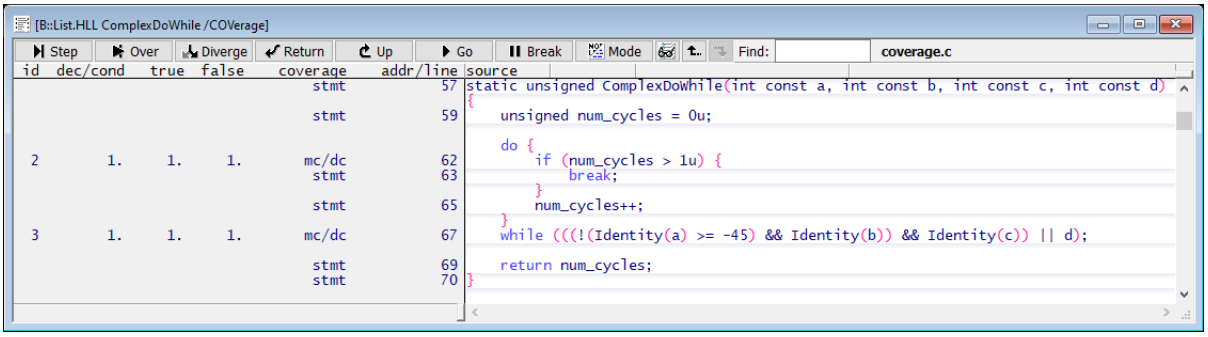
COVerage.ListFunc

The following command shows the tagging on source code level.

List.HIl /COVerage

This TRACE32 command displays the MC/DC coverage tagging for the function *ComplexDoWhile*:

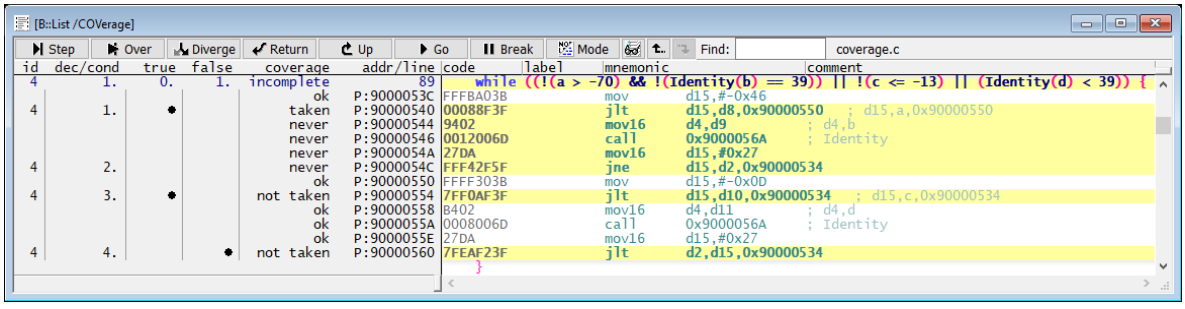
```
List.HLL ComplexDoWhile /COVerage
```



Decisions are tagged as follows:

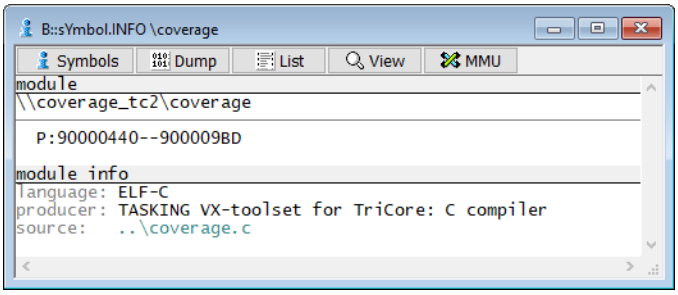
mc/dc	Each condition in a decision is shown to independently affect the outcome of that decision.
incomplete	There is at least one condition in the decision for which has not yet proven to independently affect the outcome of the decision.

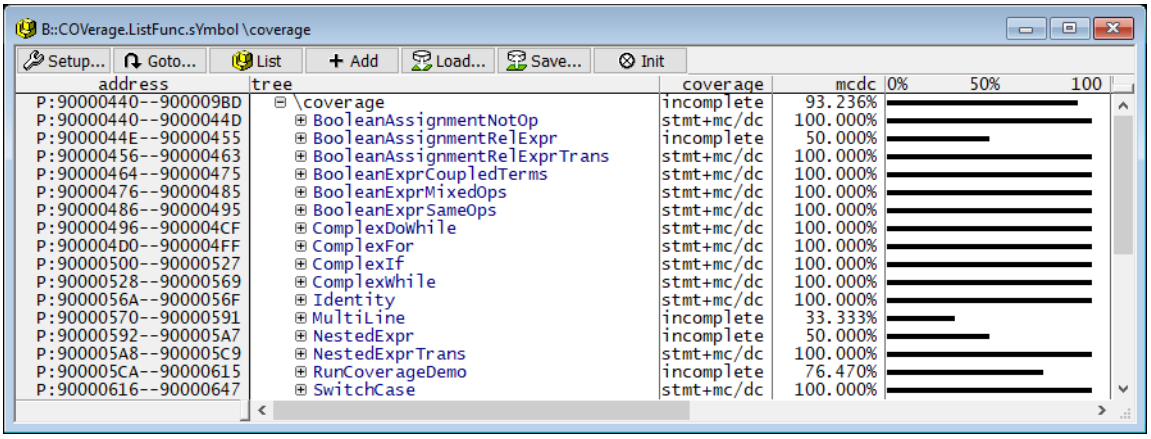
TRACE32 displays the result in **mixed mode** in such a way that it is clear which conditions are still missing for MC/DC.



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

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





Tags for Modified Condition/Decision Coverage (MC/DC)

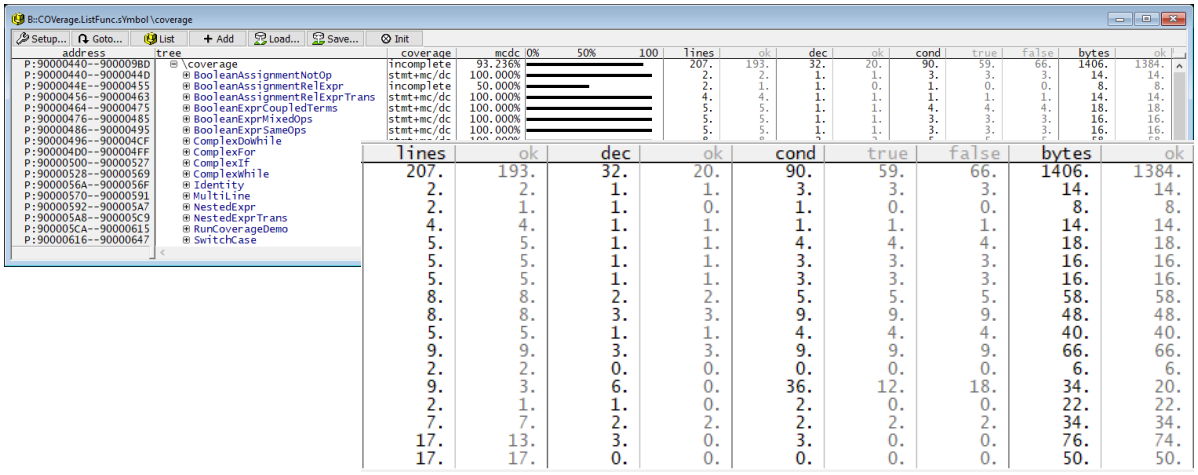
MC/DC is achieved for a group of **HLL source code statements** as soon as the independence effect of all of its associated conditional branches/instructions has been demonstrated.

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

If a tag marks the coverage status of **HLL source code statements**, the following definitions apply:

- **stmt+mc/dc**: The range contains one or more HLL source code statements. The measured code coverage of the HLL source code statement(s) is sufficient to achieve MC/DC.
- **mc/dc**: The HLL source code statement(s) contain a decision. The measured code coverage of the HLL source code statement(s) is sufficient to achieve MC/DC.
- **stmt**: The HLL source code statement(s) do not contain a decision. The measured code coverage of the HLL source code statement(s) is sufficient to achieve statement coverage.
- **incomplete**: The measured code coverage of the HLL source code statement(s) is not sufficient to achieve MC/DC.

Further details are displayed if 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 mc/dc or stmt

Decision count	
dec	Number of decisions within the function/module
ok	Number of decisions tagged with mc/dc

Condition count	
cond	Number of conditions within the function/module
true	Number of conditions that have been independently tested for true
false	Number of conditions that have been independently tested for false

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

Example Script

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

// Select code coverage metric MC/DC
COVerage.Option SourceMetric MCDC

// Load .eca files so that TRACE32 knows which source code lines
// represent decisions
sYmbol.ECA.LOADALL /SkipErrors

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

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

Function Coverage Evaluation

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

TRACE32 interpretation: A function achieves function coverage when at least one corresponding object code instruction has been executed.

Functions are tagged as follows:

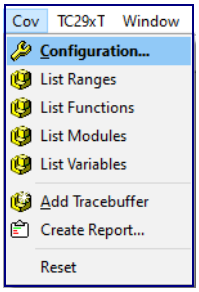
- **func** | **incomplete**

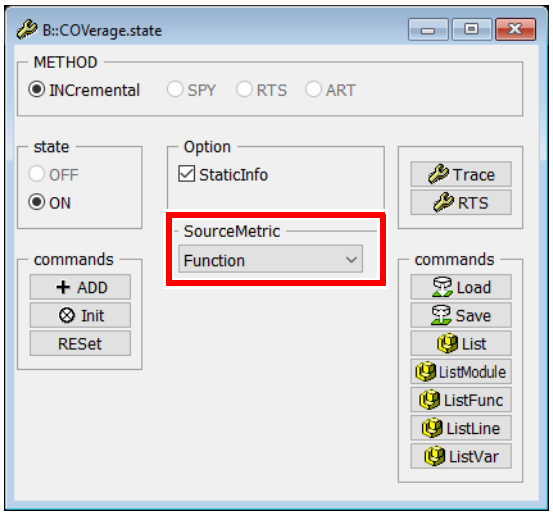
Source code lines show the corresponding tags for statement coverage, if function coverage is performed.

Object code coverage tagging is applied to instructions.

Evaluation Strategy

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





The following command shows a tabular analysis:

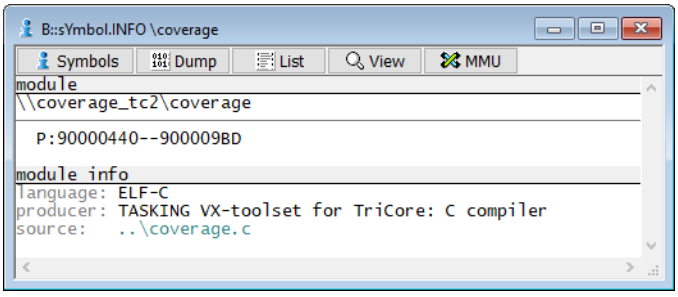
COVerage.ListModule

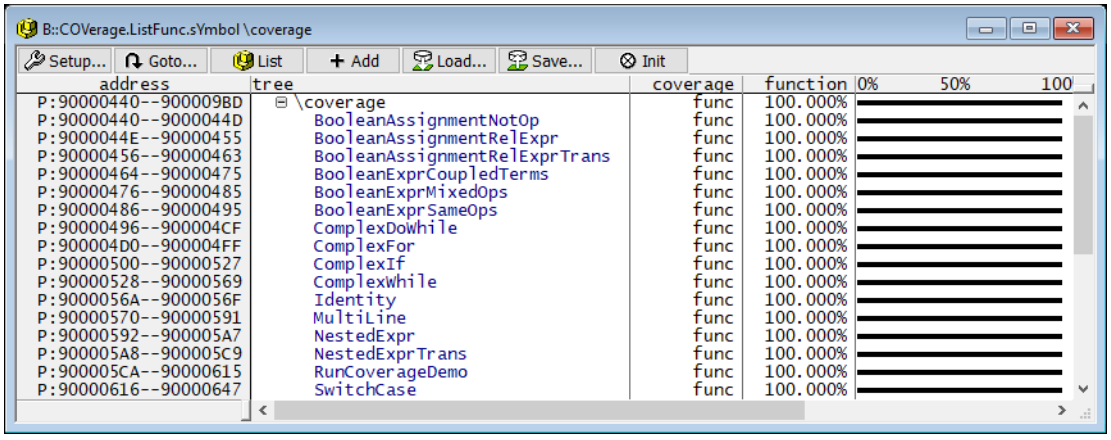
The following command shows the tagging at function level.

COVerage.ListFunc

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

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



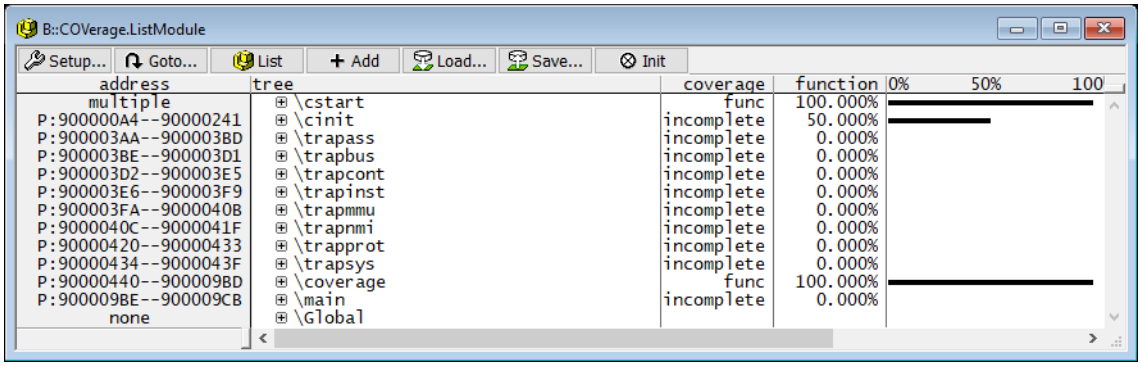


The functions are tagged as follows:

func	At least one function's object code instructions has been executed.
incomplete	None of the function's object code instructions has been executed.

This TRACE32 command displays a tabular analysis of all modules.

```
COVerage.ListModule
```



Tags for Function Coverage

Function coverage is achieved for a function as soon as soon as its function body has been partially executed.

- **func**: All functions of the module have achieved function coverage.
- **incomplete**: At least one function of the module has not achieved function coverage.

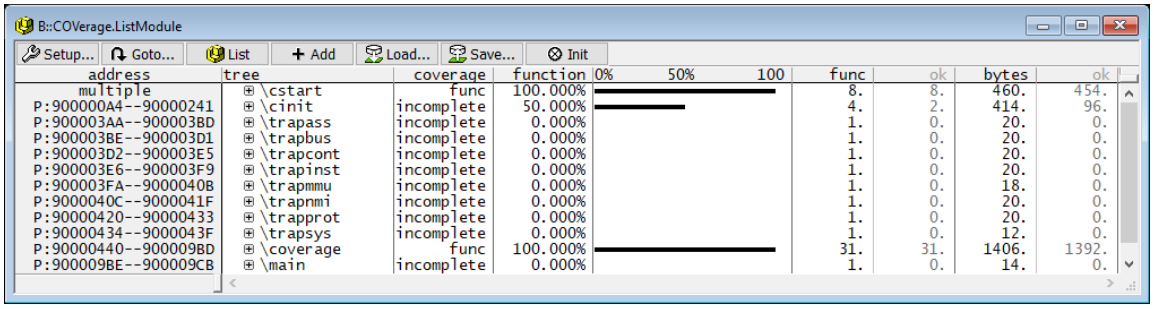
If a tag marks the coverage status of a **function**, the following definitions apply:

- **func**: The measured code coverage of the function(s) is sufficient to achieve function coverage.
- **incomplete**: The measured code coverage of the function(s) is not sufficient to achieve function coverage.

If a tag marks the coverage status of **HLL source code statements**, the following definitions apply:

- **stmt**: The measured code coverage of the HLL source code statement(s) is sufficient to achieve statement coverage.
- **incomplete**: The measured code coverage of the HLL source code statement(s) is not sufficient to achieve statement coverage.

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



Function count	
func	Number of functions
ok	Number of functions tagged with func

Byte count	
bytes	Number of bytes
ok	Number of bytes tagged with func

Example Script

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

// Select code coverage metric function
COVerage.Option SourceMetric Function

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

// List code coverage results at module level
COVerage.ListModule.sYmbol \coverage
```

Expert Usage

The following commands provide details on inlined functions:

sYmbol.List.InlineBlock	List inlined code blocks
COVerage.ListInlineBlock	List object code coverage for inlined blocks

Call Coverage Evaluation

Call Coverage: Every function call has been executed at least once.

Please note that TRACE32 also includes calls to libraries (e.g. software floating-point libraries) in its call.

TRACE32 interpretation: A function achieves call coverage when each unconditional branch that represents a function call has been executed a least once.

Functions are tagged as follows:

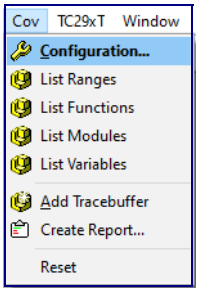
- **call** | **incomplete**

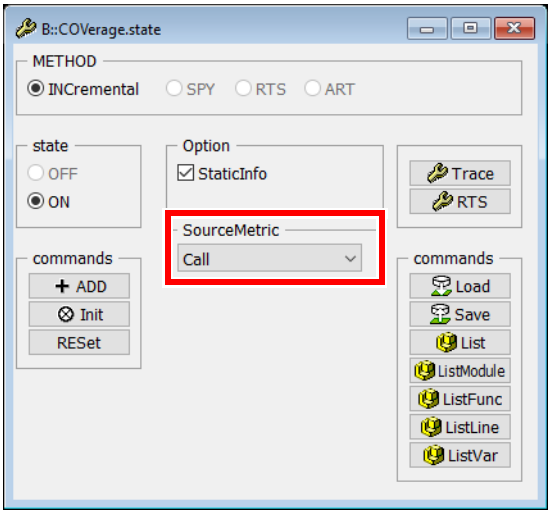
Source code lines show the corresponding tags for statement coverage, if call coverage is performed.

Object code coverage tagging is applied to instructions.

Evaluation

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





The following command shows a tabular analysis:

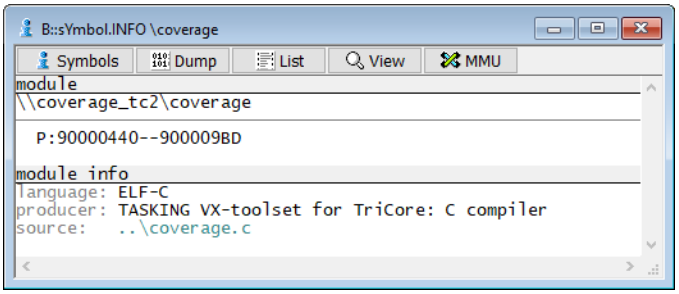
COVerage.ListModule

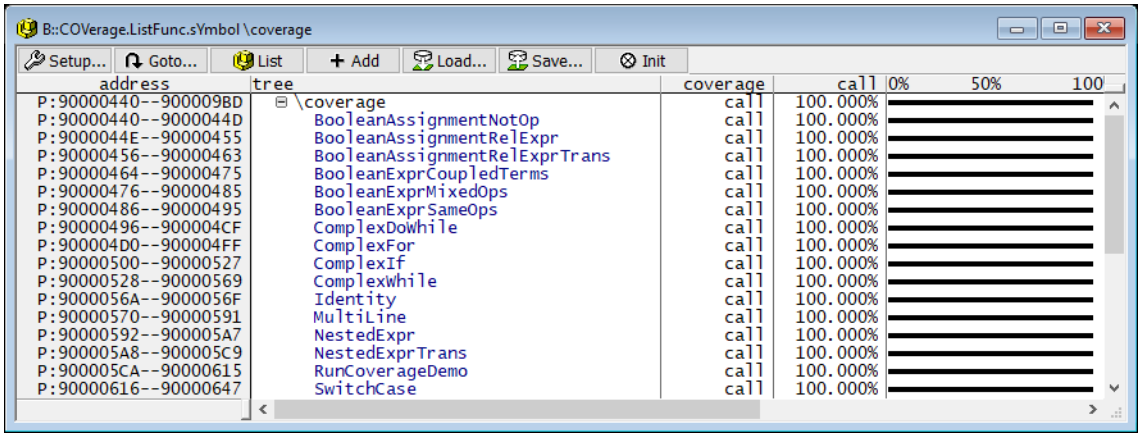
The following command shows the tagging at function level.

COVerage.ListFunc

This TRACE32 command displays the call coverage tagging for all functions of the "coverage" module. A module usually corresponds to a source code line.

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



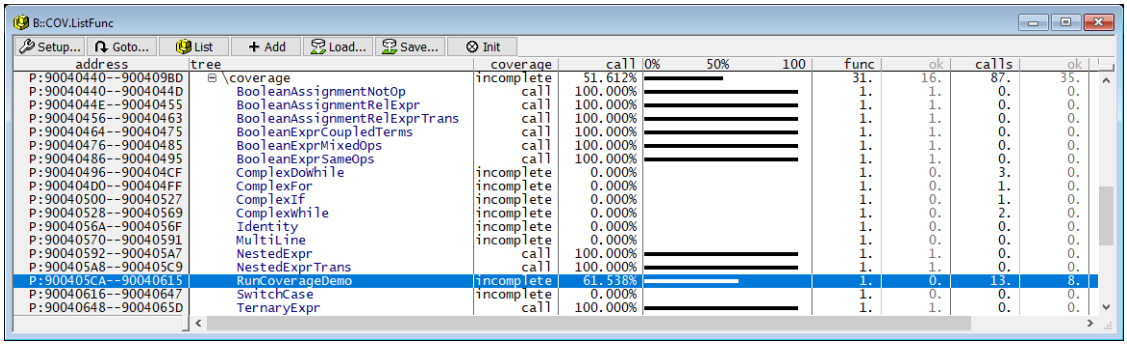


The functions are tagged as follows:

call	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	At least one unconditional branch that represent a function call has not been executed. No object code instruction generated for the function has been executed for all call-less functions.

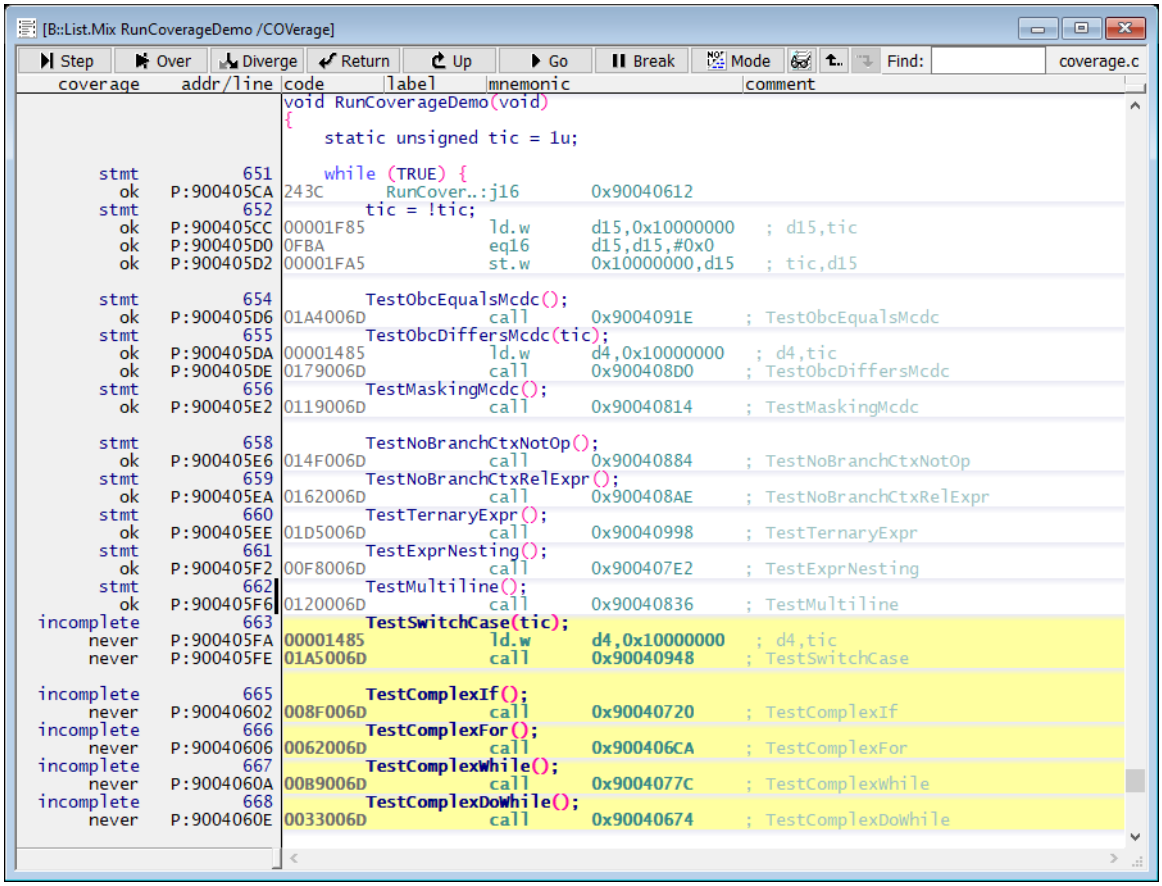
The full-width **COVerage.ListFunc** window provides details on the function calls:

- **calls column:** number of function calls within the function
- **ok column:** number of function calls that have already been executed



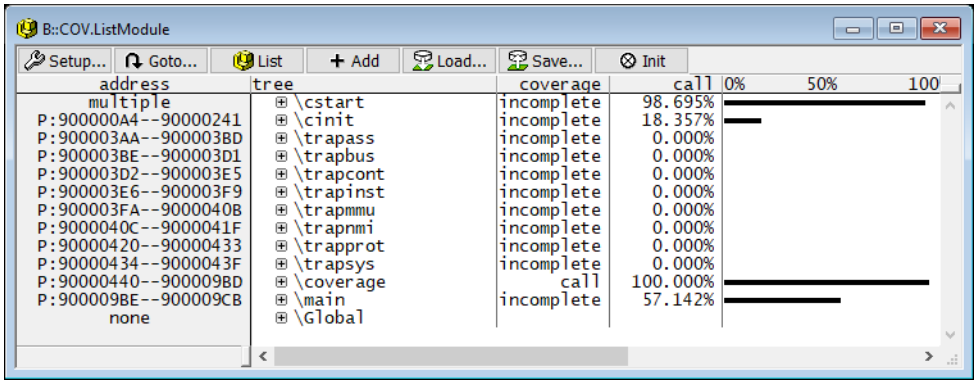
If a function is tagged as incomplete you can inspect its details. Either by doing a left mouse double click on the function‘s name or by using the following command:

```
List.Mix RunCoverageDemo /COVerage
```



This TRACE32 command displays a tabular analysis of all modules.

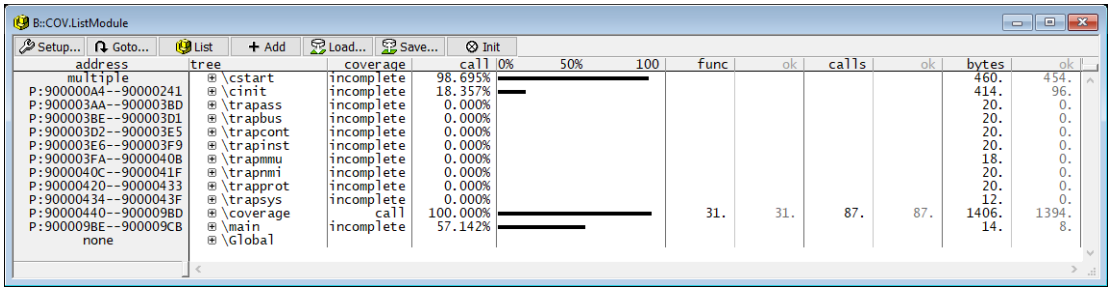
```
COVerage.ListModule
```



The following tags are used for the summary:

- **call**: All functions of the module are tagged with call.
- **incomplete**: At least one function of the module is tagged with incomplete.

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



Function count	
func	Number of functions
ok	Number of functions tagged with call

Byte count

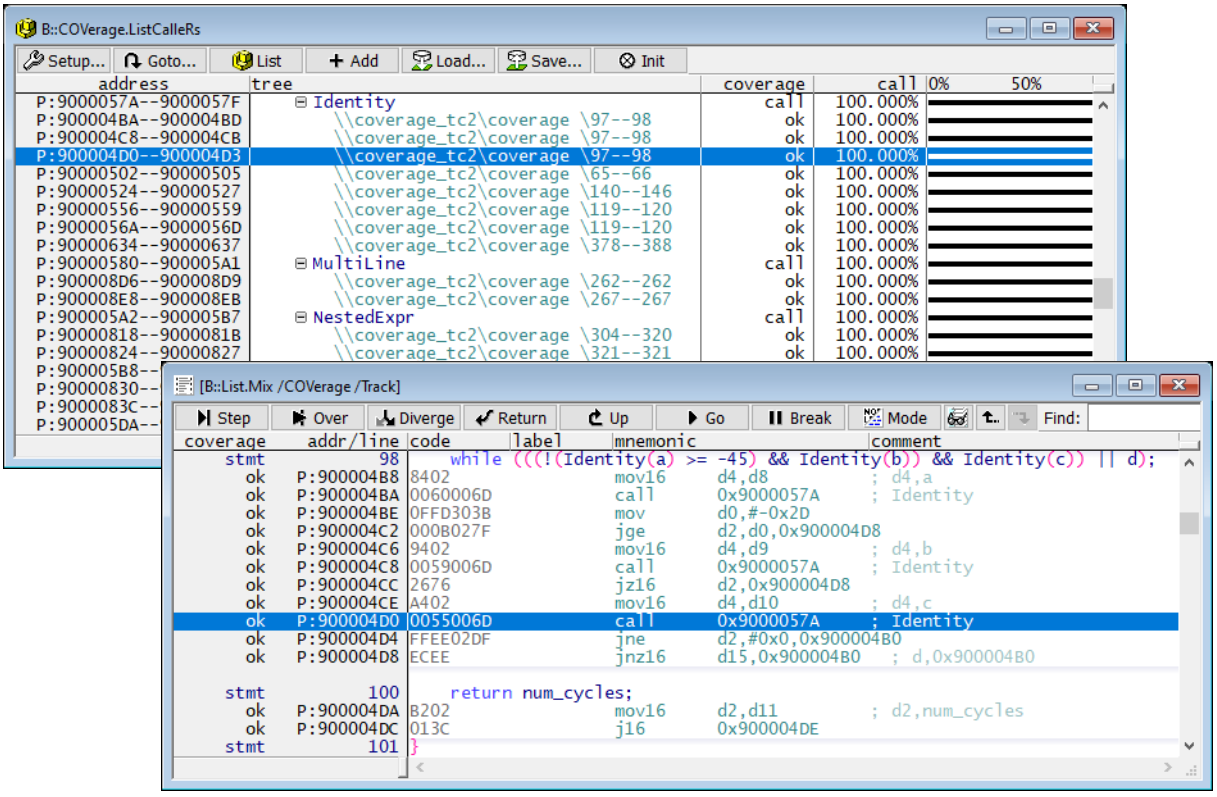
bytes	Number of bytes
ok	Number of bytes tagged with call

Details on Callers and Calles

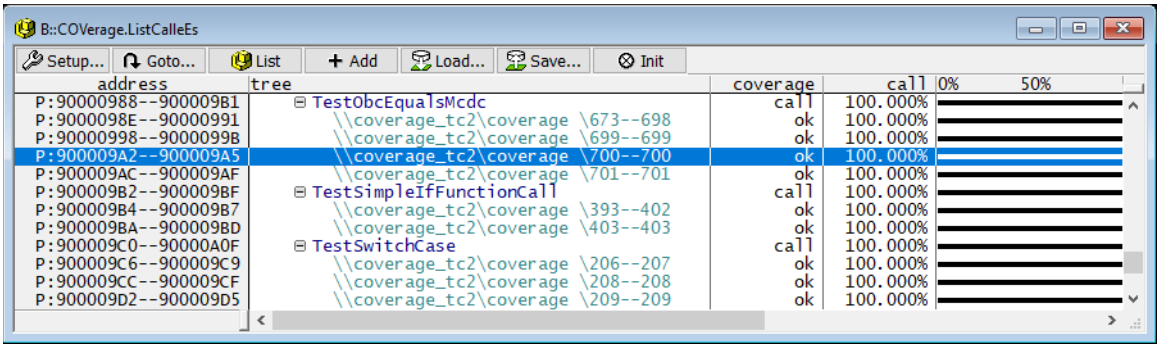
For a detailed analysis it is helpful to get details about the calling and the called functions.

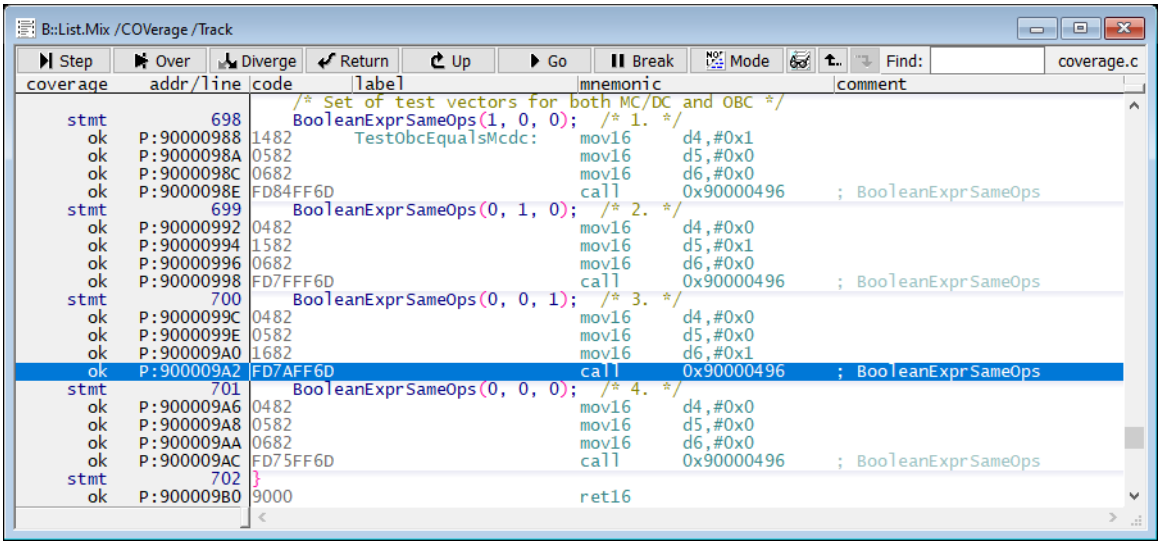
COVerage.ListCalleRs	Display call coverage with caller details at source code line level
COVerage.ListCalleEs	Display call coverage with callee details at source code line level
List.Mix /COVerage /Track	Display a source listing that displays source and object code. This window is used here to inspect the object code details.

All callers of the function **Identity** are inspected in this example. The COVerage.ListCalleRs window, displays all source code lines from which the function **Identity** is called. If you select a source code line, you can inspect the corresponding object code in the List.Mix window. This is enabled by the Track option.



All call made by the function **TestObcEqualsMcdc** are inspected in this example. The COVerage.ListCalleEs window, displays all source code lines which represent a function call. If you select a source code line, you can inspect the calls in detail in the List.Mix window. This is enabled by the Track option.





Example Script

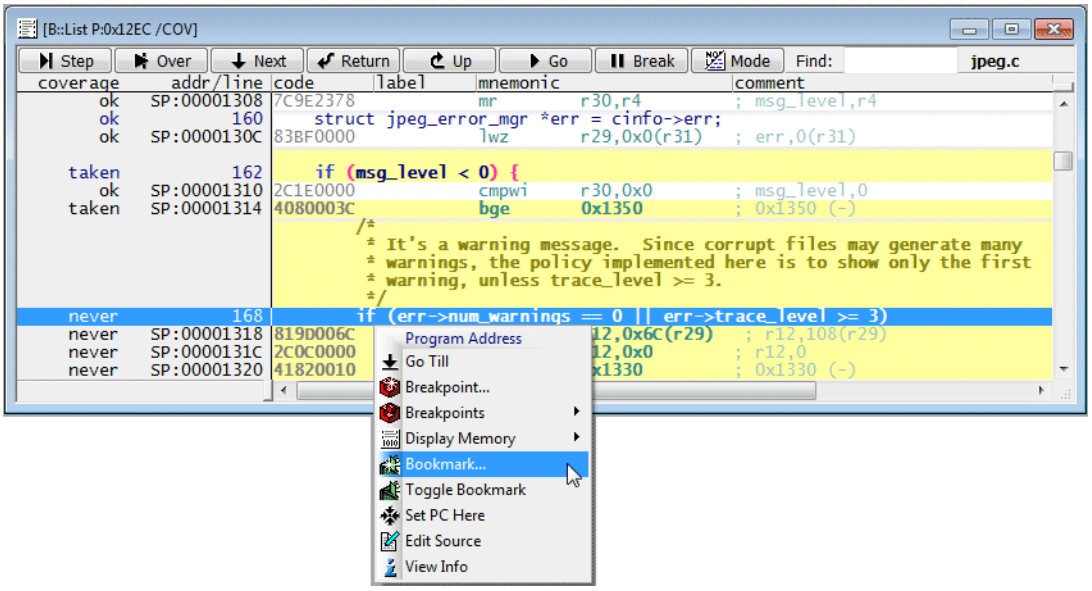
```
// Demo script "~~/demo/t32cast/eca/measure_mcdc.cmm"

// Select code coverage metric call
COVerage.Option SourceMetric Call

// Load .eca files so that TRACE32 knows which source code lines
// represent function calls
sYmbol.ECA.LOADALL /SkipErrors

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

// List code coverage results at module level
COVerage.ListModule.sYmbol \coverage
```

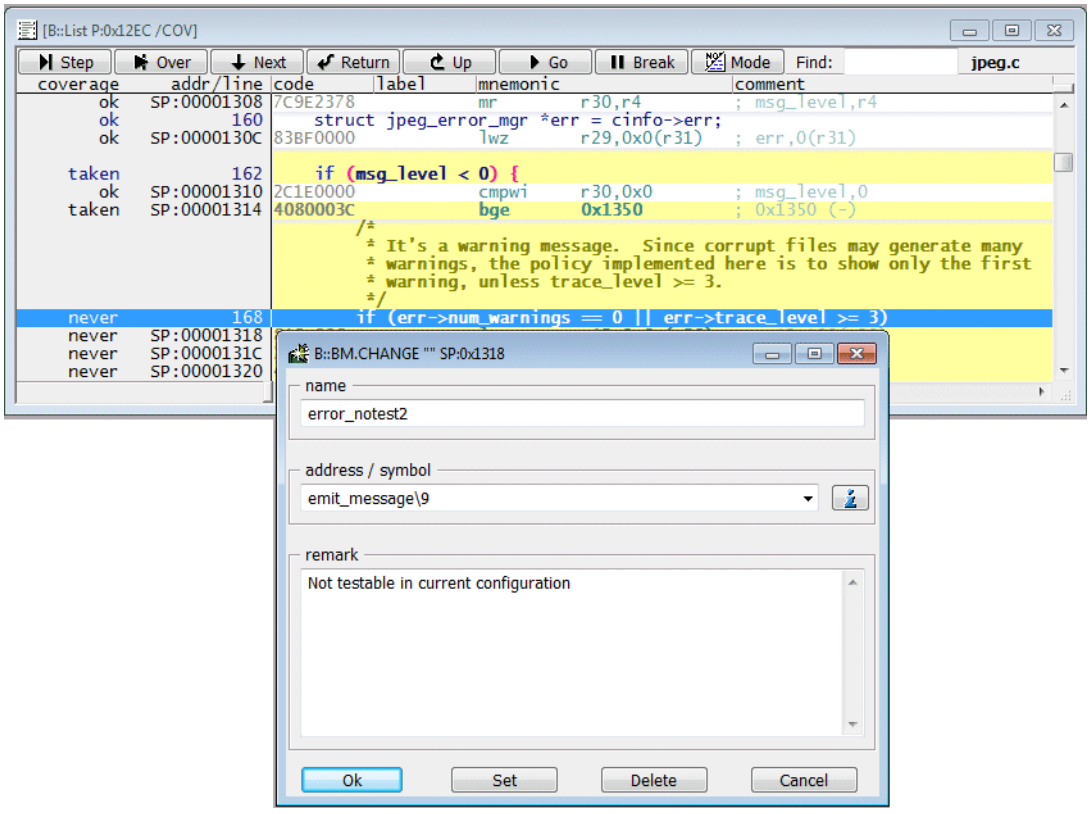
The following commands provide details on inlined functions:

sYmbol.List.InlineBlock	List inlined code blocks
COVerage.ListInlineBlock	List object code coverage for inlined blocks

Comment Your Results

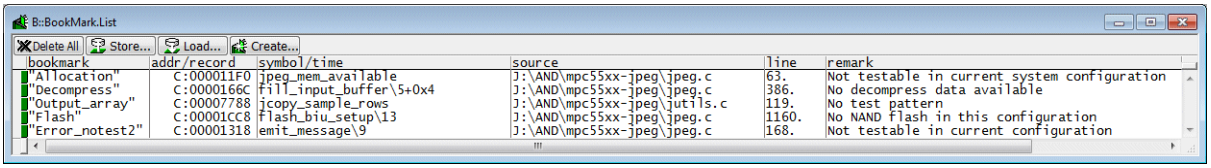
Address-based bookmarks can be used to comment not covered code ranges, which are fine but not testable in the current system configuration.





List all bookmarks:

BookMark.List



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

STOre *<file>* **BookMark**

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.
-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 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 the input files.

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

Example 3

Generate an accumulated JSON file.
- specify the files name for the accumulated JSON.
- specify the input files.

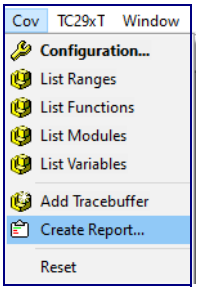
```
t32covtool --output-json sum.json
export_unittest1.json export_unittest2.json export_unittest3.json
```

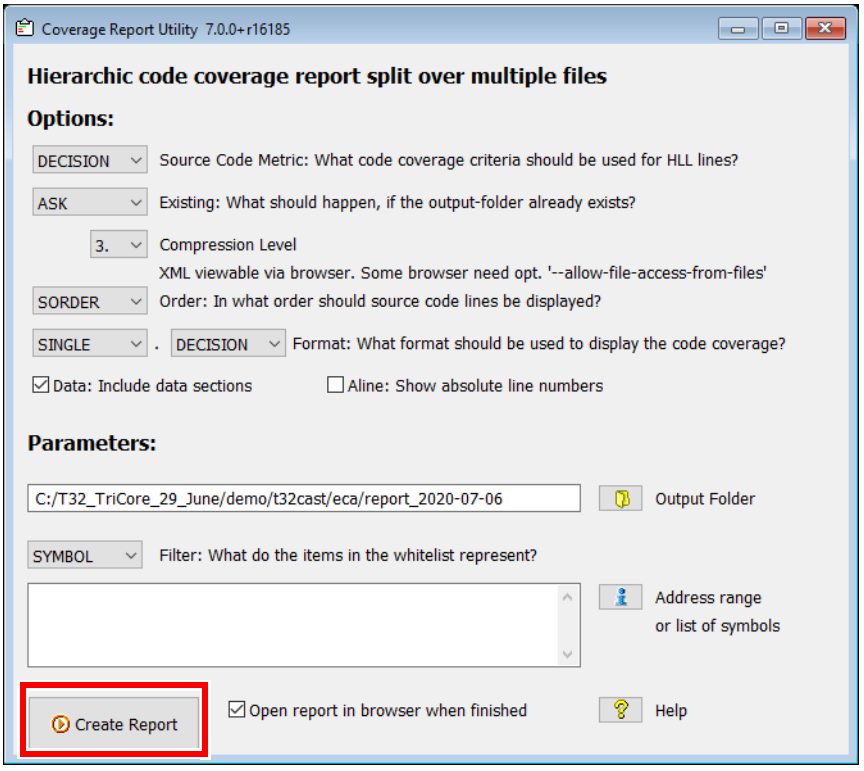
You can find a sample script for using the command line tool **t32covtool** at
`~~/demo/coverage/merge_demo/merge_unittests/demo.cmm`.

Appendix A: TRACE32 Coverage Report Utility

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

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





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

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

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

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

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

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

Note

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

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

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

…                                     ; Create your report

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

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

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

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

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

…                                       ; Create your report

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

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

Appendix B: Assemble Multiple Test Runs at Address Level

There are two ways to assemble multiple test runs.

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

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

Save and Restore Code Coverage Measurement

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

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

- Your TRACE32 debug and trace tool connected to your target hardware.
- Alternatively you can use a TRACE32 Instruction Set Simulator (see **"TRACE32 Instruction Set Simulator"** in TRACE32 Installation Guide, page 56 (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 **"TRACE32 Instruction Set Simulator"** in TRACE32 Installation Guide, page 56 (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
```

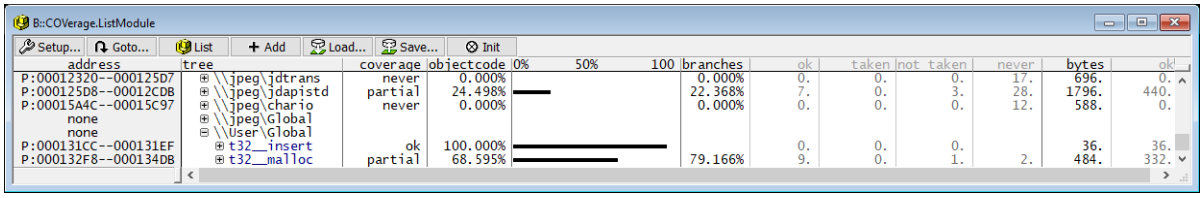
Object Code Coverage

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.

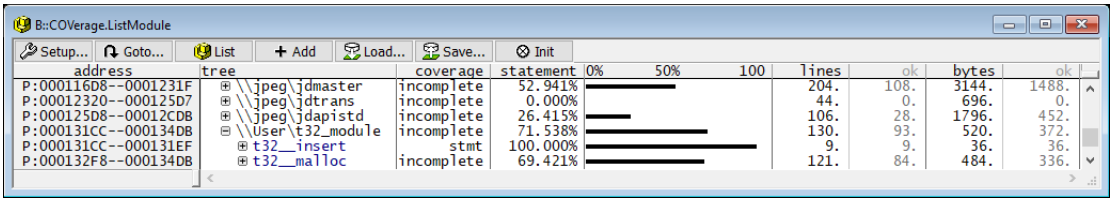
Code that is not part of a source code function is discarded for coverage. If you want to include this code you have to assign a function to it:

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.

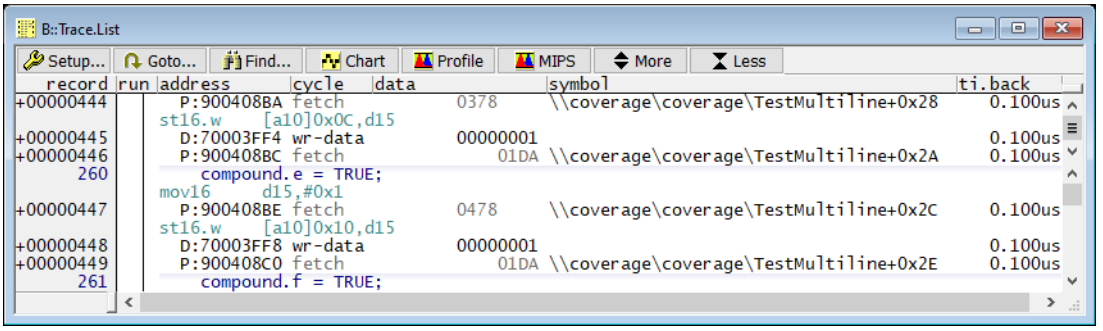
Metric	Tag	Description
Decision	stmt+dc	All assembler lines are tagged with ok.
CONDition	stmt+cc	All assembler lines are tagged with ok.
MCDC	stmt+mc/dc	All assembler lines are tagged with ok.
Function	func	All assembler lines are tagged with ok.
Call	call	All assembler lines are tagged with ok.

Appendix D: Data Coverage

Trace Data Collection

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

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



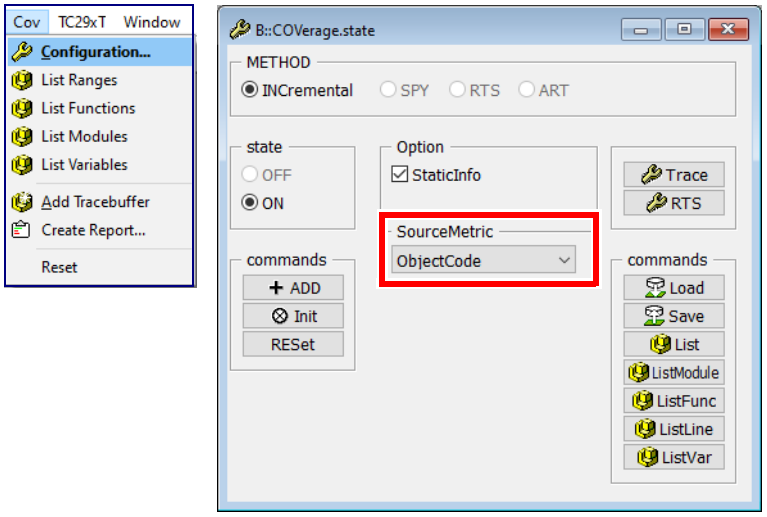
If you want to use an onchip trace or an offchip trace port for data tracing, please refer to the following documents for setup details:

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

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

It is recommended to use incremental coverage for data coverage (see **"Incremental Code Coverage"**, page 57).

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

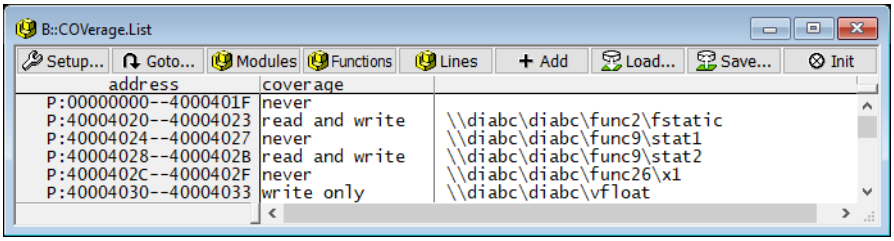
COVerage.ListVar

The following command shows the tagging per address.

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

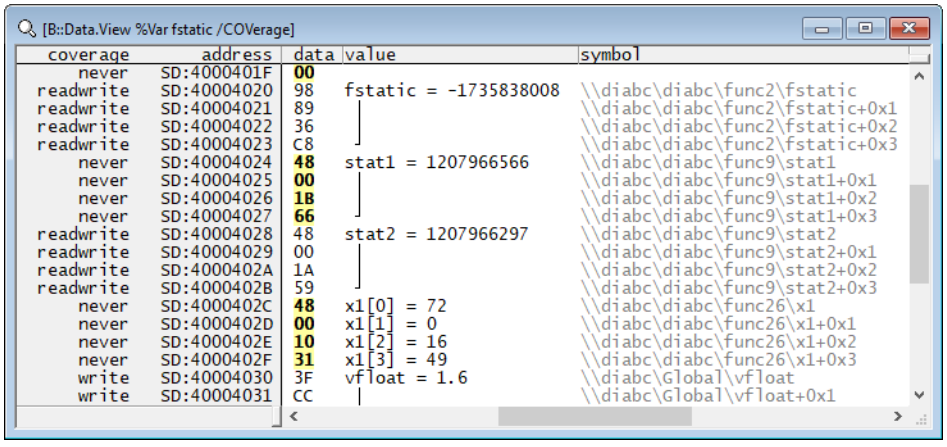
This TRACE32 command shows the coverage tagging on address range level:

```
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
```

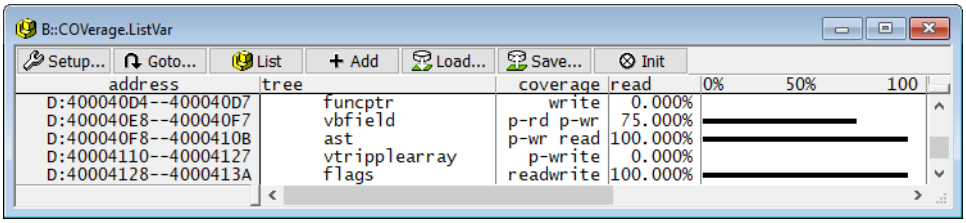


The data addresses are tagged as follow:

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:

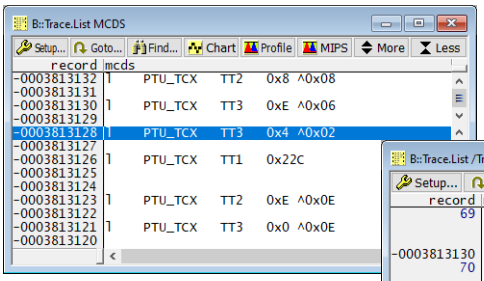
readwrite	Read and write accesses were performed for all addresses within the address range of the variable.
read	Only read accesses were performed for all addresses within the address range of the variable.
write	Only write accesses were performed for all addresses within the address range of the variable.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

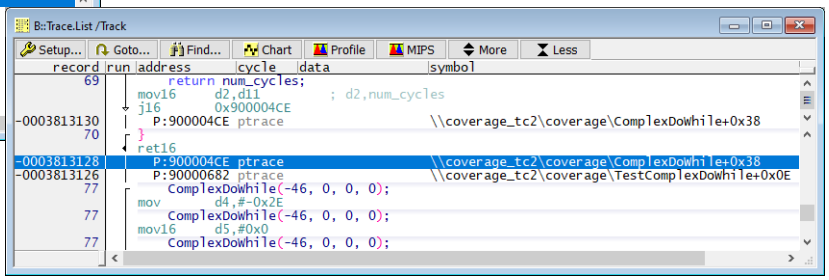
A complete list of all data coverage tags can be found in **"Appendix E: Data Coverage in Detail"**, page 154.

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

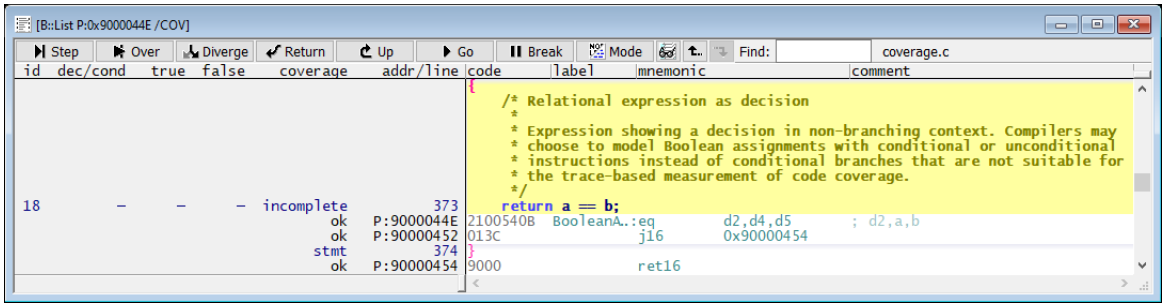
The following coding guidelines are recommended for full decision and condition coverage as well as for MC/DC. If you follow these coding guidelines you avoid false negative results. False negative means that a decision/conditions is tagged as incomplete although coverage has already been achieved.

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

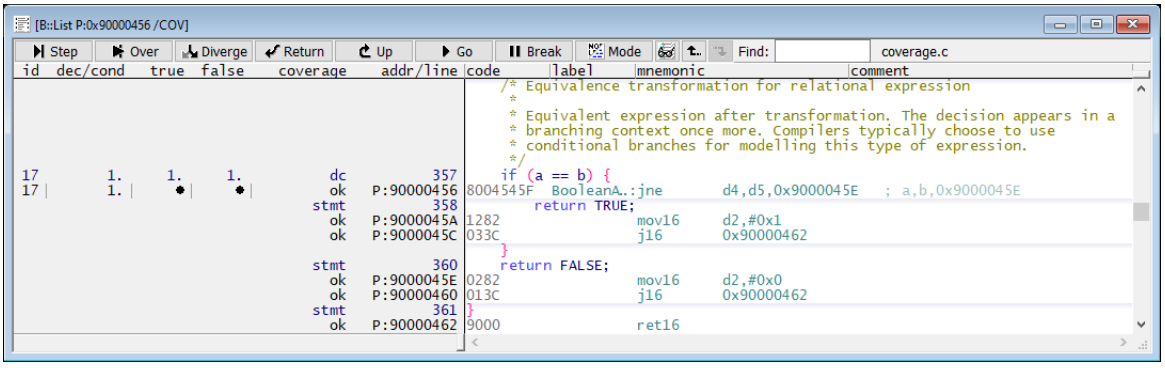
Avoid Simple Decisions in Assignment Context

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

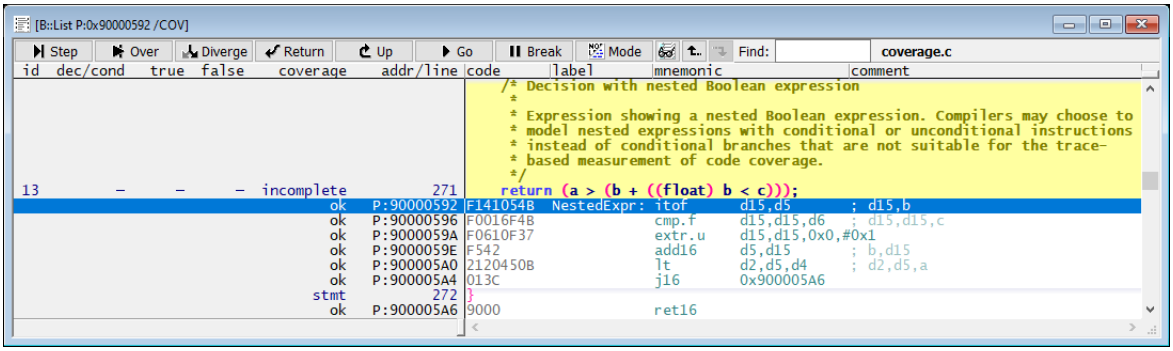
It is very likely that not all conditions are represented by a conditional branch/instruction at object code level.

This is illustrated by the following example:

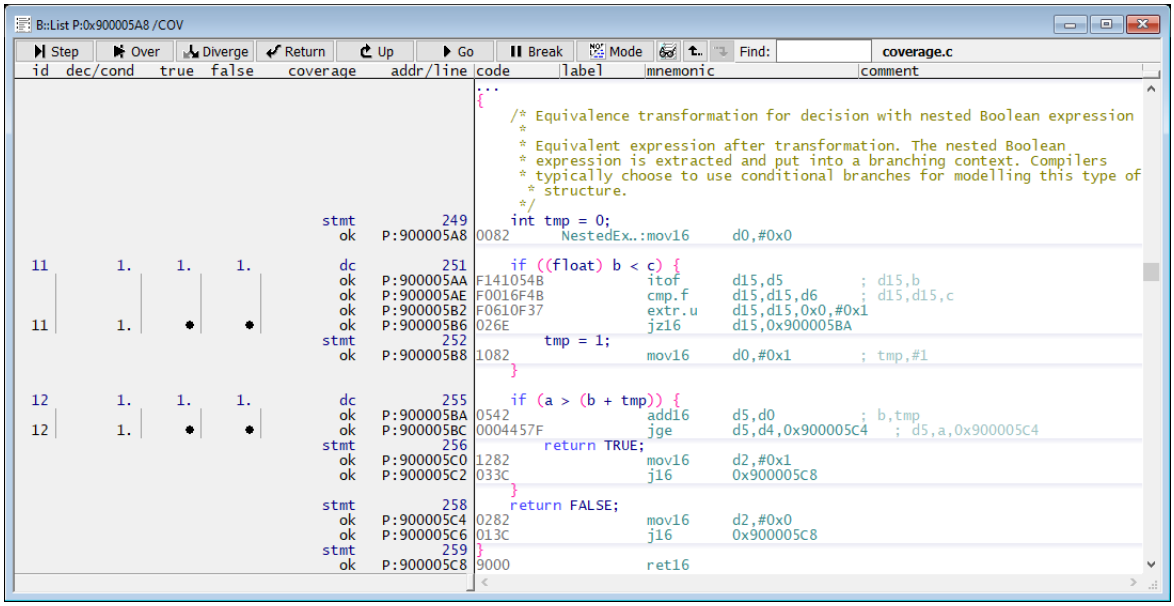
```
; source code not suitable for
; trace-based code coverage

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.