LAUTERBACH
DEVELOPMENT TOOLS

Application Note for Trace-Based Code Coverage

Release 09.2023

MANUAL

Application Note for Trace-Based Code Coverage

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ..... 🗁

Trace Application Notes ..... 🗁

Trace Analysis ..... 🗁

Application Note for Trace-Based Code Coverage **1**

History **5**

Introduction **6**

Intended Audience 6

Prerequisites 7

Trace-Based Code Coverage and Certification **8**

Trace Data Collection Overview **9**

TRACE32 Tool Configurations 9

Choose the Appropriate Trace Data Collection Method 10

Preconditions 12

Reduce the Amount of Trace Data 12

Ensure a Fault-Free Trace Recording 13

Disable Timestamps for Trace Streaming 14

SMP Multicore Systems 14

Trace Data Collection Modes **15**

Incremental Code Coverage 15

Data Collection 15

Example Script 17

Summary 17

Incremental Code Coverage in STREAM Mode 18

Data Collection 18

Example Script 21

Summary 21

RTS Mode Code Coverage 22

Data Collection 22

Example Scripts 25

Summary 26

SPY Mode Code Coverage 27

Operation States 27

Data Collection 29

Example Script 31
Summary 32
Code Coverage with Virtual Targets 33
ART Mode Code Coverage 35
Data Collection 36
Example Script 37

Supported Code Coverage Metrics 38
Overview 38
Object Code Coverage 40
Evaluation 41
Example Script 46
Statement Coverage 47
Evaluation 47
Example Script 50
Full Decision Coverage 51
Evaluation Strategy 51
Evaluation 52
Example Script 57
Object Code Based (ocb) Decision Coverage 58
Evaluation Strategy 58
Evaluation 60
Example Script 64
Condition Coverage 65
Evaluation Strategy 65
Evaluation 67
Example Script 71
Modified Condition/Decision Coverage (MC/DC) 72
Evaluation Strategy 72
Evaluation 74
Example Script 78
Function Coverage 79
Evaluation Strategy 80
Example Script 83
Expert Usage 83
Call Coverage 84
Evaluation 85
Details on Callers and Calles 89
Example Script 90
Expert Usage 91

Assemble Multiple Test Runs 92
Save and Restore Code Coverage Measurement 92
Save and Restore Trace Recording 94

Comment your Results **96**

TRACE32 Coverage Report Utility **98**

Assembler-Only Functions and Code Coverage **100**
Object Code Coverage 100
Source Code Metrics 101

Data Coverage **103**
Trace Data Collection 103
Evaluation 104
Document the Results 107

Appendix A: Trace Decoding in Detail **108**
Trace Decoding for Static Applications 108
Decoding in Stopped State for Static Applications 108
Decoding in Running State for Static Applications 108
RTS Decoding for Static Applications 109
Trace Decoding for Applications Using a Rich OS 110
Decoding in Stopped State (Rich OS) 110
Decoding in Running State (Rich OS) 110
RTS Decoding (Rich OS) 110

Appendix B: Coding Guidelines **112**

Appendix C: Conditional Non-Branch Instructions **115**
Conditional Instructions 115

Appendix D: Object Code Coverage Tags in Detail **116**
Standard Tags 116
Tags for Arm-ETMv1/v3/v4 for Arm/Cortex Architecture 117

Appendix E: Data Coverage in Detail **119**

Application Note for Trace-Based Code Coverage

Version 09-Oct-2023

History

29-Jun-21	In “TRACE32 Coverage Report Utility” a note has been added to inform the customer that for report generation in larger projects it is advisable to make the trace decoding via the TRACE32 Virtual Memory.
29-Jun-21	As of build 166565, TRACE32 supports full decision coverage in RTS mode. The summary table in chapter “Choose the Appropriate Trace Data Collection Method” and “RTS Mode Code Coverage” were updated.
29-Jun-21	Standards IEC 61508 (industrial) and IEC 62304 (medical) added to chapters “Trace-based Code Coverage and Certification”.
24-Mar-21	Chapter “Details on Callers and Calles” added.
15-Mar-21	Chapter “Data Coverage” added.
09-Mar-21	“Appendix C: Object CodeCoverage in Detail”, and “Appendix D: Read/Write Coverage” added.
02-Mar-21	Chapter “TRACE32 Tool Configurations” added.
26-Feb-21	Chapter “Assembler Functions and Code Coverage” added.
18-Feb-21	Chapters “Trace-based Code Coverage and Certification”, and “SMP Multicore Systems” added.
19-Aug-20	New application note.

Introduction

Many embedded systems have to be developed according to some kind of internationally recognized safety standard. Part of the data required to prove that a system meets these standards is some form of code coverage. Safety standard will recommend or mandate various levels of code coverage that must be provided to meet certain tiers within that standard.

Many popular embedded devices include the option for chip level trace. These technologies vary by device and manufacturer but the data they produce is very similar: a non-intrusive trace of the flow of execution of a program running on that device. Analyzing this data for code coverage is the subject of this document.

Some devices only provide on-chip trace buffers for storing collected trace data; these are often very small and therefore unsuitable for code coverage. It is up to the user to determine whether an on-chip trace buffer can hold enough data for the required coverage reports.

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.

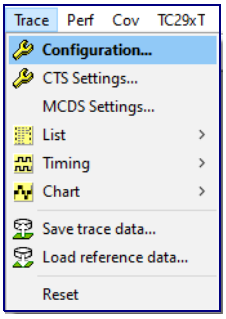
The manual is written in such a way that it is sufficient to only read the relevant chapters. If you read the manual completely, this may lead to redundancies.

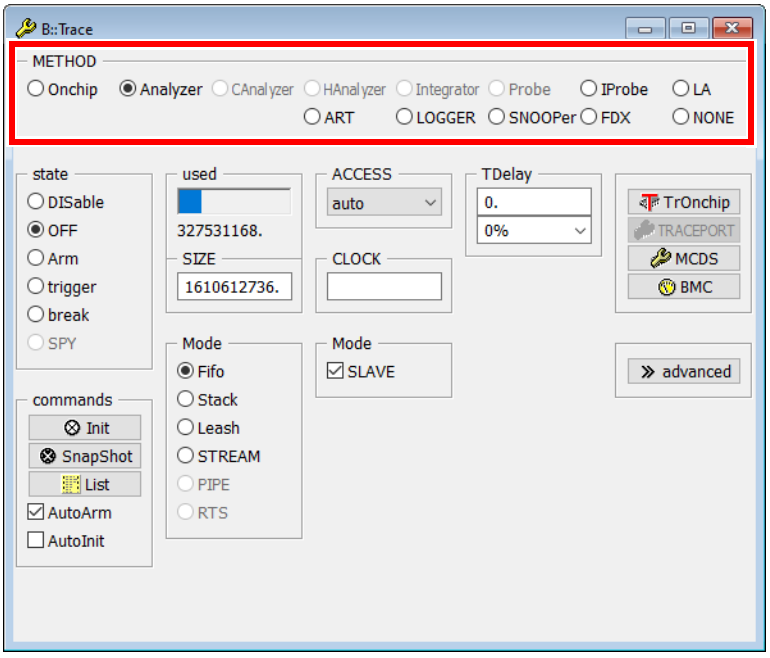
Prerequisites

It is assumed that the reader understands programming of embedded systems and is familiar with the safety/quality specification which has been chosen.

It is also assumed that TRACE32 has been correctly configured for the target and the symbols of the application under test have been loaded. The trace port has to be configured to provide trace data and TRACE32 must be configured to collect this data and have appropriate licenses to do so.

Opening the **Trace Configuration** window from the **Trace** menu shows the currently enabled **Trace.METHOD**. The TRACE32 software will grey-out any trace methods that are not available or not supported by the current TRACE32 configuration.





Trace-based code coverage can be performed for the following TRACE32 trace methods: **Analyzer**, **CAnalyzer**, **Onchip**, **ART**. All other methods are not suitable for code coverage.

Trace-Based Code Coverage and Certification

Code coverage measurement is a requirement for certification to evaluate the completeness of test cases and to demonstrate that there is no unintended functionality. TRACE32 supports all metrics from the following standards:

- **DO-178C (avionics):** statement coverage, decision coverage, MC/DC.
- **IEC 61508 (industrial):** statement coverage, branch coverage (decision coverage in TRACE32), condition coverage, MC/DC as well as function coverage.
- **IEC 62304 (medical):** Select suitable subset according to software development plan.
- **ISO 26262 (automotive):** statement coverage, branch coverage (decision coverage in TRACE32), MC/DC as well as function coverage and call coverage.

For those whose application requires tool qualification, Lauterbach offers a Tool Qualification Support Kit (TQSK in short). It provides everything needed to qualify a TRACE32 tool for use in safety-critical project.

TRACE32 Tool Configurations

The following TRACE32 tools are suitable for code coverage:

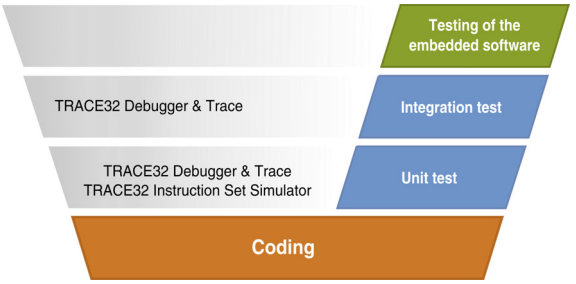
- **TRACE32 Debugger and Off-Chip Trace**
- **TRACE32 Debugger and On-Chip Trace**
- **TRACE32 Instruction Set Simulator**

  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 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 33.

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 Method

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 35.

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 Decision 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:

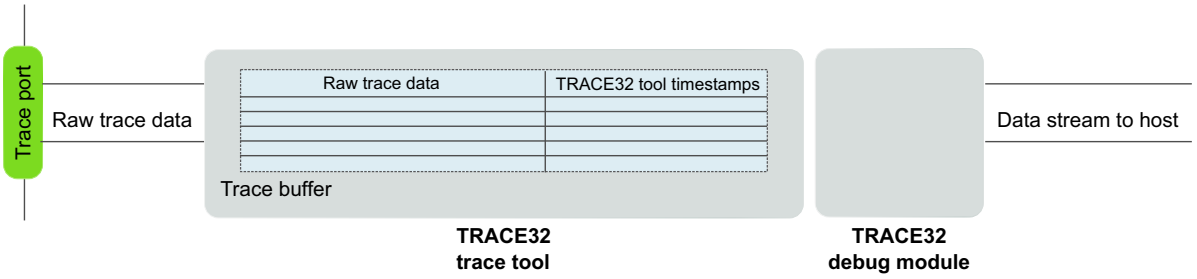
```
Go
Break
SILENT.Trace.Find FLOWERROR /ALL
IF FOUND.COUNT()!=0.
(
  PRIVATE &msg
  &msg="FLOWERRORS were found in the analyzed trace recording."
  &msg="&msg It is recommended to check"
  &msg="&msg if the trace recording works properly."
  ECHO FOUND.COUNT() "&msg"
)
ELSE
(
  ECHO "The analyzed trace 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.

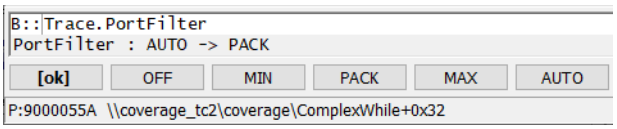
Disable Timestamps for Trace Streaming

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.

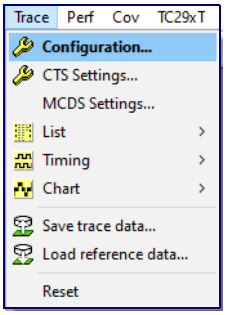
Incremental Code Coverage

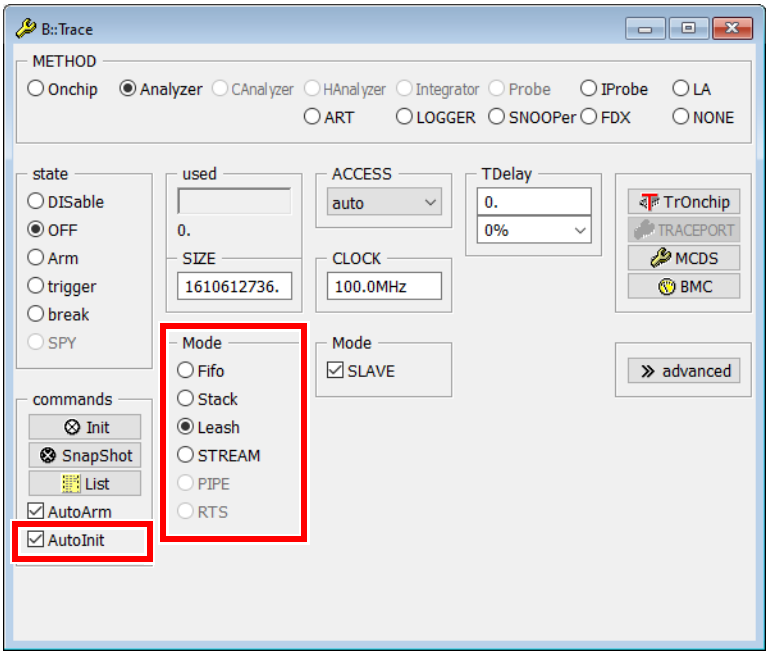
Incremental coverage is supported by all processor architectures which provide information about program flow that is saved to trace buffer and all TRACE32 configurations. It also supports all code coverage metrics supported by TRACE32. **It is a reliable fallback methods that can be used in the vast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

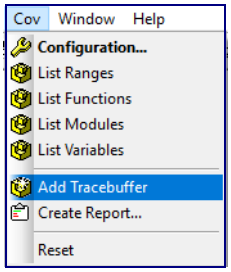
2. Enable the **AutoInit** checkbox or use the command **Trace.AutoInit ON** to ensure that the trace buffer is always cleared before the trace recording is started.

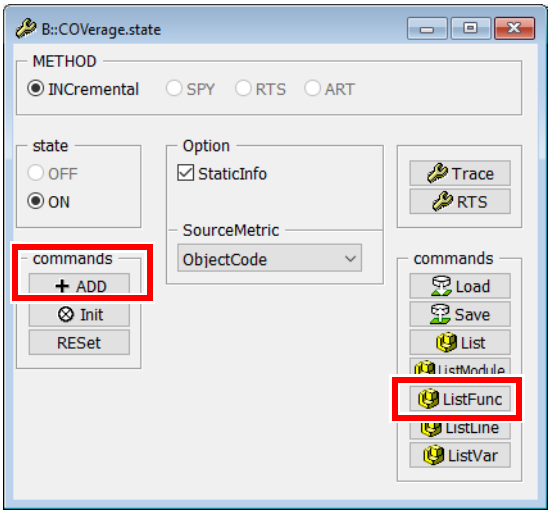




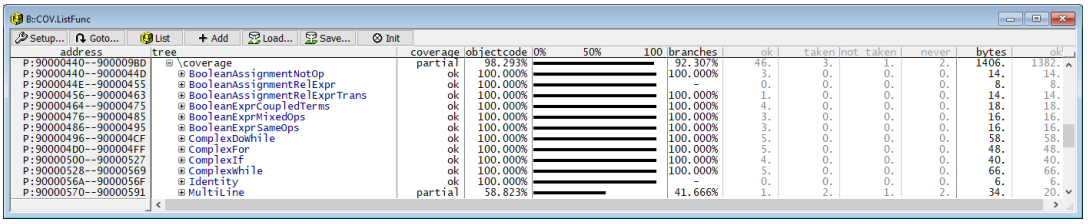
3. Start program execution and wait until it stops.

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 **"Supported Code Coverage Metrics"**, page 38.

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

Example Script

The entire process can be automated by creating a PRACTICE script. It is assumed that the preconditions listed in **"Preconditions"**, page 12 are satisfied before running the script. In the example script default settings are commented out.

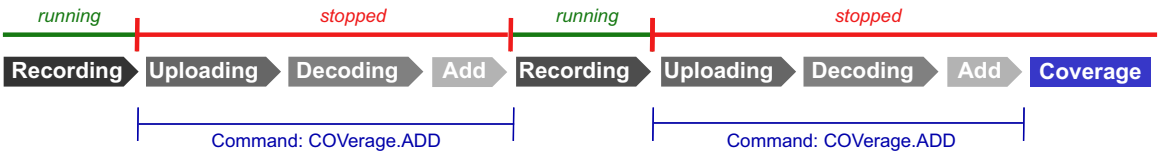
```
...
// Trace.METHOD as automatically selected by TRACE32
Trace.Mode Leash
// Trace.AutoArm ON
Trace.AutoInit ON
COVerage.RESet
// COVerage.METHOD INCremental
RePeaT 10.
(
    Go.direct
    WAIT !STATE.RUN()
    COVerage.ADD
)
COVerage.ListFunc
```

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 **"Supported Code Coverage Metrics"**, page 38.

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.

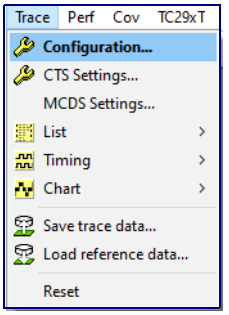
In case of large amounts of trace data, processing may take a long time. TRACE32 provides two alternative methods to avoid this situation.

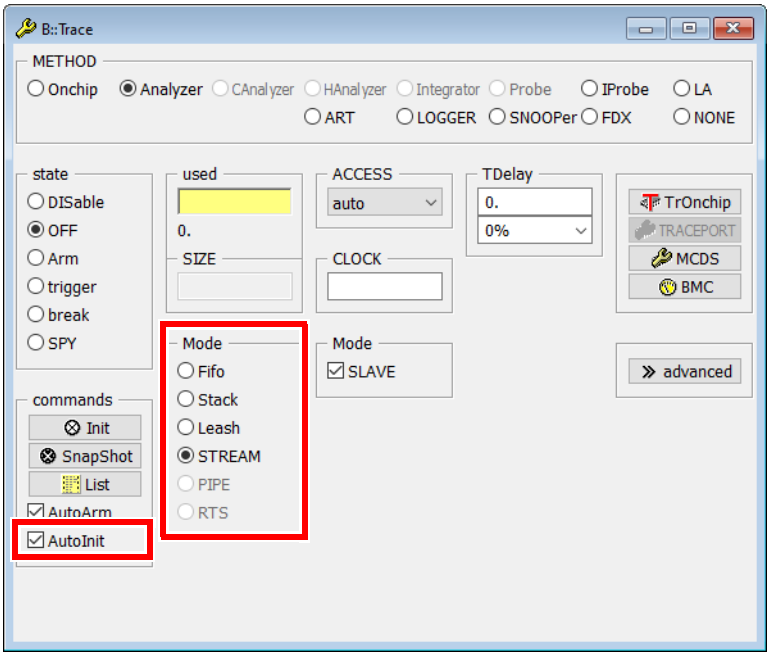
The first method is RTS, which is supported for all major architectures. RTS means that trace data is processed while being recorded and the code coverage results are displayed dynamically. Please see **"RTS Mode Code Coverage"**, page 22 for additional information.

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 27 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.

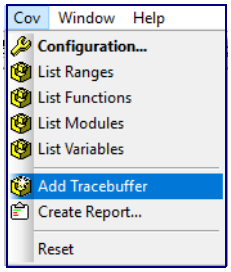
5. Since code coverage does not need any timestamp information, please use the command **Trace.PortFilter MAX** to instruct TRACE32 to stream only the raw trace data. Further background information can be found in the chapter **"Disable Timestamps for Trace Streaming"**, page 14.

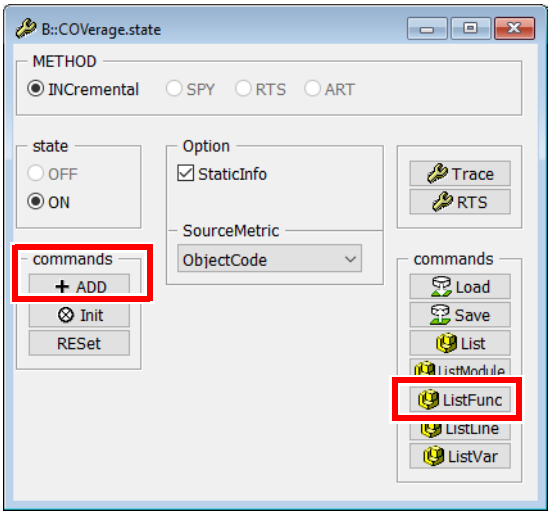
6. Start the program execution.

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

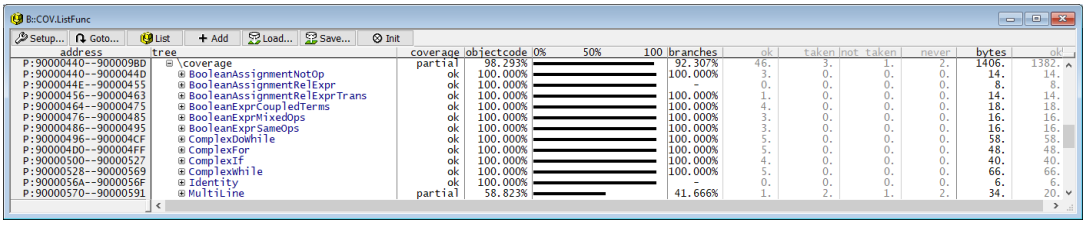
   - The user may manually stop the program execution.
   - A breakpoint may be used to stop the program execution.
   - With the help of a script, the program execution may be stopped after a specific period of time.

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 **"Supported Code Coverage Metrics"**, page 38.

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**.

Example Script

In this example script default settings are commented out. It is assumed that the preconditions listed in **"Preconditions"**, page 12 are satisfied before running the script.

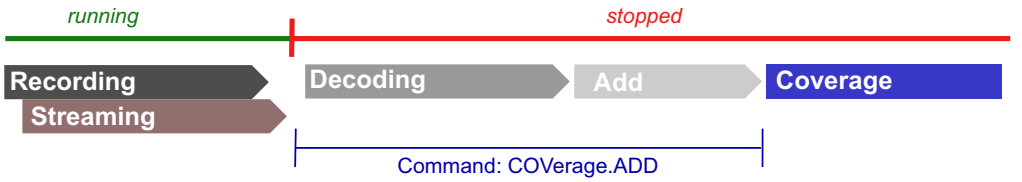
```
...
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer
// Trace.AutoArm ON
Trace.AutoInit ON

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
```

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 **"Supported Code Coverage Metrics"**, page 38.

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 27.

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 decision coverage.

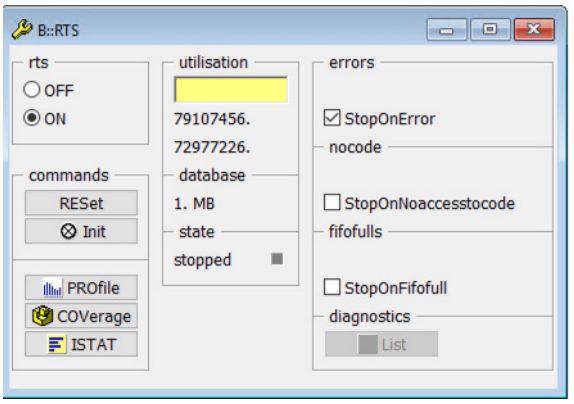
Data Collection

1. RTS mode code coverage requires RTS decoding.

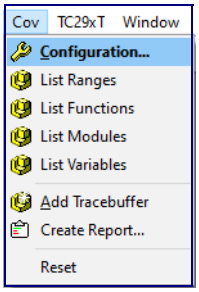
   Setup the RTS decoding by copying the object code to the **TRACE32 Virtual Memory**. For background information refer to **"RTS Decoding for Static Applications"**, page 109 or **"RTS Decoding (Rich OS)"**, page 110.

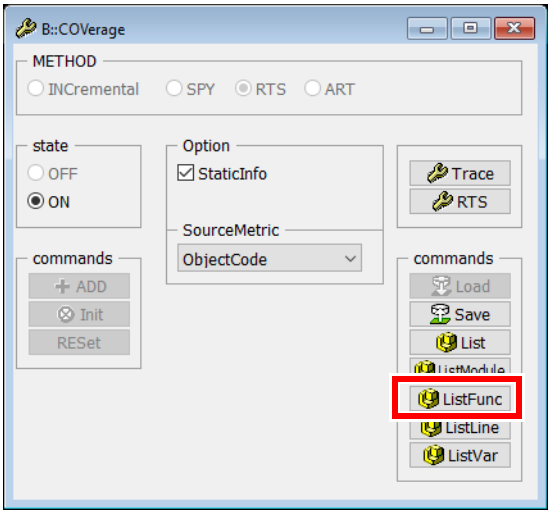
```
Data.LOAD.Elf ~~~~/tricore/coverage_tc2.elf /RelPATH /PlusVM
```

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

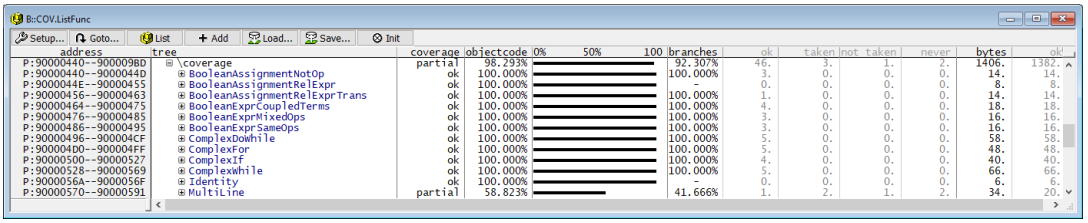


3. Open a **COVerage.ListFunc** window by using the **ListFunc** button in the **COVerage Configuration** window or by using the command **COVerage.ListFunc**. Please be aware that trace data recorded in RTS mode are only processed by TRACE32 as long as one window in TRACE32 displays code coverage information.





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



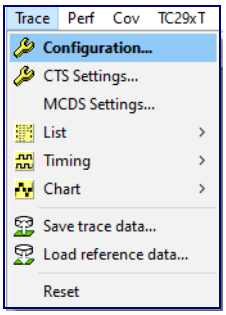
Details on the code coverage analysis itself are provided in the chapter **"Supported Code Coverage Metrics"**, page 38.

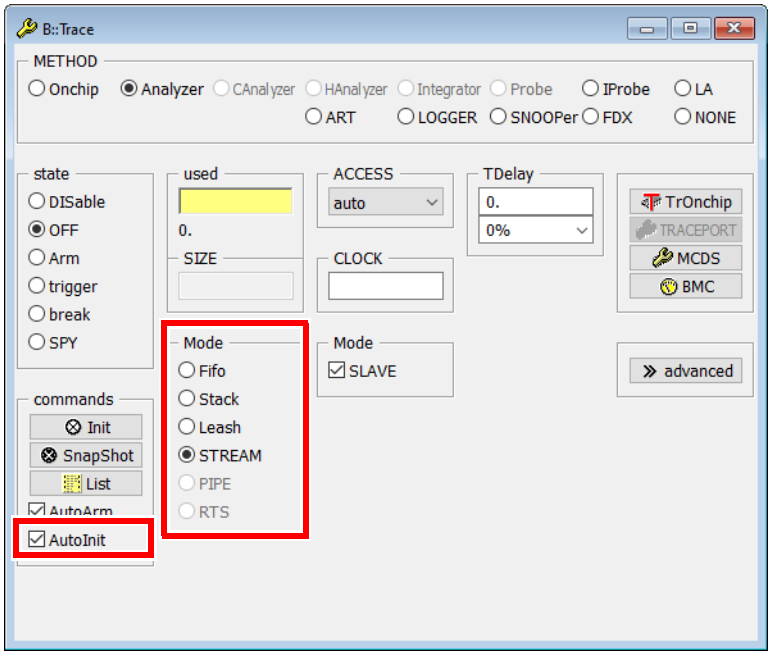
5. Stop the program exucution when your tests are completed.

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 14.

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 12 are satisfied before running the script.

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

; Load application code to target and TRACE32 Virtual Memory
Data.LOAD.Elf application.elf /PlusVM

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

COVerage.RESet
RTS.ON
COVerage.ListFunc

Go
WAIT !STATE.RUN()
...
```

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

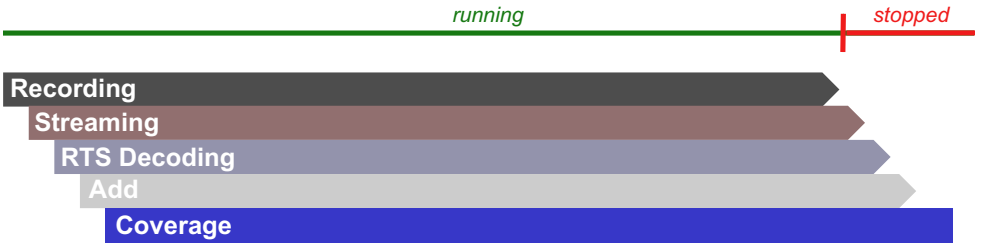
```
...
// Trace.METHOD Analyzer or Trace.METHOD CAnalyzer
// Trace.AutoArm ON
Trace.AutoInit ON

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

Trace.PortFilter MAX


; Load application code to target and TRACE32 Virtual Memory
Data.LOAD.Elf application.elf /PlusVM

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

COVerage.RESet
RTS.ON
COVerage.ListFunc

Go
WAIT !STATE.RUN()
Trace.List
...
```

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 file6
- 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 **"Supported Code Coverage Metrics"**, page 38.

SPY Mode Code Coverage

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

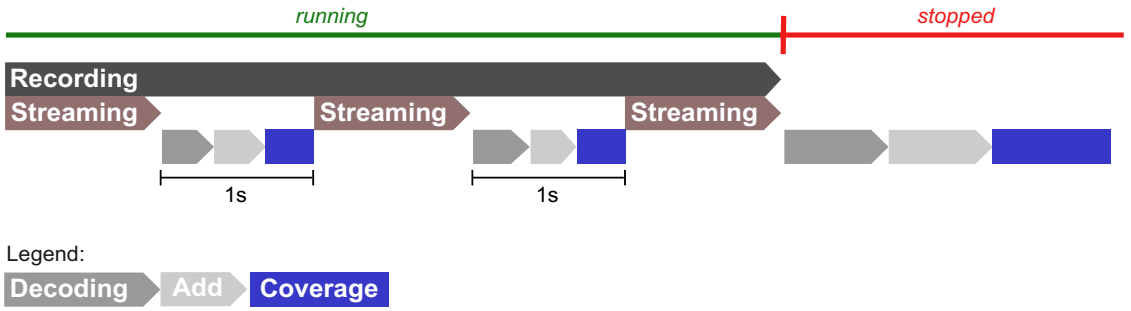
- TRACE32 trace hardware tool such as PowerTrace is required
- Streaming of the trace data to a file on the host file system is working without issues

  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.

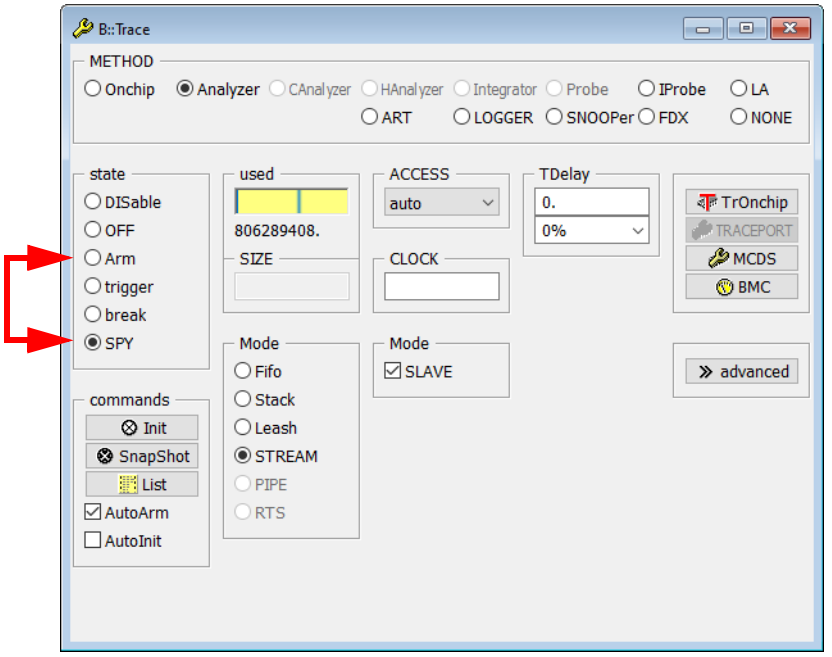
Operation States

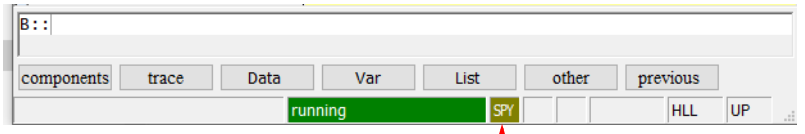
For SPY mode code coverage, trace streaming is periodically suspended in order to decode the raw trace data and to process it for code coverage. Please be aware that TRACE32 does not suspend trace streaming if the trace memory of the TRACE32 trace tool, that operates as a large FIFO, is filled more the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

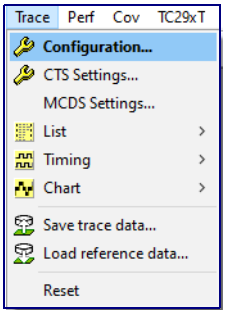
1. In order to decode the raw trace data quickly, it is recommended to mirror the application to the TRACE32 Virtual Memory:

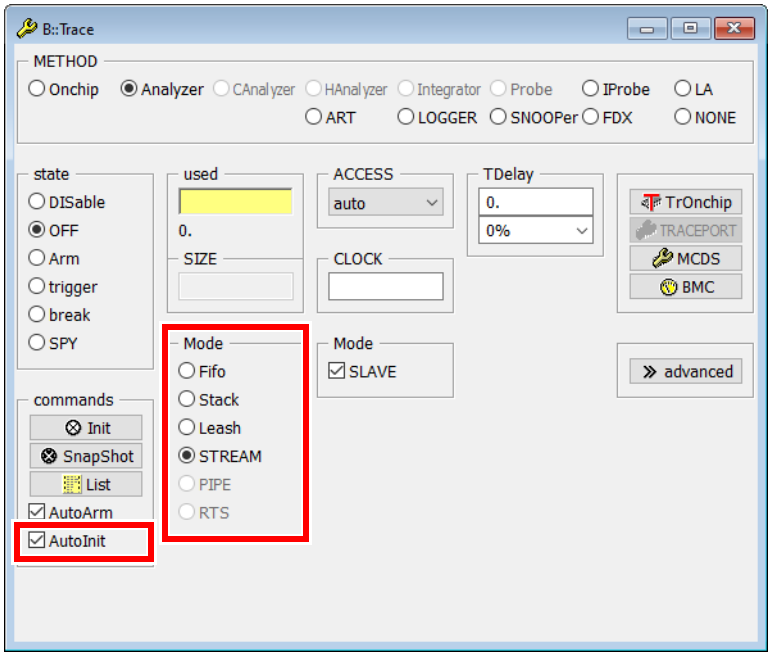
```
Data.LOAD.Elf application_1.elf /PlusVM
```

   For details refer to **"Decoding in Running State for Static Applications"**, page 108 or **"Decoding in Running State (Rich OS)"**, page 110.

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

3. 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.





4. 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"
```

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

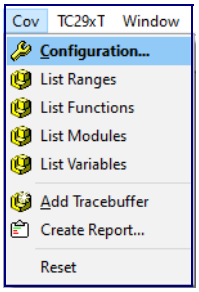
```
; limit the size of the streaming file to 5 GBytes
Trace.STREAMFileLimit 5000000000.
```

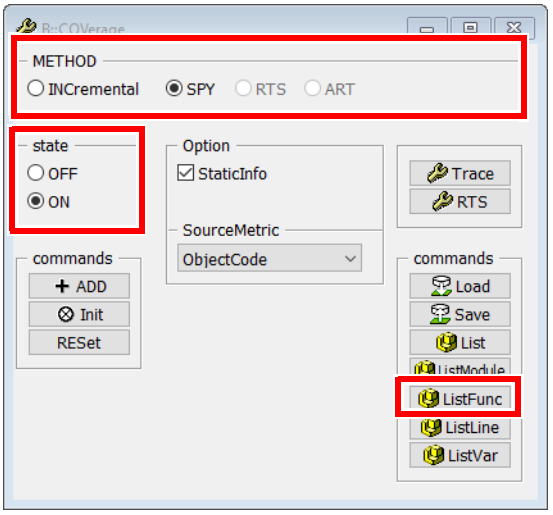
   Please be aware, that the trace recording is stopped, when the size limit for the streaming file is reached.

6. 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 14.

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

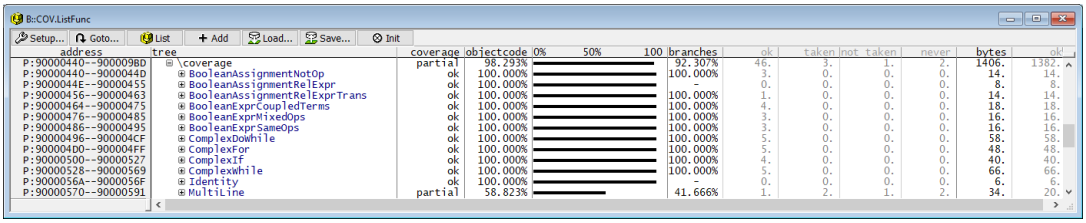
8. Enable **SPY** mode code coverage by the command **COVerage.ON** or by selecting the **ON** radio button in the state field.





9. 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.

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



address	tree	coverage	objectcode	0% 50% 100	branches	ok	taken	not_taken	never	bytes	ok
P:90000440--900009BD	\coverage	partial	98.293%		92.307%	46.	3.	1.	2.	1406.	1382.
P:90000440--9000044D	BooleanAssignmentNotOp	ok	100.000%		100.000%	3.	0.	0.	0.	14.	14.
P:9000044E--90000455	BooleanAssignmentRelExpr	ok	100.000%		-	0.	0.	0.	0.	8.	8.
P:90000456--90000463	BooleanAssignmentRelExprTrans	ok	100.000%		100.000%	1.	0.	0.	0.	14.	14.
P:90000464--90000475	BooleanExprCoupledTerms	ok	100.000%		100.000%	4.	0.	0.	0.	18.	18.
P:90000476--90000485	BooleanExprMixedOps	ok	100.000%		100.000%	3.	0.	0.	0.	16.	16.
P:90000486--90000495	BooleanExprSameOps	ok	100.000%		100.000%	3.	0.	0.	0.	16.	16.
P:90000496--900004CF	ComplexDoWhile	ok	100.000%		100.000%	5.	0.	0.	0.	58.	58.
P:900004D0--900004FF	ComplexFor	ok	100.000%		100.000%	5.	0.	0.	0.	48.	48.
P:90000500--90000527	ComplexIf	ok	100.000%		100.000%	4.	0.	0.	0.	40.	40.
P:90000528--90000569	ComplexWhile	ok	100.000%		100.000%	5.	0.	0.	0.	66.	66.
P:9000056A--9000056F	Identity	ok	100.000%		-	0.	0.	0.	0.	6.	6.
P:90000570--90000591	MultiLine	partial	58.823%		41.666%	1.	2.	1.	2.	34.	20.

Details on the code coverage analysis itself are provided in the chapter **“Supported Code Coverage Metrics”**, page 38.

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

Example Script

In the script the default settings are commented out. It is assumed that the preconditions listed in **“Preconditions”**, page 12 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


; Load application code to target and TRACE32 Virtual Memory
Data.LOAD.Elf application.elf /PlusVM

; 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
...
```

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.

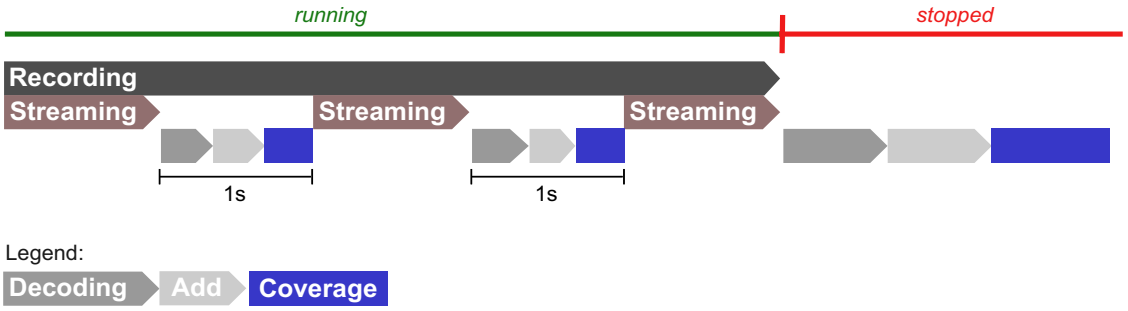
The following steps are involved:

- 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 **"Supported Code Coverage Metrics"**, page 38.

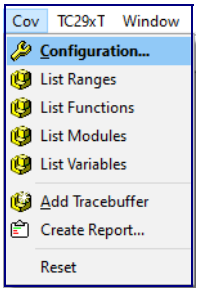
Code Coverage with Virtual Targets

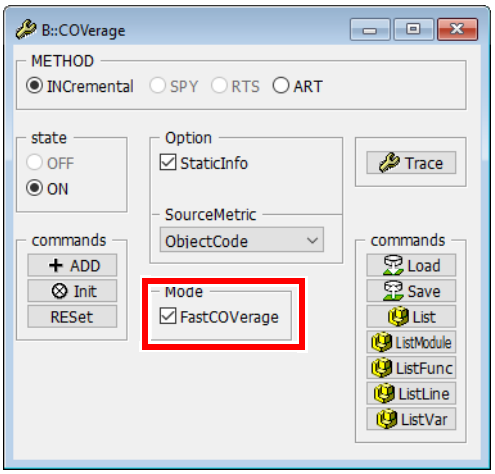
Tracing the program execution on a virtual target slows down its performance. To minimize this impact, Lauterbach works closely together with manufacturers such as Synopsys. The basic idea is that some parts of the code coverage processing are offloaded to the virtual target. This information is uploaded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

To use this feature the following conditions must be met:

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

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





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

- Object code coverage (see **“Object Code Coverage”**, page 40)
- Statement coverage (see **“Statement Coverage”**, page 47)
- Decision coverage (ocb) (see **“Object Code Based (ocb) Decision Coverage”**, page 58)
- Function coverage (see **“Function Coverage”**, page 79)

An example script might look like this:

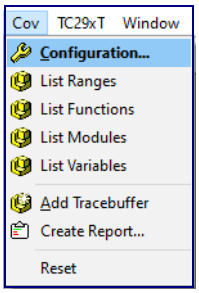
```
COVerage.RESet
COVerage.METHOD INCremental
COVerage.Mode FastCOVerage ON

Go

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

Break

COVerage.Add

COVerage.ListFunc
```

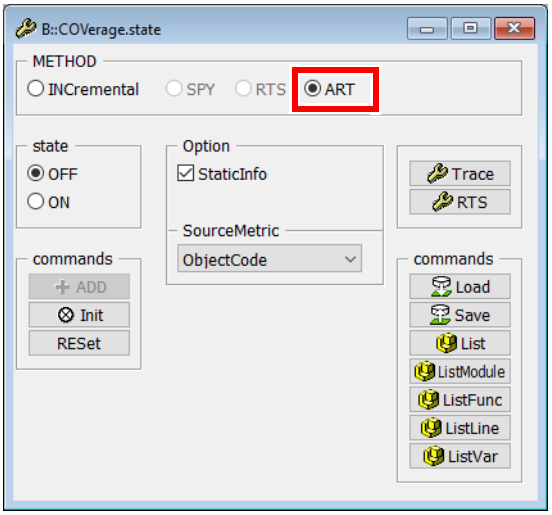
Details about the code coverage analysis itself are provided in the chapter **“Supported Code Coverage Metrics”**, page 38.

ART Mode Code Coverage

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

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





Trace data recorded with ART can be used for:

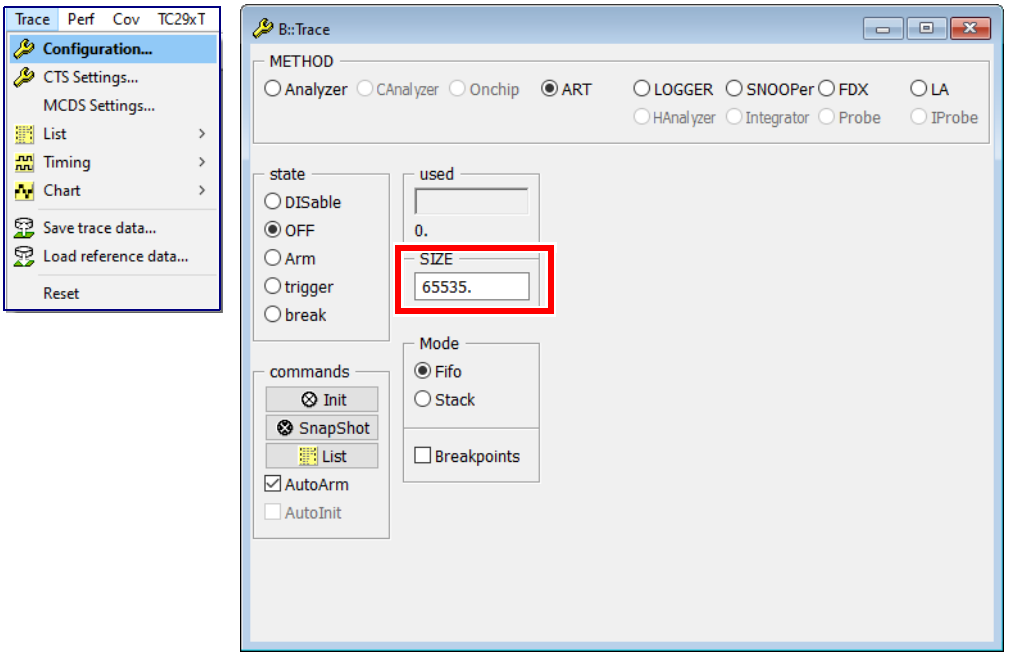
- Object code coverage (see **"Object Code Coverage"**, page 40)
- Statement coverage (see **"Statement Coverage"**, page 47)
- Decision coverage (ocb) (see **"Object Code Based (ocb) Decision Coverage"**, page 58)
- Function coverage (see **"Function Coverage"**, page 79)

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.

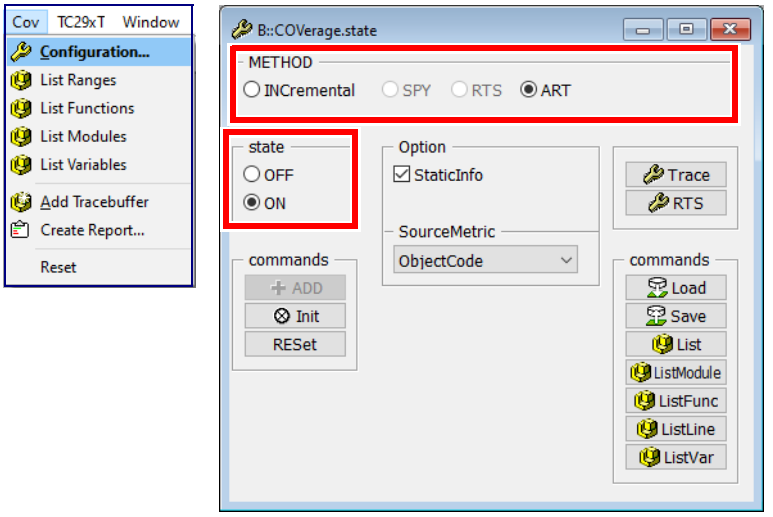
The TRACE32 Instruction Set Simulator simulates the instruction set, but does not model timing characteristics and peripherals. However, the simulator provides a bus trace so that code coverage is easy to perform. For details on how to start the TRACE32 Instruction Set Simulator refer to **"TRACE32 Instruction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

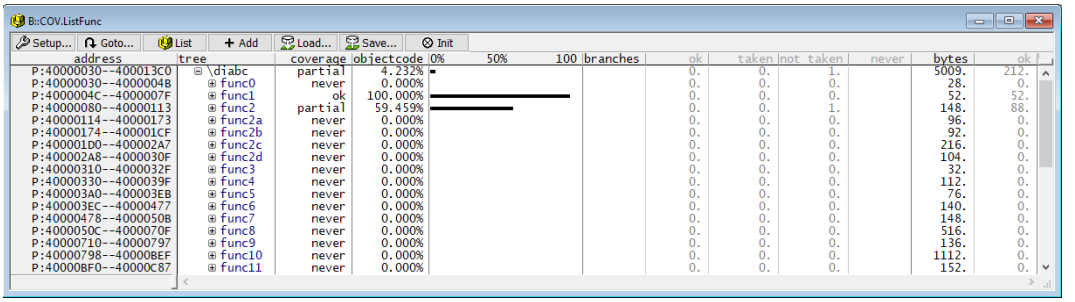
1. Select **Trace.METHOD ART** in the **Trace configuration** window.
2. Set the size of the ART buffer, using either the command **ART.SIZE** *<n>* or by entering the value in the **SIZE** field of the **Trace configuration** window.



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.



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

Details about the code coverage analysis itself are provided in the chapter **"Supported Code Coverage Metrics"**, page 38.

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
```

Supported Code Coverage Metrics

Overview

TRACE32 supports all important code coverage metrics. The table below gives an overview of the following subjects:

- **A definition for every TRACE32 source metric**
- **Requirements for the object code**

  It is recommended to perform the code coverage on non-optimized code. This way the results can be displayed clearly and concisely. Otherwise the interpretation of the result becomes more demanding.

  For decision and condition coverage, as well as for MC/DC, conditions must be implemented at object code level by conditional branches or conditional instructions. Conditional instructions are only sufficient if the trace protocol in use generates details for them.
- **Source code details that TRACE32 needs for the measurement**

  The required source code details are not part of the debug symbol information generated by the compiler, but must be generated separately. TRACE32 provides the command line tool t32cast for this purpose. For complete information about t32cast, see **"Application Note for t32cast"** (app_t32cast.pdf).

  Please note that **"RTS Mode Code Coverage"**, page 22 is currently not possible for all metrics that require additional source code details.

TRACE32 SourceMetric	Requirements for the object code	Source code details needed by TRACE32
ObjectCode 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.	Final code	—
Statement 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.	Final code	—
Decision (full) 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 the source code has to be represented by a conditional branch/instruction at object code level	TRACE32 has to know which source code lines contain a decision and how the individual decisions are structured

TRACE32 SourceMetric	Requirements for the object code	Source code details needed by TRACE32
Decision (ocb) 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.	Requires appropriate optimization level to prevent false-positive or false-negative results	—
Condition All conditions in the program have evaluated both true and false.	Each condition in the source code has to be represented by a conditional branch/instruction at object code level	TRACE32 has to know which source code lines contain a decision and how the individual decisions are structured
MCDC 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.	Each condition in the source code has to be represented by a conditional branch/instruction at object code level	TRACE32 has to know which source code lines contain a decision and how the individual decisions are structured
Function Every function in the program has been invoked at least once.	Final code Inlined functions make the interpretation of the results more demanding	—
Call Every function call has been executed at least once.	Final code Inlined functions make the interpretation of the results more demanding	TRACE32 must know which source code lines contain function calls

Object Code Coverage

Object code coverage can be performed directly on the final code.

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**

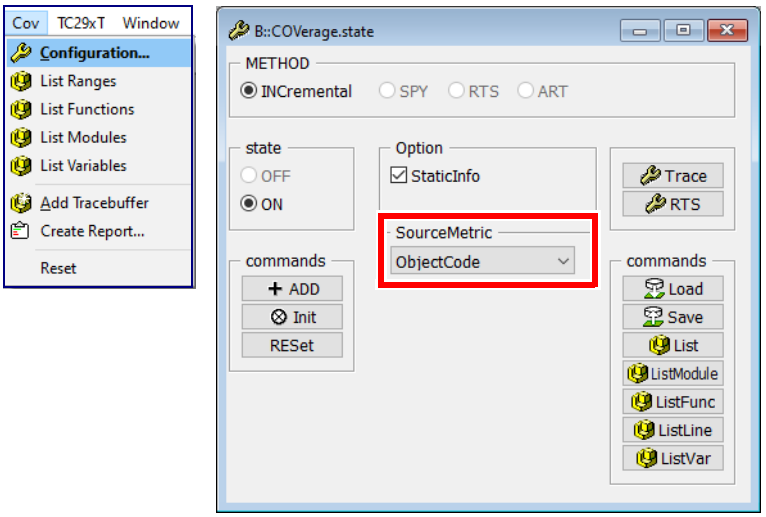
  For Arm/Cortex cores that use the protocols Arm-ETMv1 or Arm-ETMv3, as well as Arm-ETMv4 with **ETM.COND ON**.

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

  Otherwise.

For details refer to **"Appendix D: Object Code Coverage Tags in Detail"**, page 116.

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
```

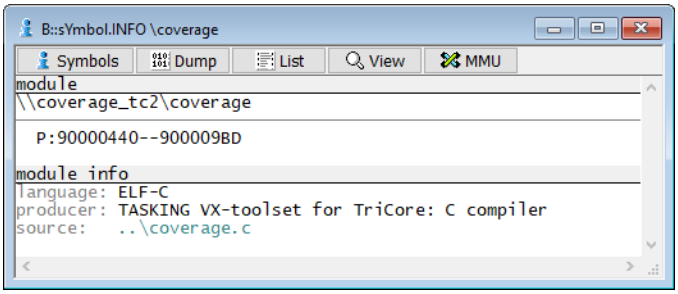
```
B::List.Mix MultiLine /COVerage
Step  Over  Diverge  Return  Up  Go  Break  Mode  Find:  coverage.c
true  false   coverage    addr/line  code      label     mnemonic          comment

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

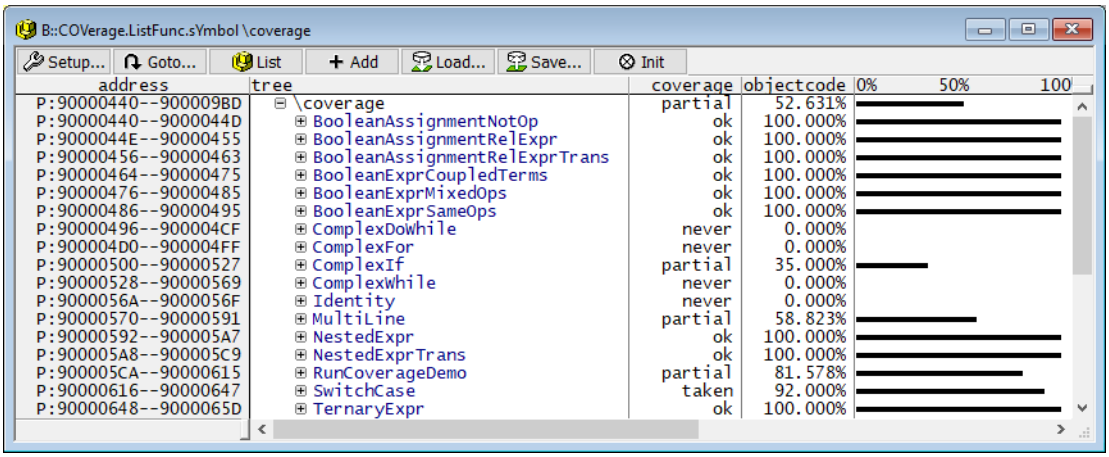
The screenshot on the previous page was taken with the Infineon TriCore™ debugger. Its instruction set contains no conditional instructions beyond conditional branches. Thus the object code is tagged as follows:

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

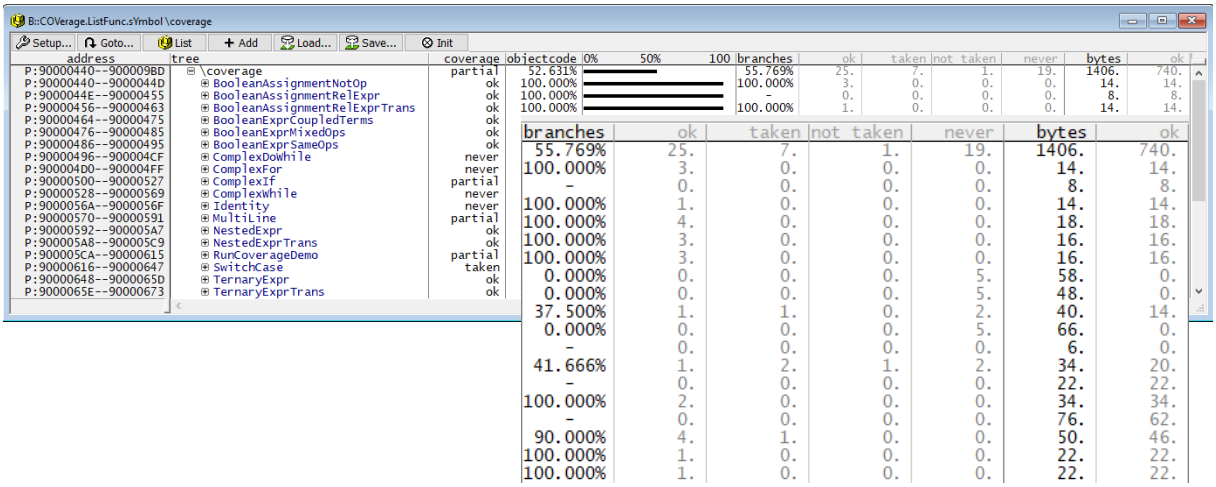
This TRACE32 command displays a tabular analysis of all functions of the module "coverage". A module usually corresponds to a source code file.

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





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

Statement coverage can be performed directly on the final code.

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**.**

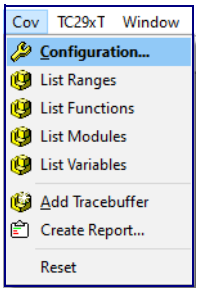
TRACE32 interpretation: A source code line achieves statement coverage when at least one corresponding object code instruction has been executed.

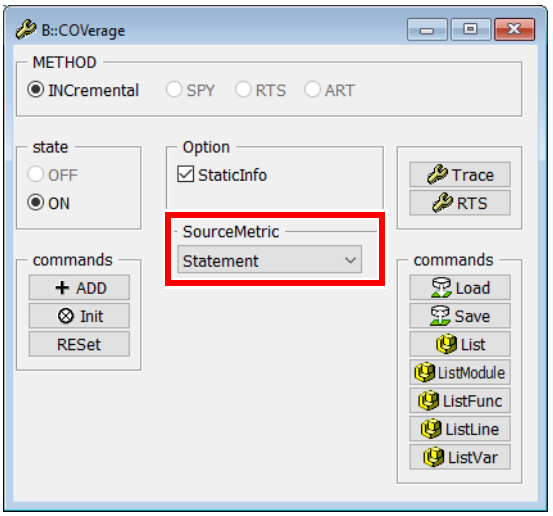
The following tagging is performed:

- **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:

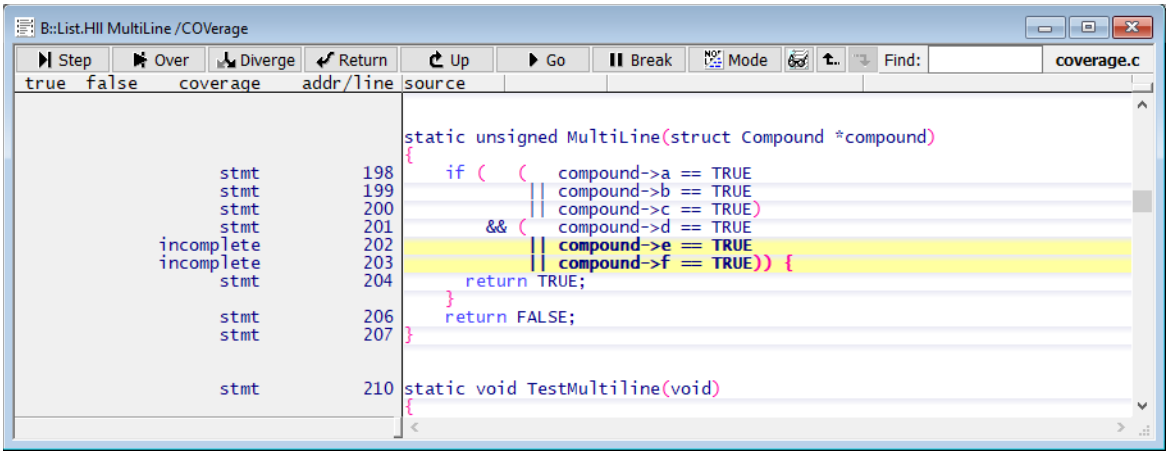
```
COVerage.ListModule
COVerage.ListFunc
```

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
```



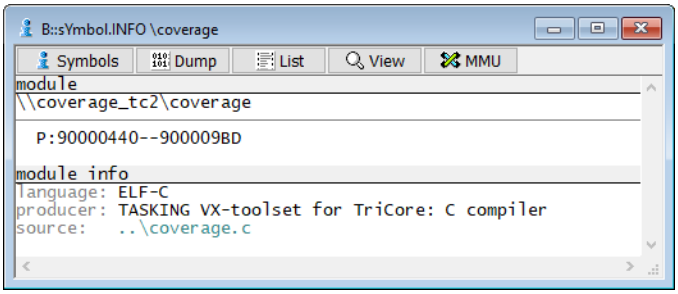
The source code lines are tagged as follows:

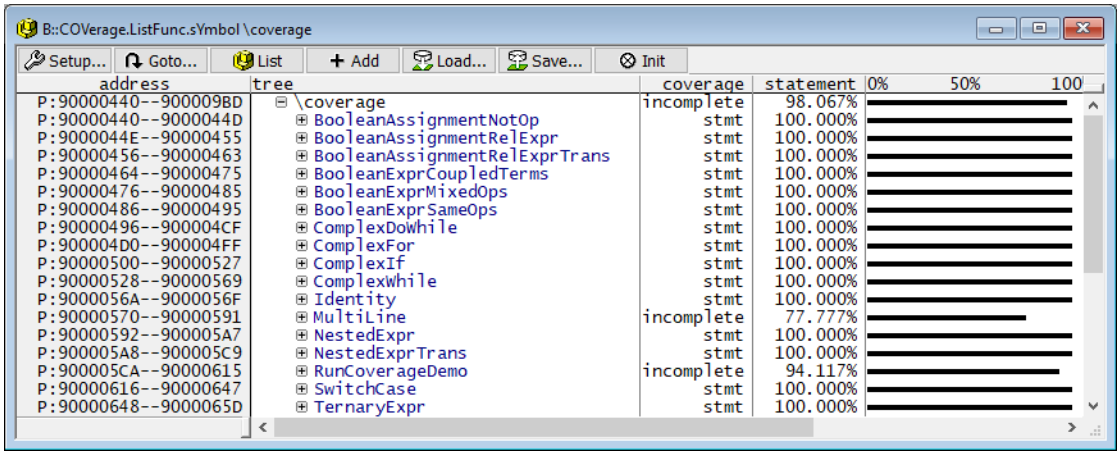
stmt	At least one corresponding object code instruction generated for the block of source code lines has been executed.
incomplete	None of the object code instructions generated for the block of source code lines has been executed.

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

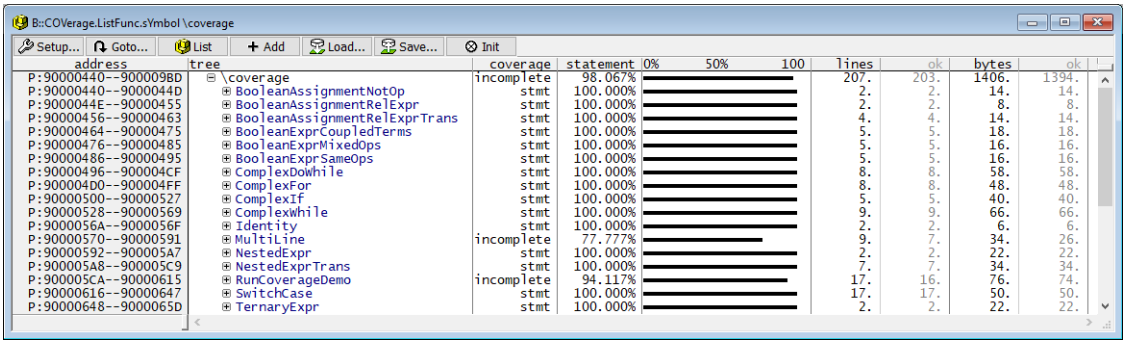
Statement coverage is achieved for a group of **HLL source code statements** as soon as one of its associated assembly instructions has been partially executed.

- **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:



address	tree	coverage	statement	0% 50% 100	lines	ok	bytes	ok
P:90000440--900009BD	⊟ \coverage	incomplete	98.067%		207.	203.	1406.	1394.
P:90000440--9000044D	⊞ BooleanAssignmentNotOp	stmt	100.000%		2.	2.	14.	14.
P:9000044E--90000455	⊞ BooleanAssignmentRelExpr	stmt	100.000%		2.	2.	8.	8.
P:90000456--90000463	⊞ BooleanAssignmentRelExprTrans	stmt	100.000%		4.	4.	14.	14.
P:90000464--90000475	⊞ BooleanExprCoupledTerms	stmt	100.000%		5.	5.	18.	18.
P:90000476--90000485	⊞ BooleanExprMixedOps	stmt	100.000%		5.	5.	16.	16.
P:90000486--90000495	⊞ BooleanExprSameOps	stmt	100.000%		5.	5.	16.	16.
P:90000496--900004CF	⊞ ComplexDoWhile	stmt	100.000%		8.	8.	58.	58.
P:900004D0--900004FF	⊞ ComplexFor	stmt	100.000%		8.	8.	48.	48.
P:90000500--90000527	⊞ ComplexIf	stmt	100.000%		5.	5.	40.	40.
P:90000528--90000569	⊞ ComplexWhile	stmt	100.000%		9.	9.	66.	66.
P:9000056A--9000056F	⊞ Identity	stmt	100.000%		2.	2.	6.	6.
P:90000570--90000591	⊞ MultiLine	incomplete	77.777%		9.	7.	34.	26.
P:90000592--900005A7	⊞ NestedExpr	stmt	100.000%		2.	2.	22.	22.
P:900005A8--900005C9	⊞ NestedExprTrans	stmt	100.000%		7.	7.	34.	34.
P:900005CA--90000615	⊞ RunCoverageDemo	incomplete	94.117%		17.	16.	76.	74.
P:90000616--90000647	⊞ SwitchCase	stmt	100.000%		17.	17.	50.	50.
P:90000648--9000065D	⊞ TernaryExpr	stmt	100.000%		2.	2.	22.	22.

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

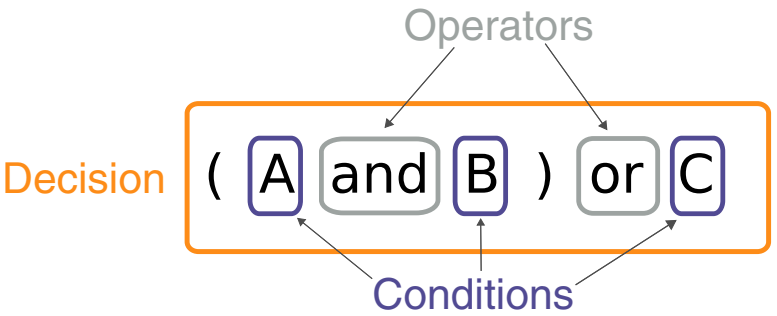
Byte count	
bytes	Number of bytes
ok	Number of bytes tagged with stmt

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

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"**, page 58)

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 all possible outcomes at least once.

To measure decision coverage accurately the following prerequisites must be fulfilled:

1. It is necessary that the code is compiled so that each condition in the source code is represented by a distinct conditional branch/instruction at object code level. Conditional instructions, however, require that the trace protocol includes information about conditional instructions.

   Please read **"Appendix B: Coding Guidelines"**, page 112 to ensure that you write decisions and conditions at source code level in such a way that your build toolchain generates conditional branches/instructions for them.

   Ensure that the compiler generates conditional branches for switch-case statements. A dedicated compiler option is commonly available to control this. Please refer to the documentation of your build toolchain.

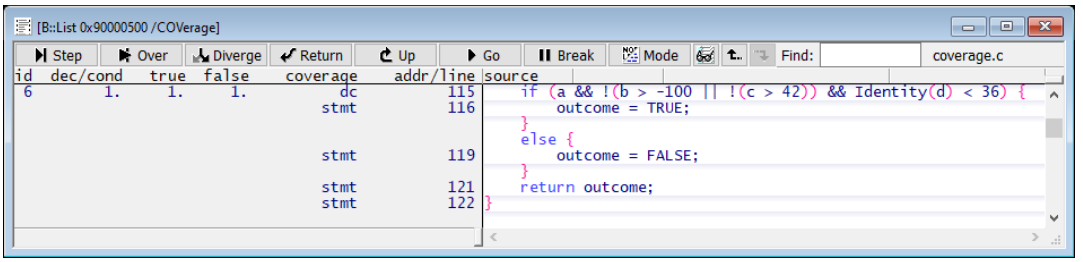
2. TRACE32 has to know which source code lines contain decisions and their conditions. Moreover, for each condition used in a decision its structure and the mapping between conditions and conditional branches/instructions must be known.

   These details are not part of the debug symbol information generated by the compiler, but must be generated separately. TRACE32 provides the command line tool t32cast for this purpose. For complete information about t32cast, see **"Application Note for t32cast"** (app_t32cast.pdf).

   The t32cast command line tool generates an Extended Code Analysis (ECA) data file for each source code file. These files have to be loaded into TRACE32 before starting the code coverage analysis.

If these prerequisites are met, full decision coverage can be performed with the optimal number of test cases.

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.

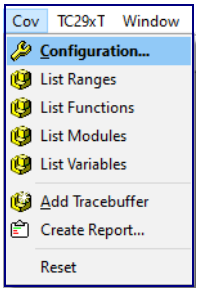
Source code lines that represent decisions are tagged as follows:

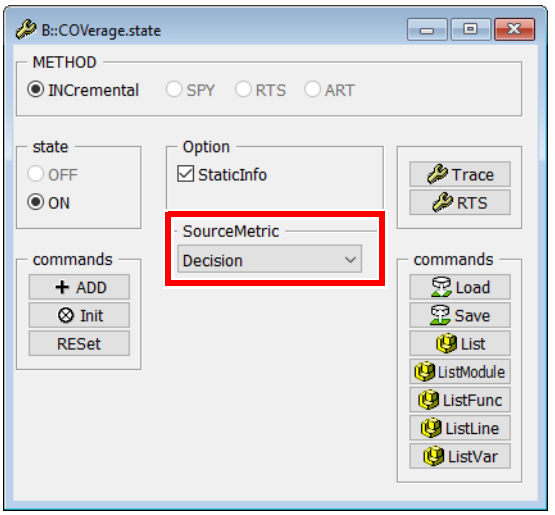
- **dc | incomplete**

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

Evaluation

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.





Before you start the code coverage analysis, you have to load the .eca files created by the command line tool t32cast:

```
sYmbol.ECA.LOADALL /SkipErrors
```

The following commands show a tabular analysis:

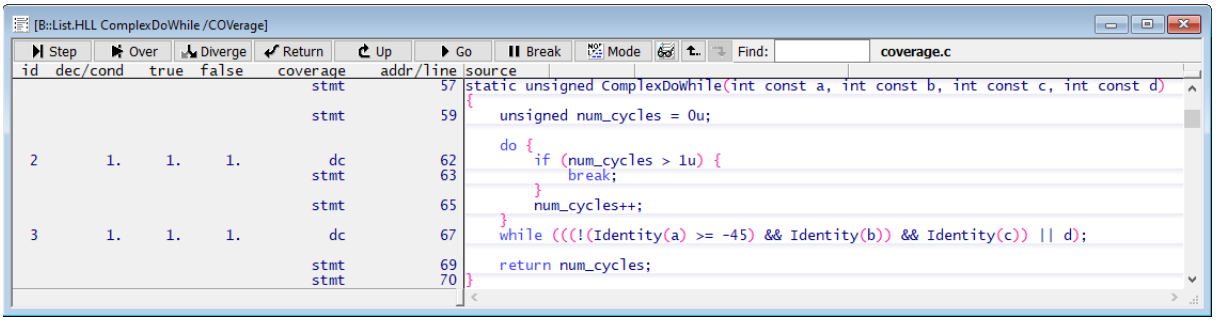
```
COVerage.ListModule
COVerage.ListFunc
```

The following command shows the tagging on source code level.

```
List.HII /COVerage
```

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

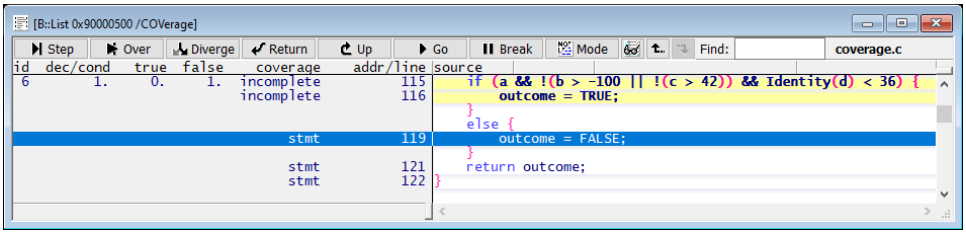
```
List.HLL ComplexDoWhile /COVerage
```



Decisions are tagged as follows:

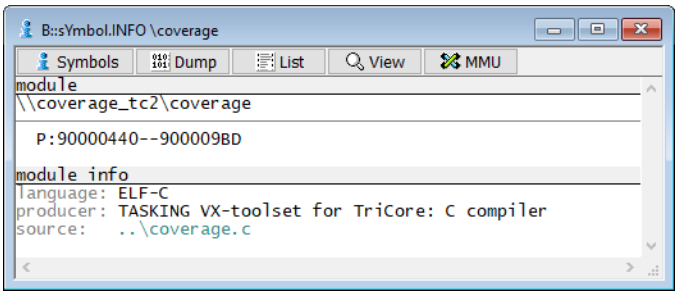
dc	Decisions have taken all possible outcomes at least once.
incomplete	There is at least one possible outcome missing for the decisions.

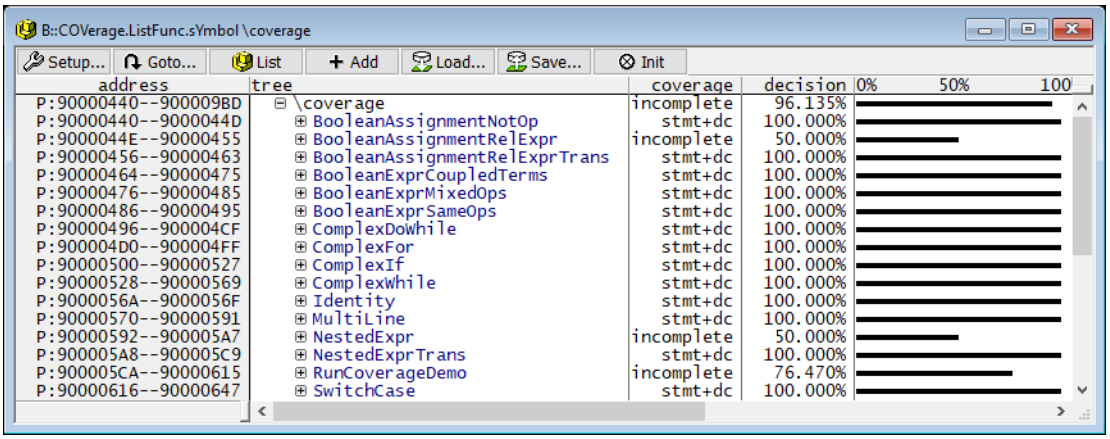
Not executed decision paths are tagged with incomplete at source code level. Already taken decision paths are tagged with stmt.



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

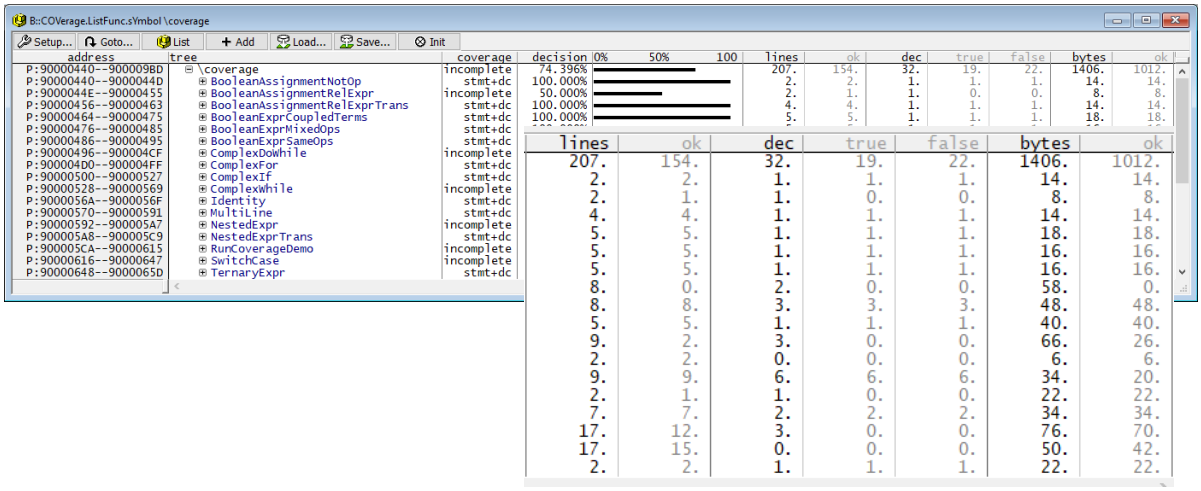
Decision coverage is achieved for a group of **HLL source code statements** as soon as all of its associated assembly instructions have been fully covered.

- **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

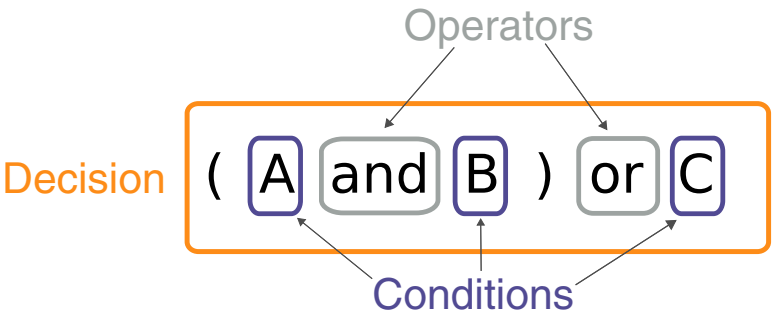
Byte count	
bytes	Number of bytes within the function/module
ok	Number of bytes tagged with dc or stmt

Example Script

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

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

// 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
```

Object Code Based (ocb) Decision 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”**, page 51) 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.

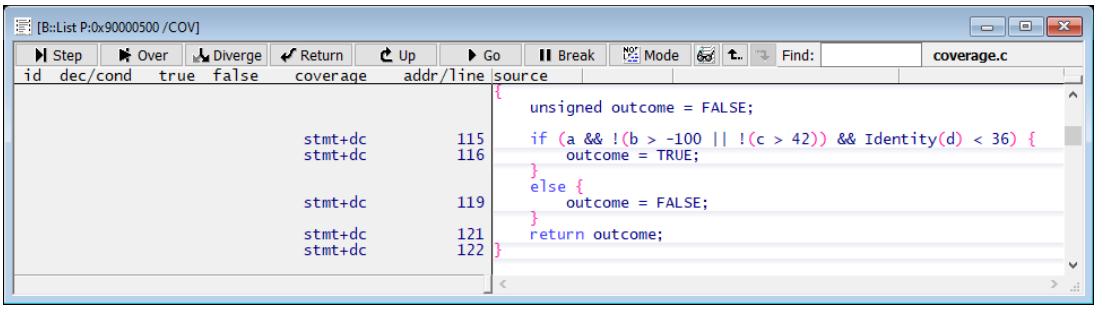
This eliminates the prerequisites necessary for full decision coverage. However, the following should be considered:

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

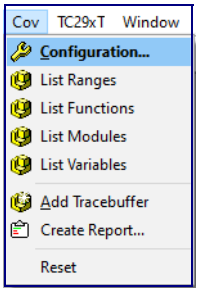
Optimized code can lead to false positive results if a condition is no longer represented by a conditional branch/instruction or the trace protocol provides no information about the state of conditional instructions. False positive means that decision coverage is indicated too early.

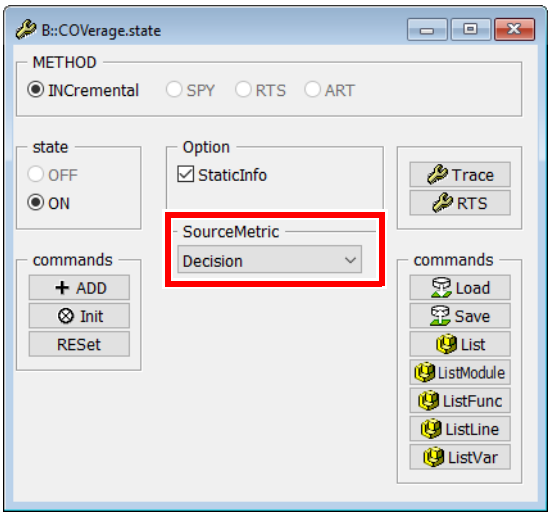
Since the source code is not analyzed for ocb decision coverage, TRACE32 does not know where decisions are located. Therefor source code lines are tagged as follows:

- **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

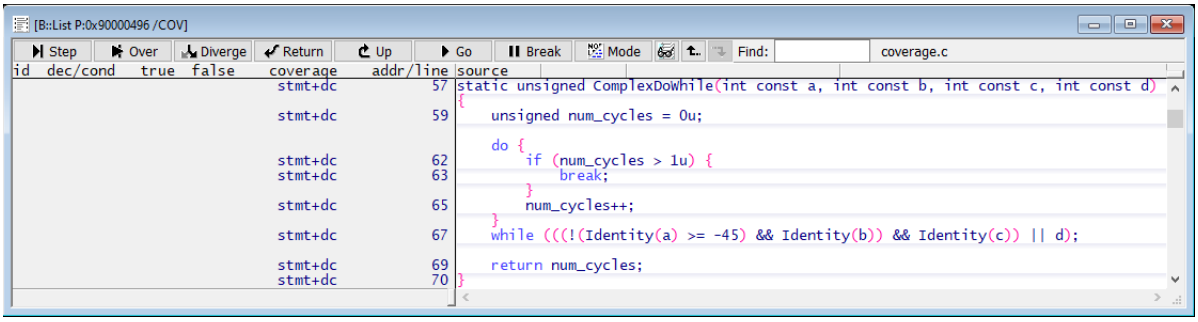
COVerage.ListFunc

The following command shows the tagging on source code level.

List.HII /COVerage

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

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



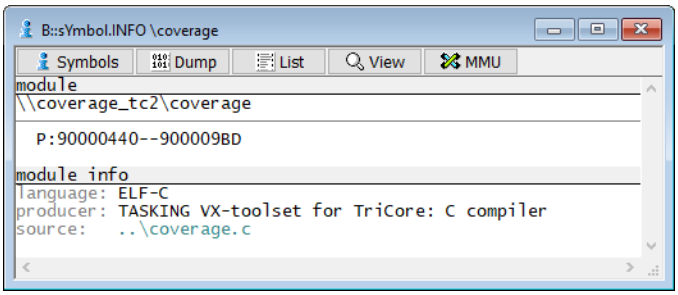
Source code lines are tagged as follows:

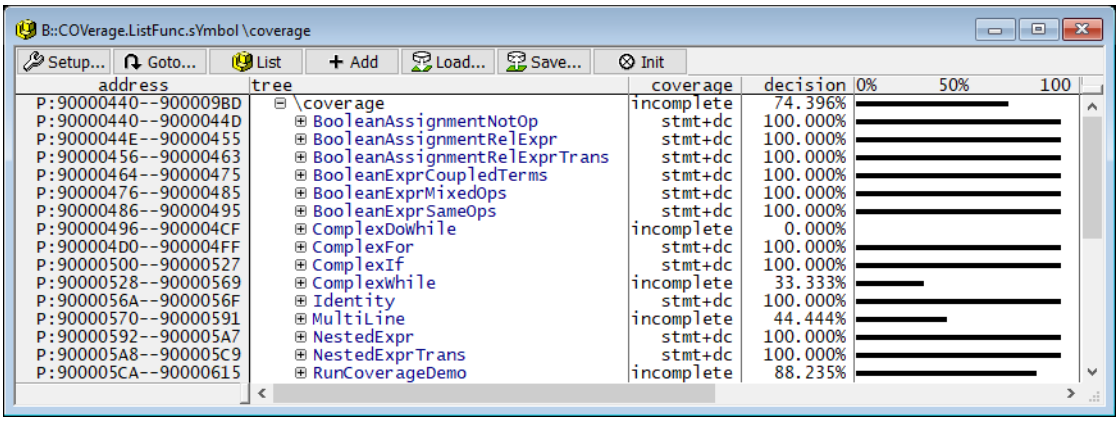
dc+stmt	The source code line achieved full object code coverage and thereby either decision or statement coverage.
incomplete	The source code line did not achieve full object code coverage and thereby no decision or statement coverage.

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
```

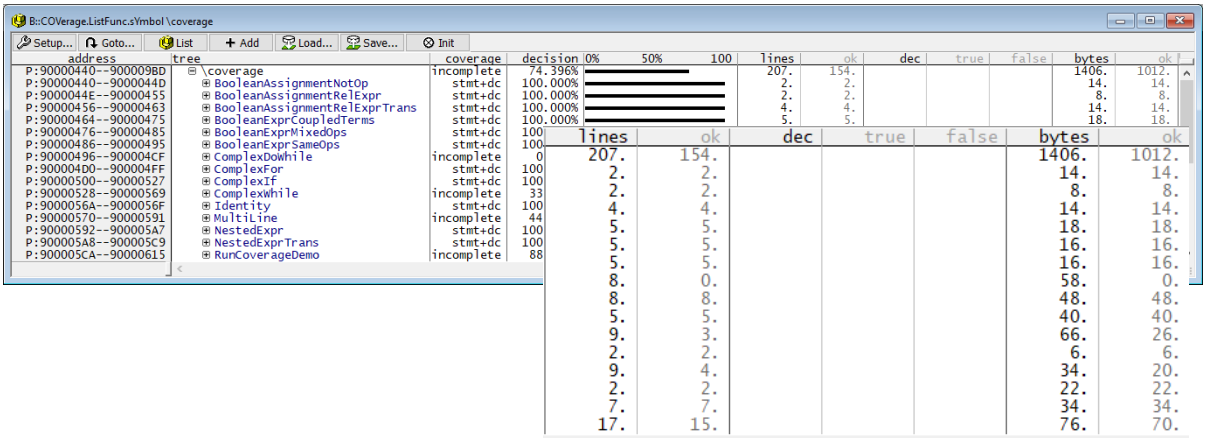




Tags for Object Code Based (ocb) Decision Coverage

- **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

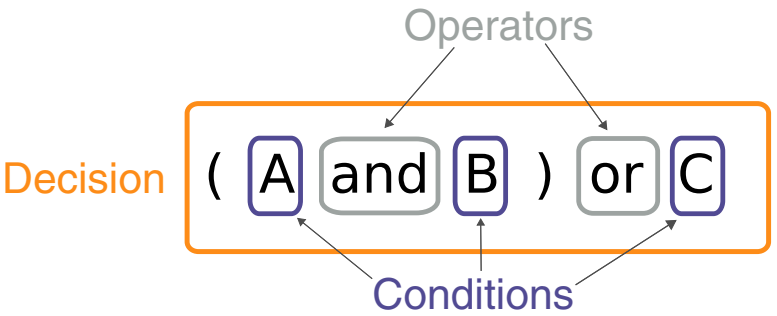
Byte count	
bytes	Number of bytes within the function/module
ok	Number of bytes tagged with stmt+dc

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
```

Condition Coverage

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.

To measure condition coverage accurately the following prerequisites must be fulfilled:

1. It is necessary that the code is compiled in such a way that each condition in the source code is represented by a distinct conditional branch/instruction at object code level. Conditional instructions, however, require that the trace protocol includes information about conditional instructions.

   Please read **"Appendix B: Coding Guidelines"**, page 112 to ensure that you write decisions and conditions at source code level in such a way that your build toolchain generates conditional branches/instructions for them.

   Ensure that the compiler generates conditional branches for switch-case statements. A dedicated compiler option is commonly available to control this. Please refer to the documentation of your build toolchain.

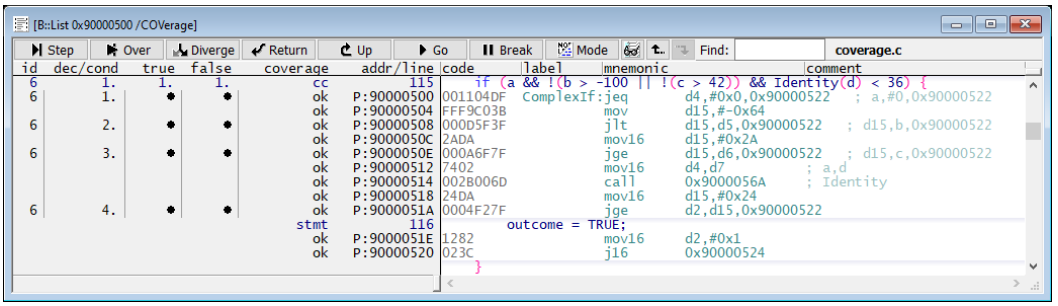
2. TRACE32 has to know which source code lines contain a condition.

   These source code details are not part of the debug symbol information generated by the compiler, but must be generated separately. TRACE32 provides the command line tool t32cast for this purpose. For complete information about t32cast, see **"Application Note for t32cast"** (app_t32cast.pdf).

   The t32cast command line tool generates an Extended Code Analysis (ECA) data file for each source code file. These files have to be loaded into TRACE32 before starting the code coverage analysis.

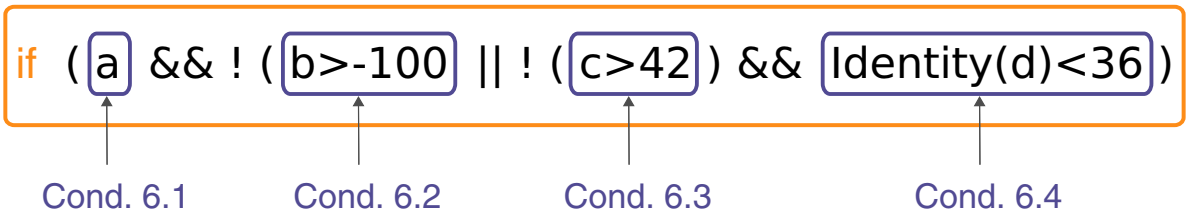
TRACE32 Interpretation: A condition achieved condition coverage when the execution of its conditional branches/instructions results in both a true and false outcome.

The following screenshot illustrates this:



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

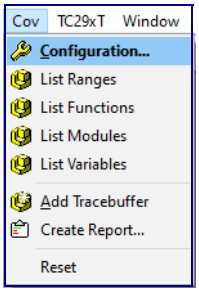


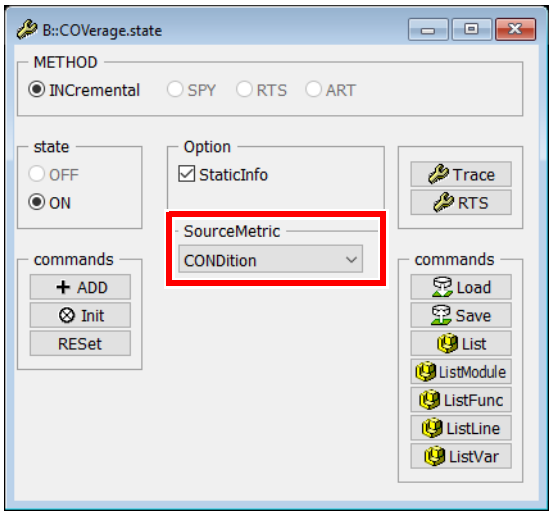
Source code lines that contain conditions are tagged as follows:

- **cc | incomplete**

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

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

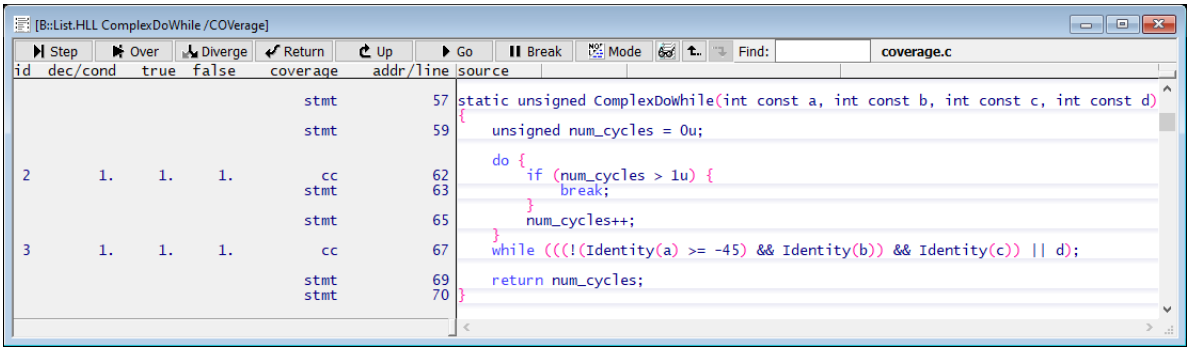
COVerage.ListFunc

The following command shows the tagging on source code level.

List.HII /COVerage

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

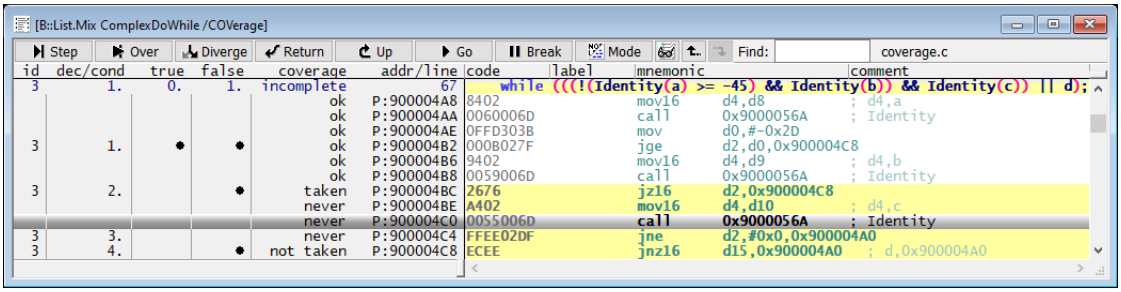
```
List.HLL ComplexDoWhile /COVerage
```



Decisions are tagged as follows:

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

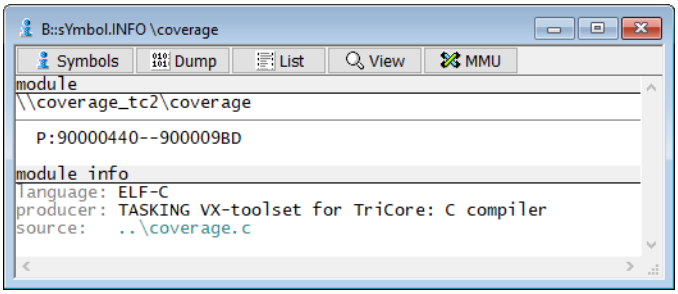
TRACE32 displays the result in **mixed mode** in such a way that it is clear which atomic conditions are still missing for a full condition coverage.

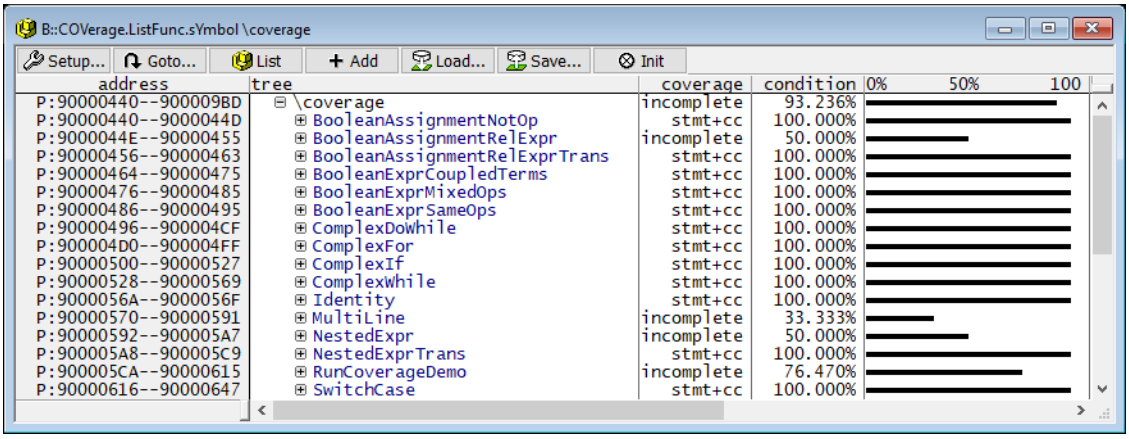


Object code instructions show the corresponding tags for object code coverage, if condition coverage is selected.

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
```

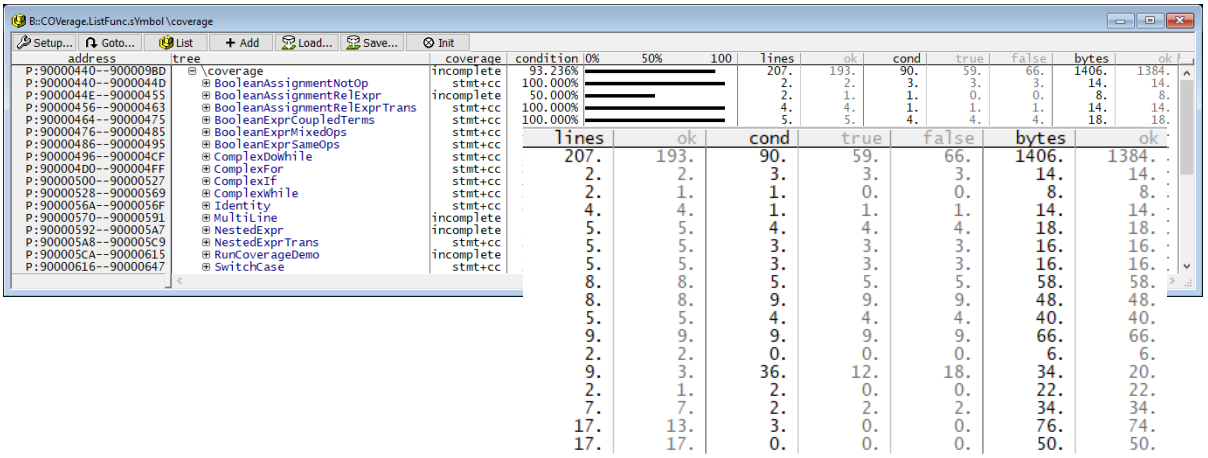




Tags for Condition Coverage

- **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

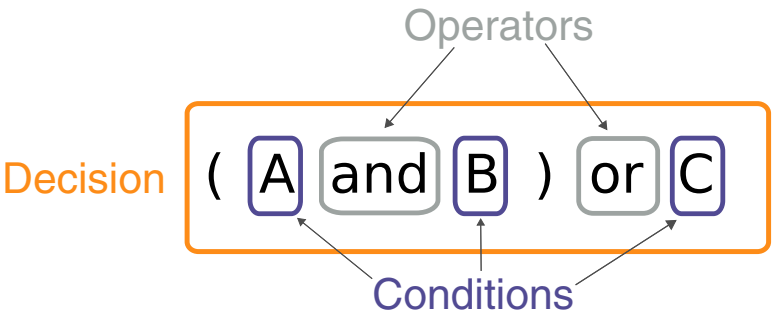
Byte count	
bytes	Number of bytes within the function/module
ok	Number of bytes tagged with cc or stmt

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)

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.

Independence Pairs are used to proof that each condition in a decision is shown to independently affect the outcome of that decision. An independence pair has two characteristics:

1. All conditions except the one to be tested are fixed.
2. The decision changes its outcome when the condition under test is changed.

The following figure shows the truth table for the decision (A and B) or C. The independence pairs for the individual conditions are highlighted in color.

#	A	B	C	
1	F	F	F	F
2	T	F	F	F
3	F	T	F	F
4	F	F	T	T
5	T	T	F	T
6	T	F	T	T
7	F	T	T	T
8	T	T	T	T

To measure MC/DC accurately the following prerequisites must be fulfilled:

1. It is necessary that the code is compiled in such a way that each condition in the source code is represented by a distinct conditional branch/instruction at object code level. Conditional instructions, however, require that the trace protocol includes information about conditional instructions.

   Please read **"Appendix B: Coding Guidelines"**, page 112 to ensure that you write decisions and conditions at source code level in such a way that your build toolchain generates conditional branches/instructions for them.

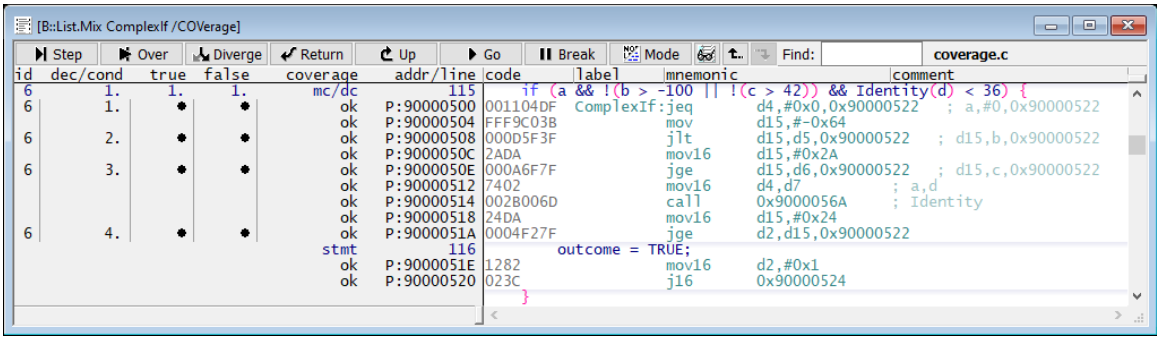
   Ensure that the compiler generates conditional branches for switch-case statements. A dedicated compiler option is commonly available to control this. Please refer to the documentation of your build toolchain.

2. TRACE32 has to know which source code lines contain decisions and their conditions. And for each condition used in a decision the mapping between the conditions and their conditional branches/instructions is required.

   These source code details are not part of the debug symbol information generated by the compiler, but must be generated separately. TRACE32 provides the command line tool t32cast for this purpose. For complete information about t32cast, see **"Application Note for t32cast"** (app_t32cast.pdf).

   The t32cast command line tool generates an Extended Code Analysis (ECA) data file for each source code file. These files have to be loaded into TRACE32 before starting the code coverage analysis.

The following screenshot illustrates all this:



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

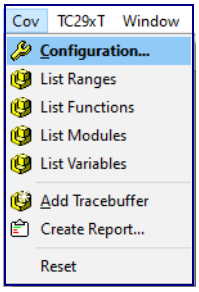
The point for true is set in the true column if the condition has been independently tested for true. The same applies to false.

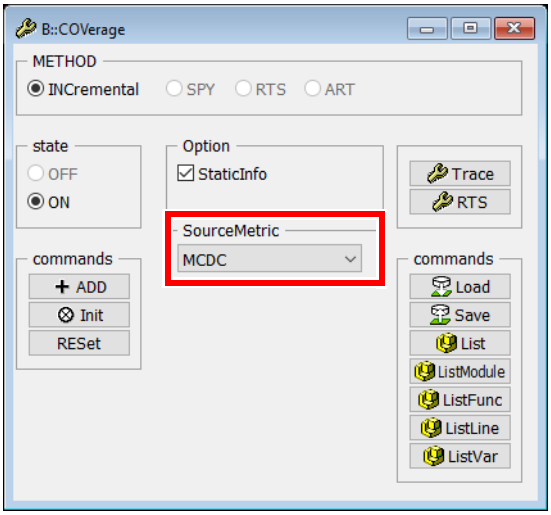
Source code lines that contain decisions are tagged as follows:

- **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

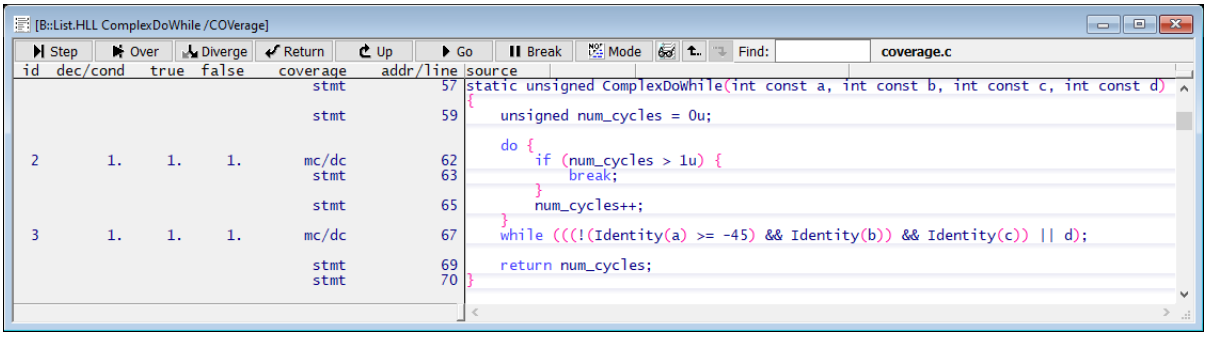
COVerage.ListFunc

The following command shows the tagging on source code level.

List.HII /COVerage

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

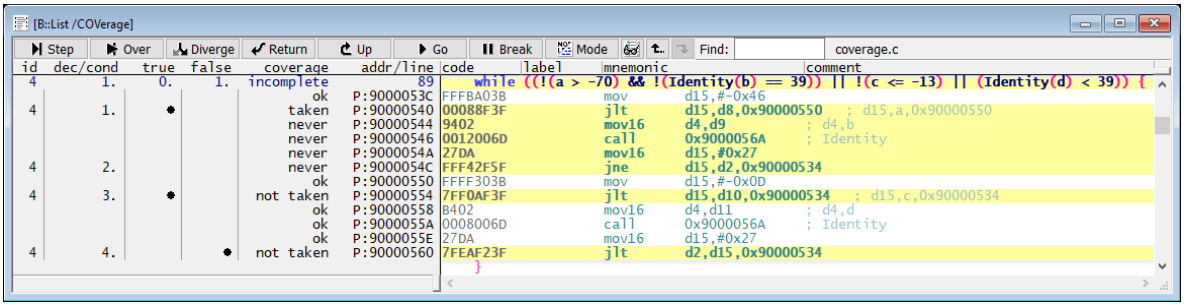
```
List.HLL ComplexDoWhile /COVerage
```



Decisions are tagged as follows:

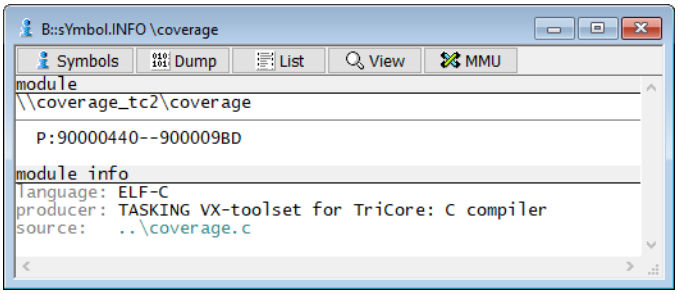
mc/dc	Each condition in a decision is shown to independently affect the outcome of that decision.
incomplete	There is at least one condition in the decision for which has not yet proven to independently affect the outcome of the decision.

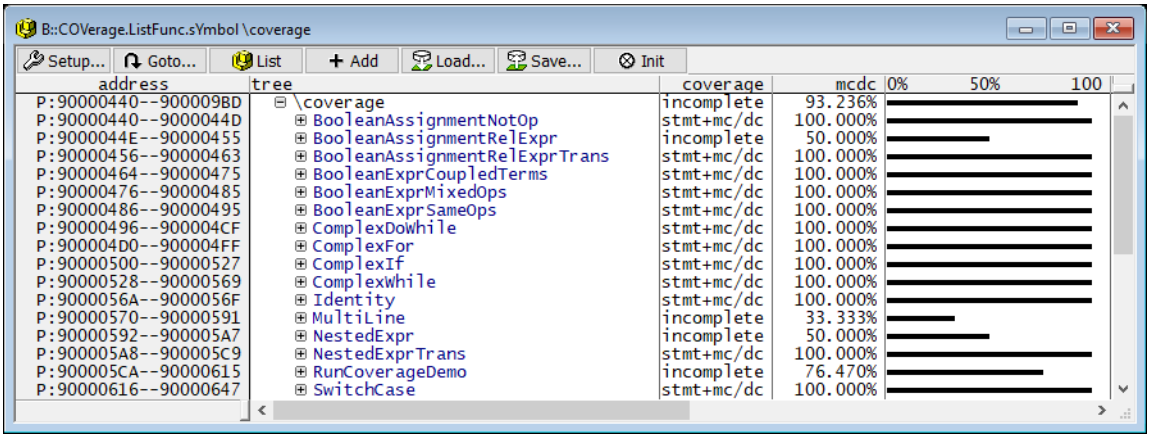
TRACE32 displays the result in **mixed mode** in such a way that it is clear which conditions are still missing for MC/DC.



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)

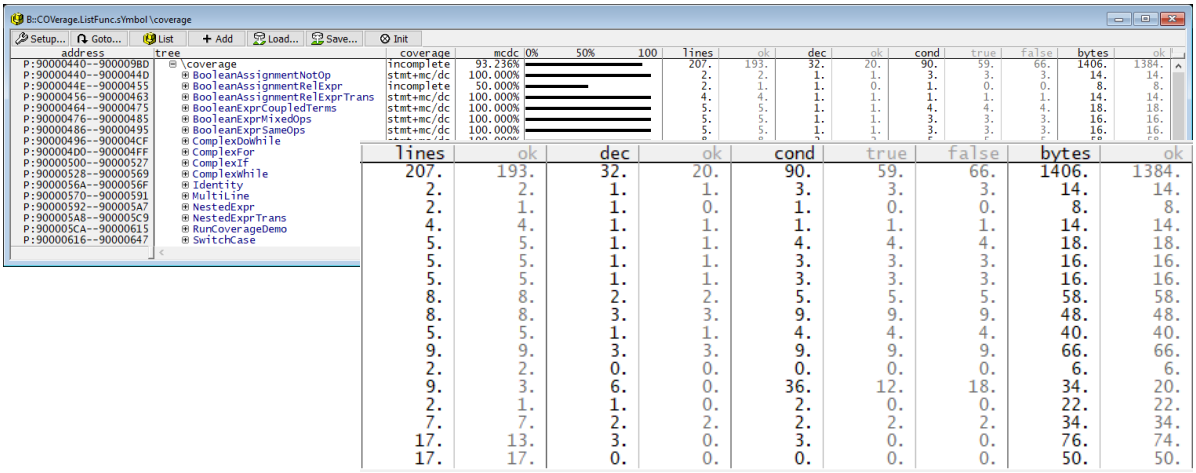
MC/DC is achieved for a group of **HLL source code statements** as soon as the independence effect of all of its associated conditional branches/instructions has been demonstrated.

- **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

It is recommended to perform function coverage on unoptimized code. This way the results can be displayed clearly and concisely. In case of highly optimized code that inlines functions, a deep understanding of the inlining is necessary to interpret the results.

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.

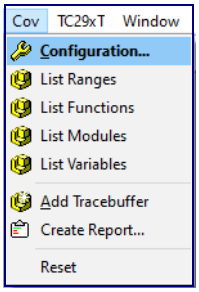
Functions are tagged as follows:

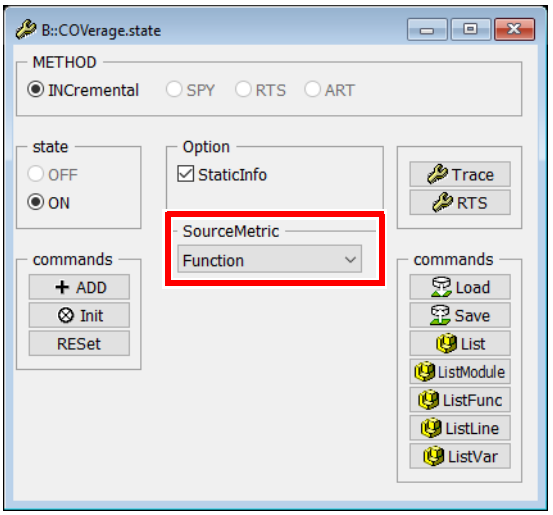
- **func** | **incomplete**

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

Object code coverage tagging is applied to instructions.

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:

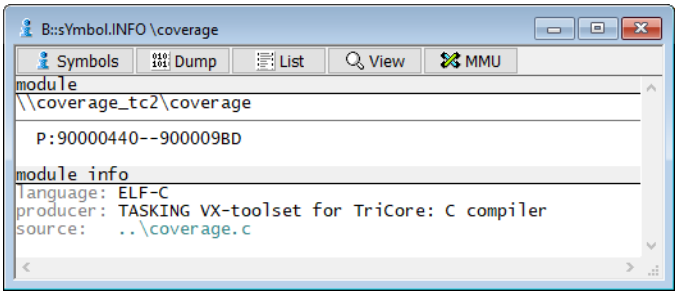
COVerage.ListModule

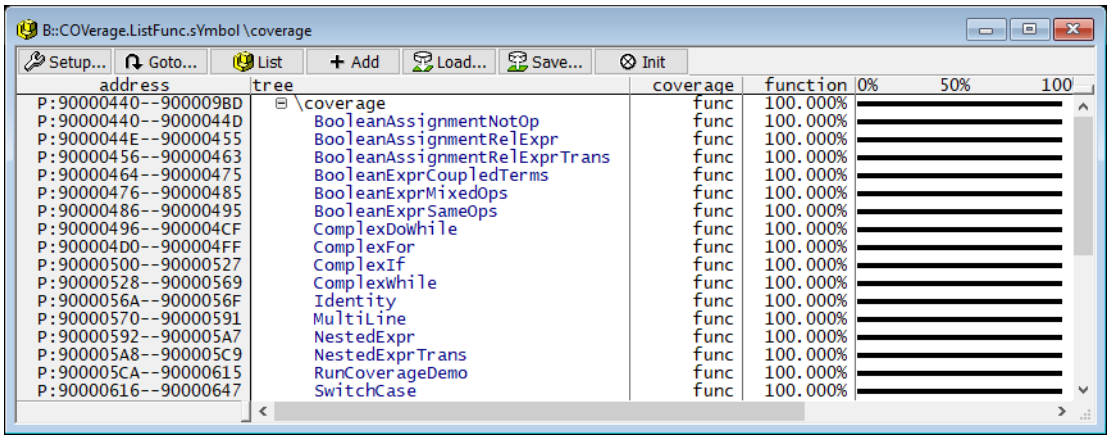
The following command shows the tagging at function level.

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
```



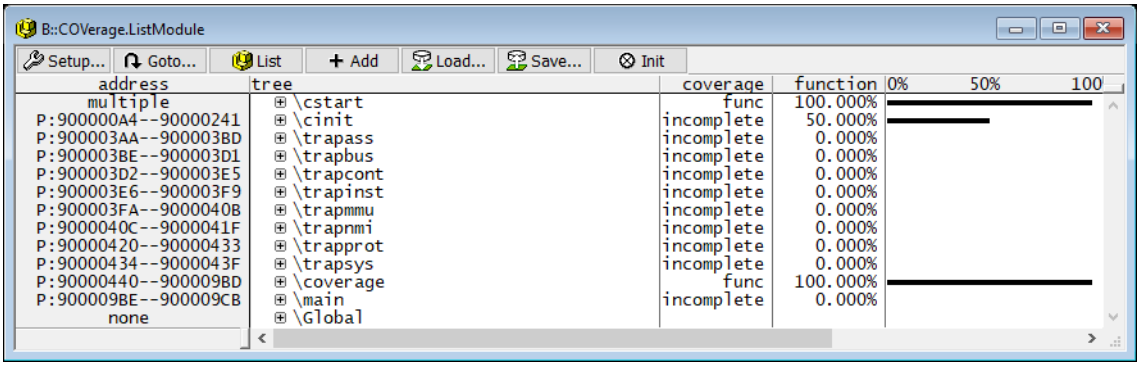


The functions are tagged as follows:

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.

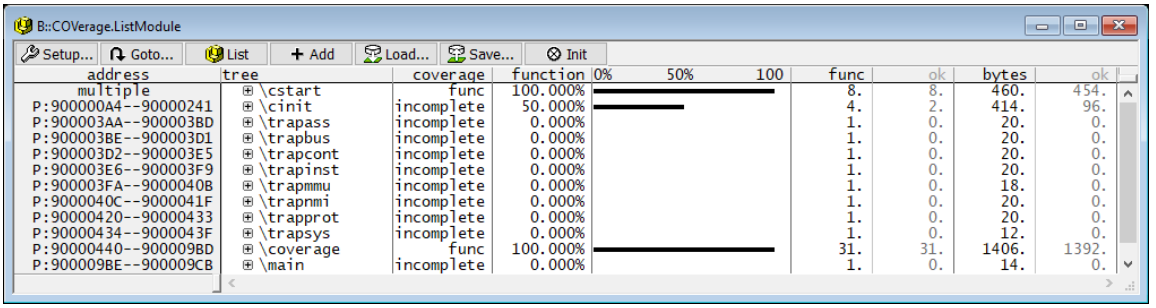
If a tag marks the coverage status of a **function**, the following definitions apply:

- **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:



address	tree	coverage	function	func	ok	bytes	ok
multiple	⊞ \cstart	func	100.000%	8.	8.	460.	454.
P:900000A4--90000241	⊞ \cinit	incomplete	50.000%	4.	2.	414.	96.
P:900003AA--900003BD	⊞ \trapass	incomplete	0.000%	1.	0.	20.	0.
P:900003BE--900003D1	⊞ \trapbus	incomplete	0.000%	1.	0.	20.	0.
P:900003D2--900003E5	⊞ \trapcont	incomplete	0.000%	1.	0.	20.	0.
P:900003E6--900003F9	⊞ \trapinst	incomplete	0.000%	1.	0.	20.	0.
P:900003FA--9000040B	⊞ \trapmmu	incomplete	0.000%	1.	0.	18.	0.
P:9000040C--9000041F	⊞ \trapnmi	incomplete	0.000%	1.	0.	20.	0.
P:90000420--90000433	⊞ \trapprot	incomplete	0.000%	1.	0.	20.	0.
P:90000434--9000043F	⊞ \trapsys	incomplete	0.000%	1.	0.	12.	0.
P:90000440--900009BD	⊞ \coverage	func	100.000%	31.	31.	1406.	1392.
P:900009BE--900009CB	⊞ \main	incomplete	0.000%	1.	0.	14.	0.

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

It is recommended to perform call coverage on unoptimized code. This way the results can be displayed clearly and concisely. In case of highly optimized code that inlines functions, a good understanding of the inlining is necessary to interpret the results.

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

TRACE32 has to know which source code lines contain a function call. This information is not part of the debug symbol information generated by the compiler, but must be generated separately. TRACE32 provides the command line tool t32cast for this purpose. For complete information about t32cast, see **"Application Note for t32cast"** (app_t32cast.pdf).

The t32cast command line tool generates an Extended Code Analysis (ECA) data file for each source code file. These files have to be loaded into TRACE32 before starting the code coverage analysis.

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

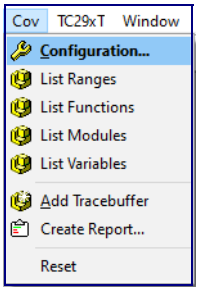
Functions are tagged as follows:

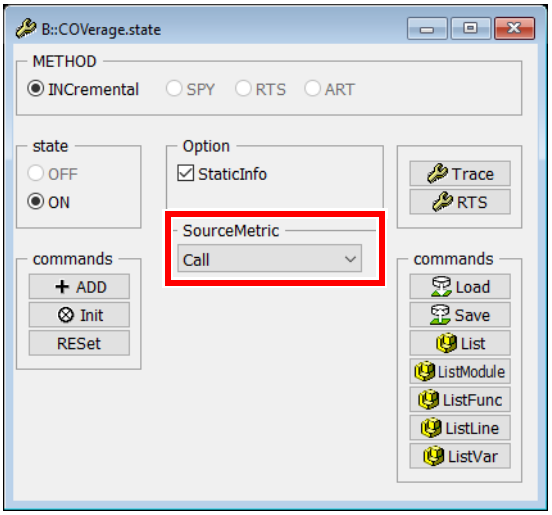
- **call** | **incomplete**

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

Object code coverage tagging is applied to instructions.

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.





Before you start the code coverage analysis, you have to load the .eca files created by the command line tool t32cast:

```
sYmbol.ECA.LOADALL /SkipErrors
```

The following command shows a tabular analysis:

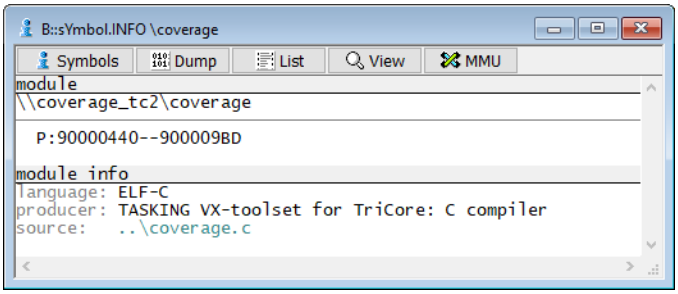
```
COVerage.ListModule
```

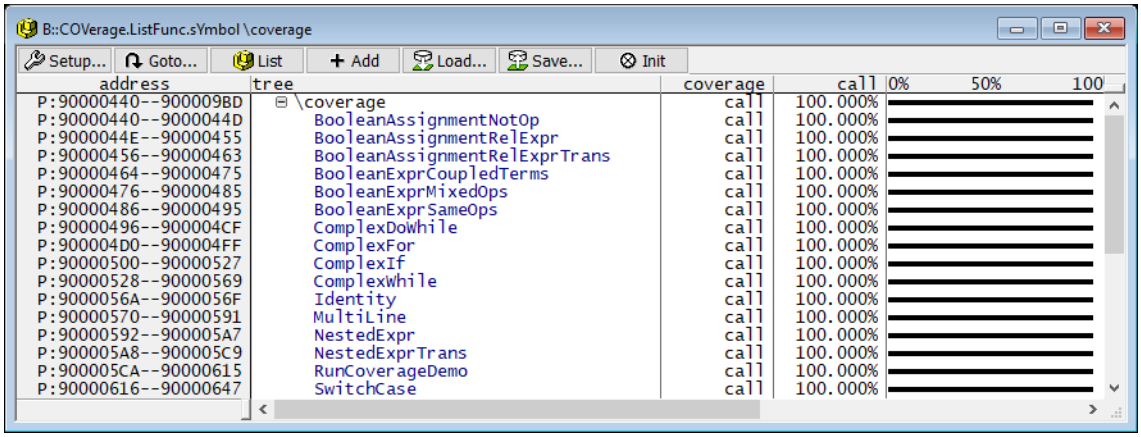
The following command shows the tagging at function level.

```
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
```



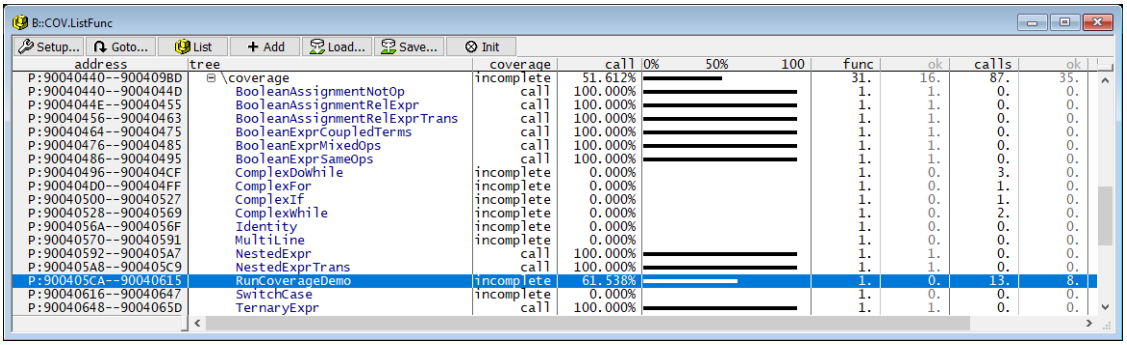


The functions are tagged as follows:

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.

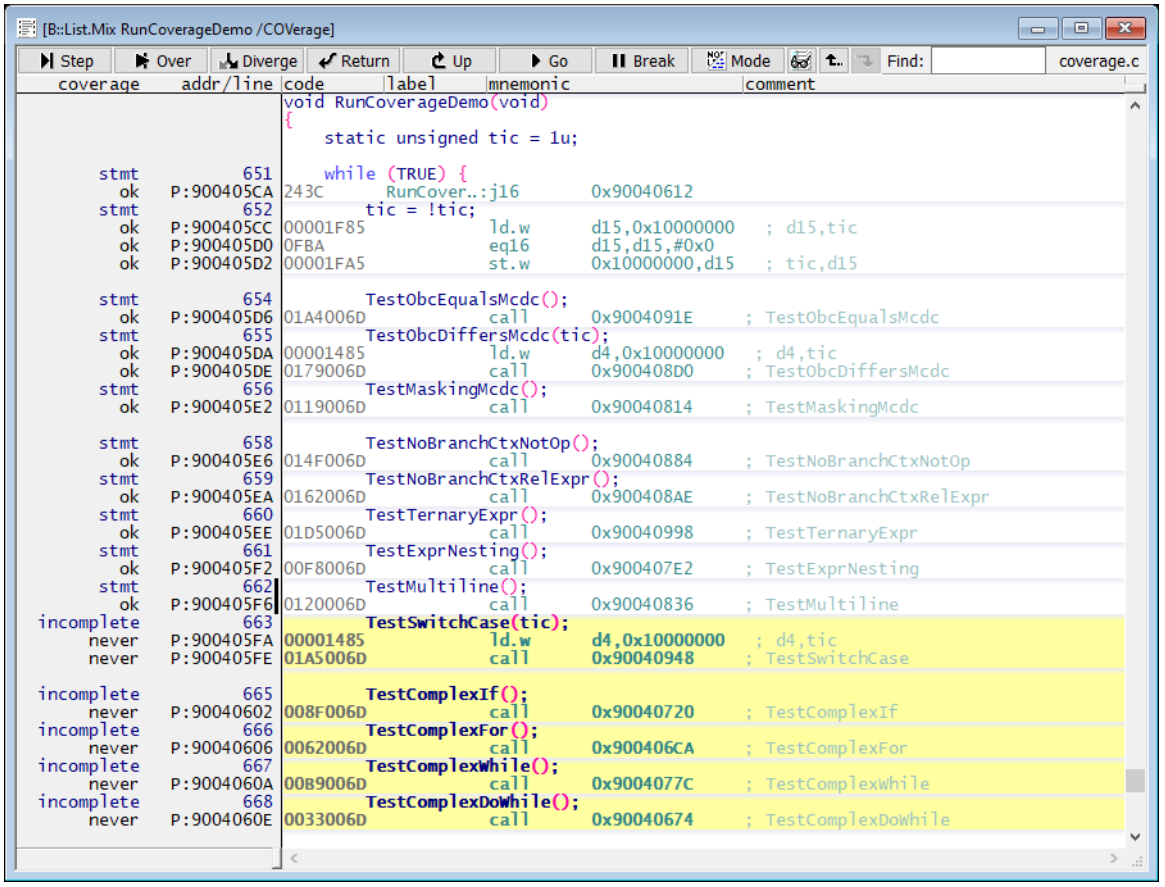
The full-width **COVerage.ListFunc** window provides details on the function calls:

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



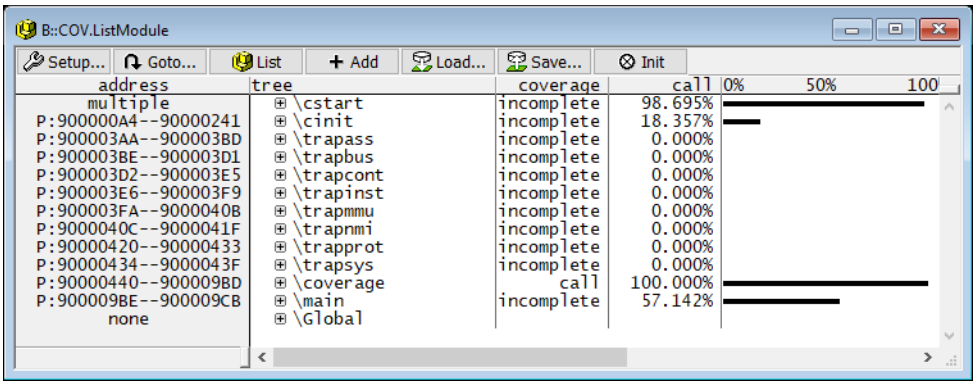
If a function is tagged as incomplete you can inspect its details. Either by doing a left mouse double click on the function‘s name or by using the following command:

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



This TRACE32 command displays a tabular analysis of all modules.

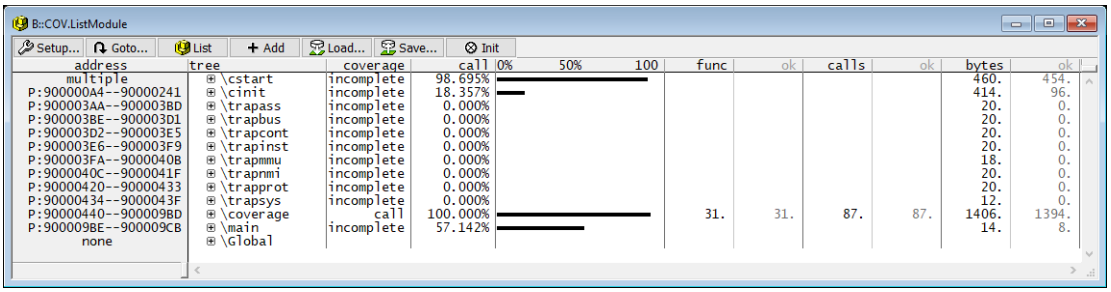
```
COVerage.ListModule
```



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

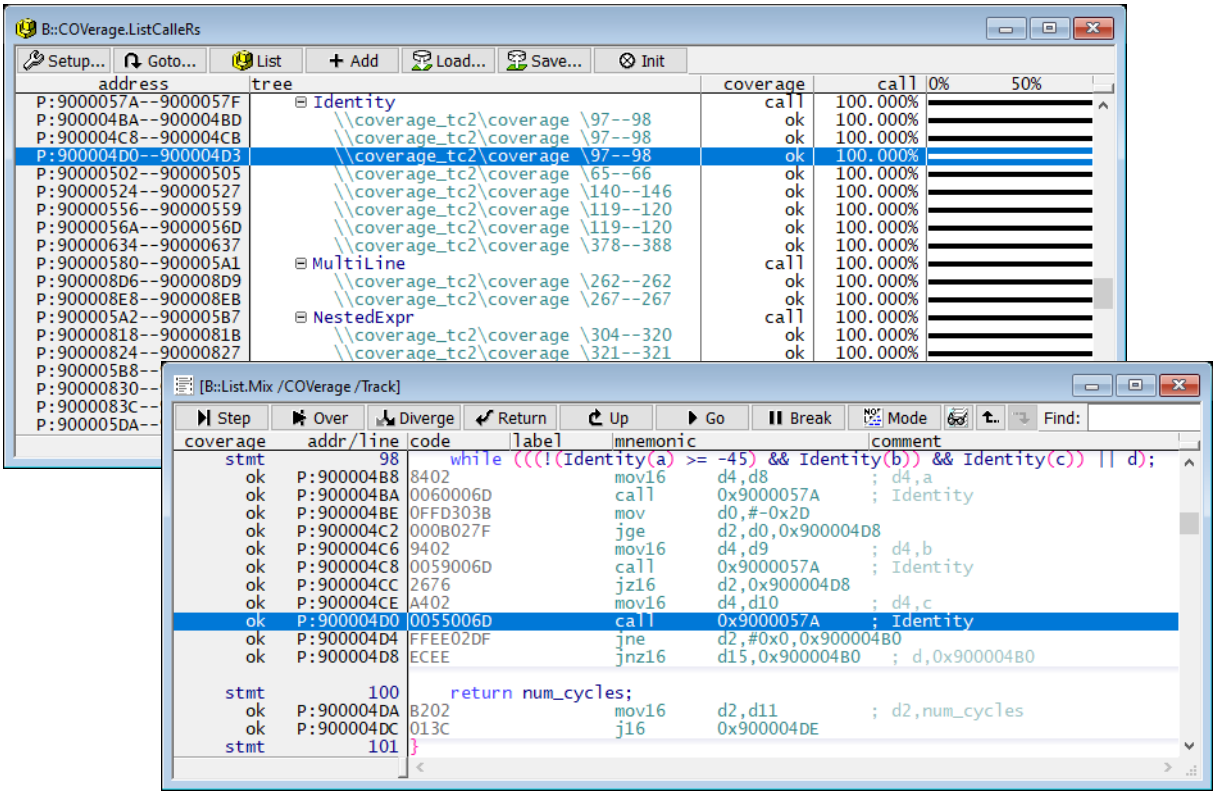
bytes	Number of bytes
ok	Number of bytes tagged with call

Details on Callers and Calles

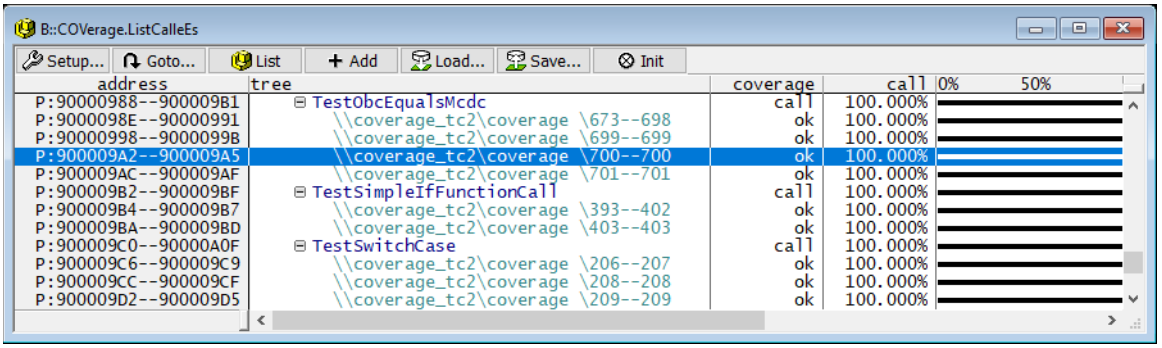
For a detailed analysis it is helpful to get details about the calling and the called functions.

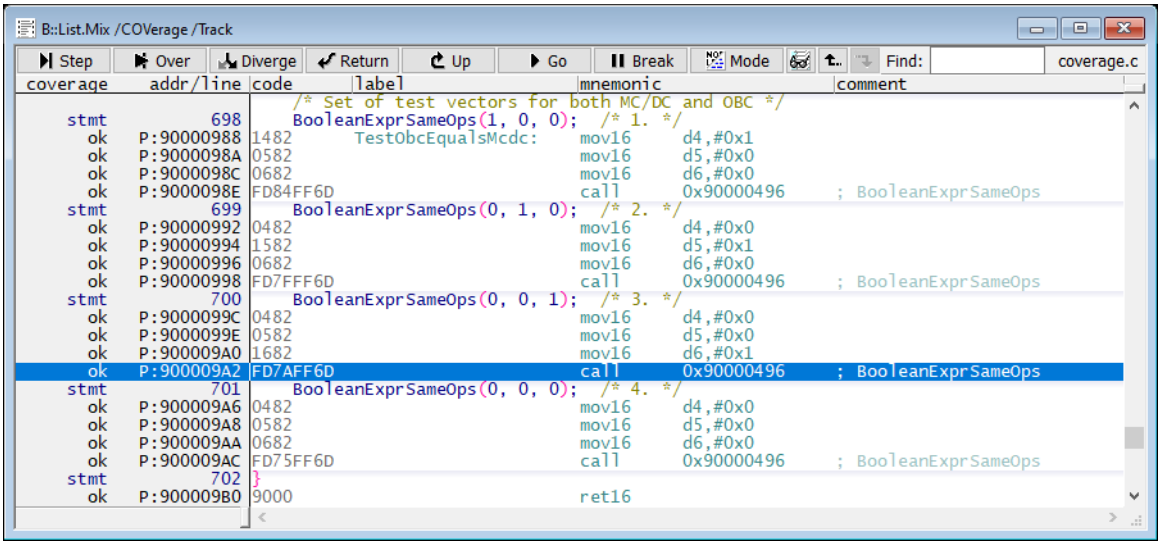
COVerage.ListCalleRs	Display call coverage with caller details at source code line level
COVerage.ListCalleEs	Display call coverage with callee details at source code line level
List.Mix /COVerage /Track	Display a source listing that displays source and object code. This window is used here to inspect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

All callers of the function **Identity** are inspected in this example. The COVerage.ListCallRs 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.ListCallEs 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

Assemble Multiple Test Runs

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

...
```

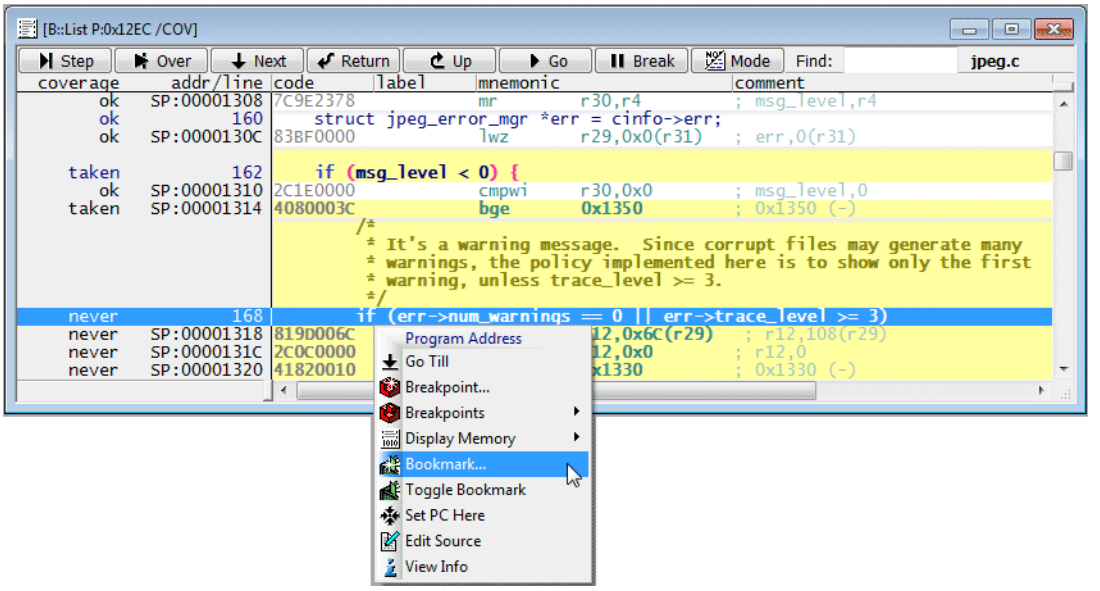
Reload saved trace buffer contents and add them to the code coverage system.

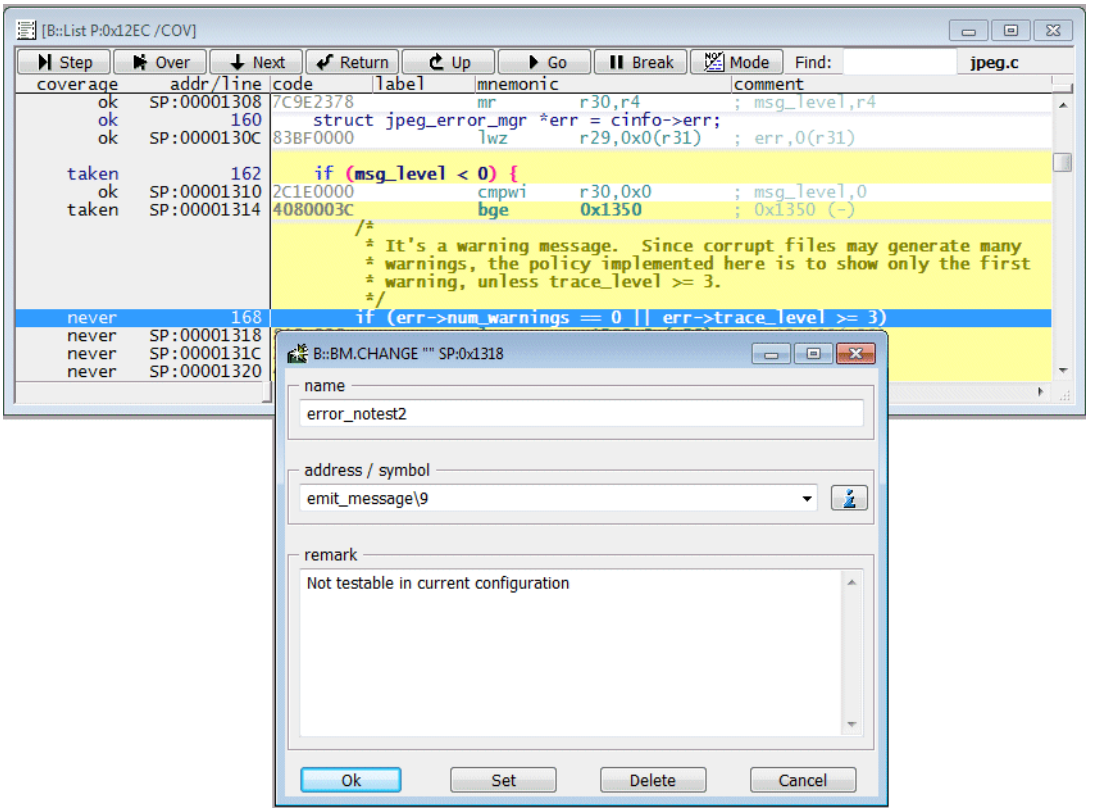
```
...                                          ; Basic setups

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
```

Comment your Results

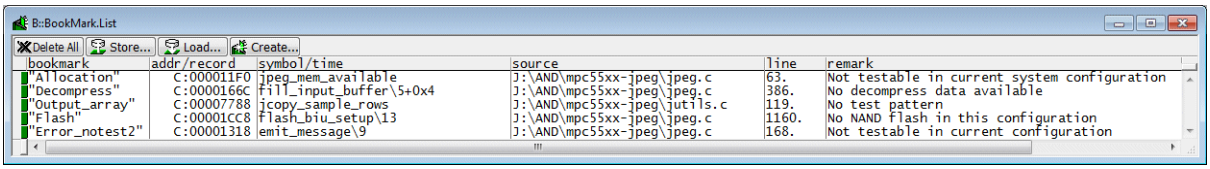
Address-based bookmarks can be used to comment not covered code ranges, which are fine but not testable in the current system configuration.





List all bookmarks:

BookMark.List



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

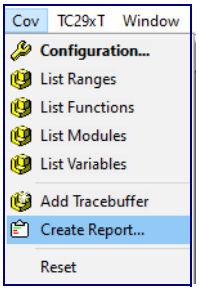
The current bookmarks can be saved to a file and reloaded later on.

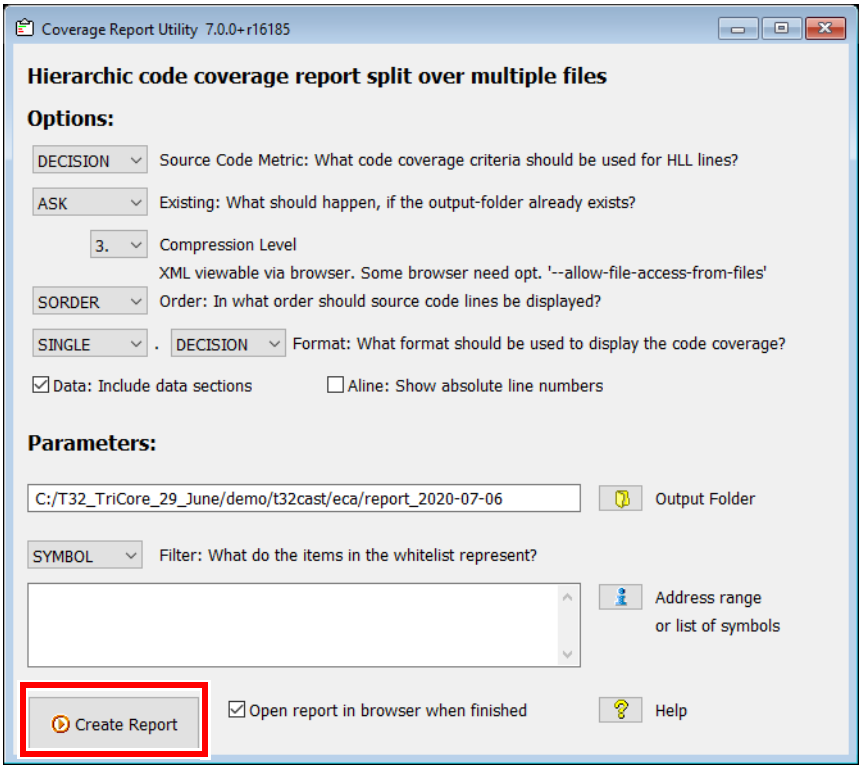
STOre *<file>* BookMark

TRACE32 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
```

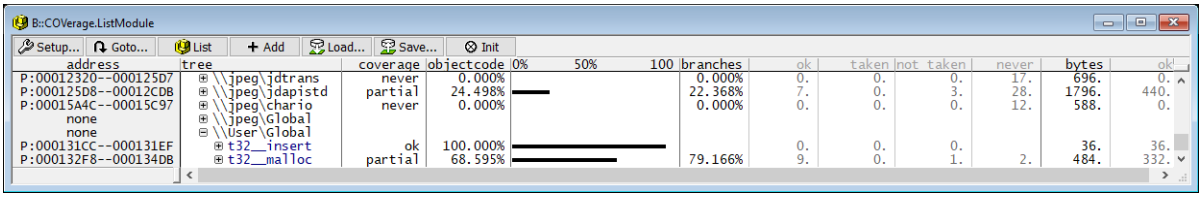
Object Code Coverage

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

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

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

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

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



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

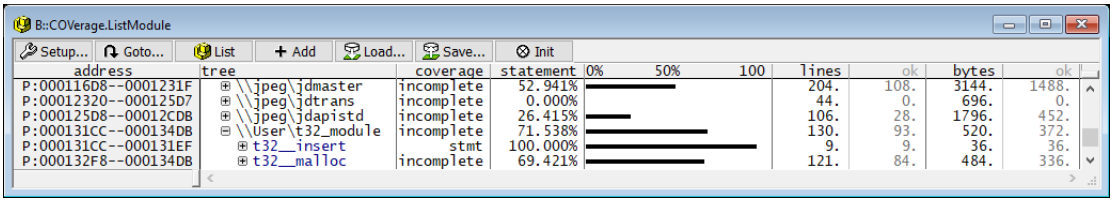
Code that is not part of a source code function is discarded for coverage. If you want to include this code you have to assign a function to it:

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

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

```
sYmbol.INFO __malloc

sYmbol.INFO __insert

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

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

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



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

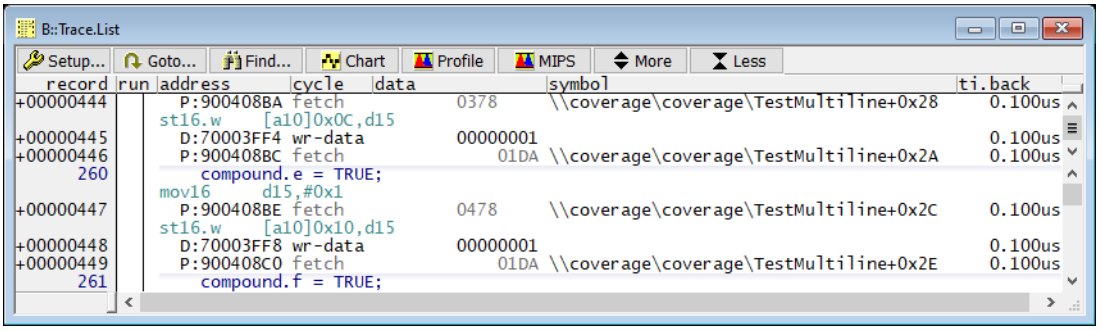
Metric	Tag	Description
all source code metrics	incomplete	At least one assembler line within the function is tagged with never, taken or not taken.
Statement	stmt	All assembler lines are tagged with ok.

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

Trace Data Collection

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

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



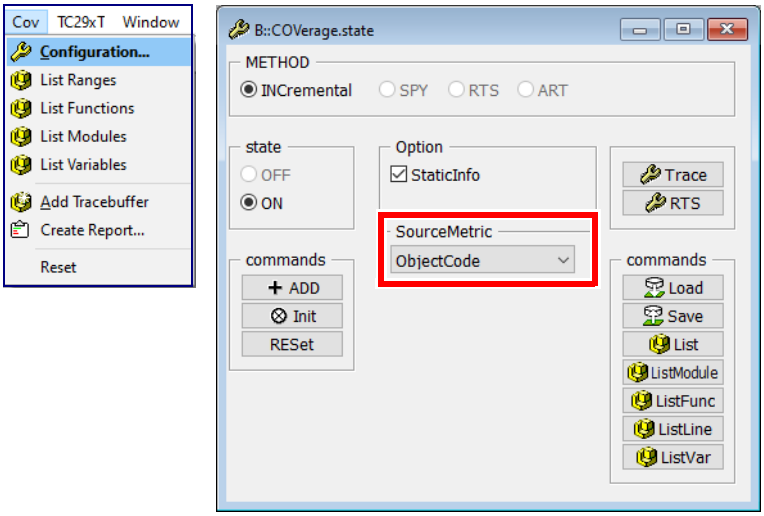
If you want to use an onchip trace or an offchip trace port for data tracing, please refer to the following documents for setup details:

- Arm: **"Training 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 15).

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:

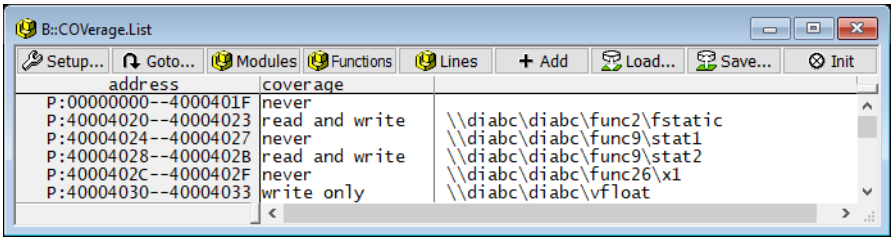
> **COVerage.List**
>
> **COVerage.ListVar**

The following command shows the tagging per address.

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

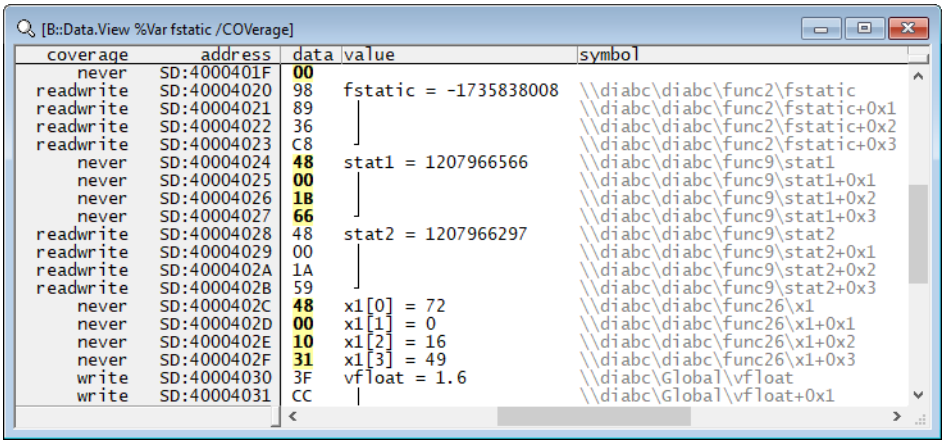
This TRACE32 command shows the coverage tagging on address range level:

```
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
```

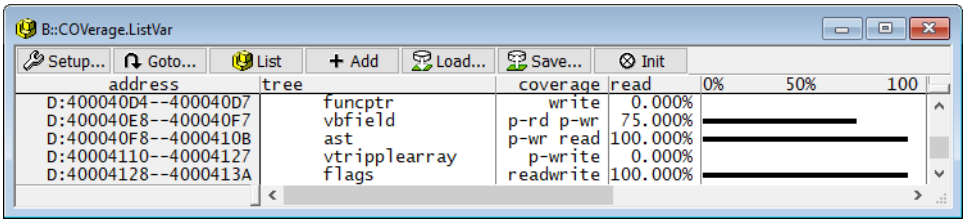


The data addresses are tagged as follow:

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 119.

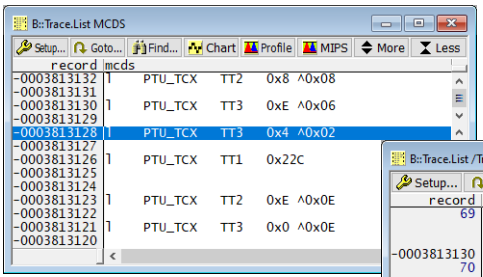
Document the Results

With the script "~~/demo/coverage/single_file_report/create_report.cmm" you can create a coverage report.

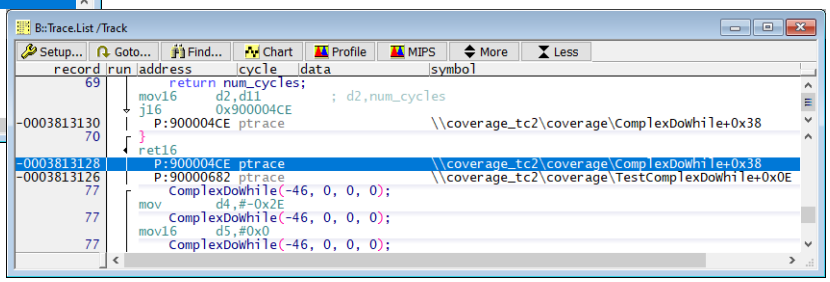
Data coverage is not yet integrated into the TRACE32 Coverage Report Utility (see **“TRACE32 Coverage Report Utility”**, page 98). If you need this, please contact your local support.

Appendix A: 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.

Trace Decoding for Applications Using a Rich OS

Also in this case, the object code and source code are needed to decode the trace raw data. But paging used by the operating system makes decoding more complex.

Since the onchip trace logic generates the program flow data based on virtual addresses, TRACE32 has to know the valid memory space for each trace record in order to read the object code from the physical memory for trace decoding. A task or context switch in the trace recording normally identifies the memory space for the subsequent logical addresses.

Decoding in Stopped State (Rich OS)

This decoding is used for incremental code coverage and incremental code coverage in stream mode.

TRACE32 state: program execution stopped, no recording of trace data.

Trace decoding is performed in three steps:

1. TRACE32 reads the current task list and all task page tables with the help of the TRACE32 OS Awareness from the target, when the program execution is stopped.
2. Task/context switches from the trace recording are decoded with the help of the task list.
3. The object code for each task is then read with the help of its page table. Links to the source code files are part of the debug symbol information, which TRACE32 maintains for each memory space.

   Reading the object code fails, when a task/context switch from the trace recording can not be decoded with the help of the current task list, e.g. because the task was terminated.

Decoding in Running State (Rich OS)

This decoding is used in Spy mode code coverage.

TRACE32 state: program execution is running, trace data is recorded, but trace streaming is stalled while trace decoding is performed.

TRACE32 has no access to the current task list and the task page tables while the program execution is running. The **TRACE32 Virtual Memory** must contain the task list, all task page tables and the object code to enable TRACE32 to decode the raw trace data.

This requires a complex setup. Please contact the Lauterbach support in this case.

RTS Decoding (Rich OS)

This decoding is used in RTS mode code coverage.

TRACE32 state: program execution is running, trace data is recorded and streamed to the host computer.

TRACE32 has no access to the current task list and the task page tables while the program execution is running. The **TRACE32 Virtual Memory** must contain the task list, all task page tables and the object code to enable TRACE32 to decode the raw trace data.

This requires a complex setup. Please contact the Lauterbach support in this case.

Appendix B: 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`.

```
[B::List P:0x9000044E /COV]
Step  Over  Diverge  Return  Up  Go  Break  Mode  Find:  coverage.c
id  dec/cond  true  false  coverage  addr/line  code  label  mnemonic  comment
                                                {
                                                  /* Relational expression as decision
                                                   *
                                                   * Expression showing a decision in non-branching context. Compilers may
                                                   * choose to model Boolean assignments with conditional or unconditional
                                                   * instructions instead of conditional branches that are not suitable for
                                                   * the trace-based measurement of code coverage.
                                                   */
18        -     -     -  incomplete      373      return a == b;
                                   ok  P:9000044E  2100540B  BooleanA..:eq   d2,d4,d5   ; d2,a,b
                                   ok  P:90000452  013C              j16    0x90000454
                                 stmt        374  }
                                   ok  P:90000454  9000              ret16
```

It is recommended to write the source code in a way that ensures that the conditional branches/instructions required for the trace-based code coverage are generated.

```
[B::List P:0x90000456 /COV]
Step  Over  Diverge  Return  Up  Go  Break  Mode  Find:  coverage.c
id  dec/cond  true  false  coverage  addr/line  code  label  mnemonic  comment
                                                  /* Equivalence transformation for relational expression
                                                   *
                                                   * Equivalent expression after transformation. The decision appears in a
                                                   * branching context once more. Compilers typically choose to use
                                                   * conditional branches for modelling this type of expression.
                                                   */
17        1.    1.    1.        dc        357     if (a == b) {
17 |      1. |   *|    *|      ok  P:90000456  8004545F  BooleanA..:jne   d4,d5,0x9000045E   ; a,b,0x9000045E
                                 stmt        358       return TRUE;
                                   ok  P:9000045A  1282              mov16  d2,#0x1
                                   ok  P:9000045C  033C              j16    0x90000462
                                                   }
                                 stmt        360     return FALSE;
                                   ok  P:9000045E  0282              mov16  d2,#0x0
                                   ok  P:90000460  013C              j16    0x90000462
                                 stmt        361  }
                                   ok  P:90000462  9000              ret16
```

A few examples:

```
; source code not suitable for
; 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.

```
[B::List P:0x90000592 /COV]
Step  Over  Diverge  Return  Up  Go  Break  Mode  Find:  coverage.c
id  dec/cond  true  false  coverage  addr/line  code  label  mnemonic  comment
                                                  /* Decision with nested Boolean expression
                                                   *
                                                   * Expression showing a nested Boolean expression. Compilers may choose to
                                                   * model nested expressions with conditional or unconditional instructions
                                                   * instead of conditional branches that are not suitable for the trace-
                                                   * based measurement of code coverage.
                                                   */
13        -     -     -  incomplete        271    return (a > (b + ((float) b < c)));
                                     ok  P:90000592  F141054B  NestedExpr:  itof     d15,d5        ; d15,b
                                     ok  P:90000596  F0016F4B              cmp.f    d15,d15,d6    ; d15,d15,c
                                     ok  P:9000059A  F0610F37              extr.u   d15,d15,0x0,#0x1
                                     ok  P:9000059E  F542                  add16    d5,d15        ; b,d15
                                     ok  P:900005A0  2120450B              lt       d2,d5,d4      ; d2,d5,a
                                     ok  P:900005A4  013C                  j16      0x900005A6
                                  stmt          272  }
                                     ok  P:900005A6  9000                  ret16
```

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

```
B::List P:0x900005A8 /COV
Step  Over  Diverge  Return  Up  Go  Break  Mode  Find:  coverage.c
id  dec/cond  true  false  coverage  addr/line  code  label  mnemonic  comment
                                                  ...
                                                  {
                                                     /* Equivalence transformation for decision with nested Boolean expression
                                                      *
                                                      * Equivalent expression after transformation. The nested Boolean
                                                      * expression is extracted and put into a branching context. Compilers
                                                      * typically choose to use conditional branches for modelling this type of
                                                       * structure.
                                                      */
                                  stmt          249    int tmp = 0;
                                     ok  P:900005A8  0082      NestedEx..:mov16     d0,#0x0

11        1.    1.    1.        dc          251    if ((float) b < c) {
                                     ok  P:900005AA  F141054B              itof     d15,d5        ; d15,b
                                     ok  P:900005AE  F0016F4B              cmp.f    d15,d15,d6    ; d15,d15,c
                                     ok  P:900005B2  F0610F37              extr.u   d15,d15,0x0,#0x1
11        1.    *     *         ok  P:900005B6  026E                  jz16     d15,0x900005BA
                                  stmt          252        tmp = 1;
                                     ok  P:900005B8  1082                  mov16    d0,#0x1       ; tmp,#1
                                                       }

12        1.    1.    1.        dc          255    if (a > (b + tmp)) {
                                     ok  P:900005BA  0542                  add16    d5,d0         ; b,tmp
12        1.    *     *         ok  P:900005BC  0004457F              jge      d5,d4,0x900005C4  ; d5,a,0x900005C4
                                  stmt          256        return TRUE;
                                     ok  P:900005C0  1282                  mov16    d2,#0x1
                                     ok  P:900005C2  033C                  j16      0x900005C8
                                                       }
                                  stmt          258    return FALSE;
                                     ok  P:900005C4  0282                  mov16    d2,#0x0
                                     ok  P:900005C6  013C                  j16      0x900005C8
                                  stmt          259  }
                                     ok  P:900005C8  9000                  ret16
```

Appendix C: Conditional Non-Branch Instructions

Conditional Instructions

Architecture	Instruction Set	Trace Decoder
TriCore	—	—
PowerPC Qorivva(e200) **QorIQ (e5500, e6500)**	yes	yes, if **NEXUS.HTM ON**.
ARC	yes	yes
RISC-V	—	—
Cortex-A9/-A15	yes	no

Appendix D: Object Code Coverage Tags in Detail

Standard Tags

Standard tagging applies to all core architectures and all trace protocols. The only exception are Arm/Cortex cores that use the protocols Arm-ETMv1 or Arm-ETMv3, as well as Arm-ETMv4. However, for the Arm-ETMv4 protocol, this only applies if no trace information about the execution of conditional non-branch instructions is generated in order to save bandwidth (command **ETM.COND OFF**).

The following tags are used for object code coverage tagging:

Tag	Tagging object	Description
ok	conditional branch	The conditional branch has be at least once *taken* and *not taken*.
	conditional instruction	The object code instruction has been executed at least once with its condition code true and once with its condition code false.
	all other object code instructions	The object code instruction has been executed at least once.
taken	conditional branch	The conditional branch has be at least once *taken*, but never *not taken*.
	conditional instruction	The object code instruction has been executed at least once with its condition code true, but never with its condition code false.
not taken	conditional branch	The conditional branch has be at least once *not taken*, but never *taken*.
	conditional instruction	The object code instruction has been executed at least once with its condition code false, but never with its condition code true.
never	all object code instructions	The object code instruction has never been executed.

The following tags apply for analysis at the source code, function or module level:

Tag	Tagging object	Description
ok	range of object code instructions	All object code instructions within the range are tagged with ok.
partial	range of object code instructions	Not all object code instructions within the range are tagged with ok.
branches	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *taken* and one that is only *not taken*.
taken	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *taken*.
not taken	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *not taken*.
never	range of object code instructions	Not a single object code instruction within the range has been executed.

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

The following tags are used for object code coverage tagging:

Tag	Tagging object	Description
ok	conditional branch	The conditional branch has be at least once taken and not taken.
	conditional instruction	The object code instruction has been executed at least once with its condition code true and once with its condition code false.
	all other object code instructions	The object code instruction has been executed at least once.

Tag	Tagging object	Description
only exec	conditional branch	The conditional branch has be at least once taken, but never not taken.
	conditional instruction	The object code instruction has been executed at least once with its condition code true, but never with its condition code false.
not exec	conditional branch	The conditional branch has be at least once not taken, but never taken.
	conditional instruction	The object code instruction has been executed at least once with its condition code false, but never with its condition code true.
never	all object code instructions	The object code instruction has never been executed.

The following tags apply for analysis at the source code, function or module level:

Tag	Tagging object	Description
ok	range of object code instructions	All object code instructions within the range are tagged with ok.
partial	range of object code instructions	Not all object code instructions within the range are tagged with ok.
cond exec	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *only exec* and one that is only *not exec*.
only exec	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *only exec*.
not exec	range of object code instructions	All object code instructions within the range were executed, but there is at least one conditional branch/conditional instruction that is only *not exec*.
never	range of object code instructions	Not a single object code instruction within the range has been executed.

Appendix E: Data Coverage in Detail

The data addresses are tagged as follow:

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

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

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

rdwr ok	The address range achieved full object code coverage, and at least one read and one write access occurred to address range.
write ok	The address range achieved full object code coverage, and at least one write access occurred to address range.

read ok	The address range achieved full object code coverage, and at least one read access occurred to address range.
partial	The address range did not achieve full object code coverage. The amount of read and write accesses that have taken place is not further specified.

The coverage status of **HLL source code statements** that have associated data values is indicated by the following tags if a **data trace** is available:

- **rdwr ok**: The HLL source code statement(s) have been fully covered. All associated assembly instructions have been fully covered and at least one read and write access to the data values has been recorded.
- **write ok**: The HLL source code statement(s) have been fully covered. All associated assembly instructions have been fully covered and at least one write access to the data values has been recorded.
- **read ok**: The HLL source code statement(s) have been fully covered. All associated assembly instructions have been fully covered and at least one read access to the data values has been recorded.
- **partial**: The HLL source code statement(s) have not been fully covered. At least one of the associated assembly instructions has not been fully covered. The amount of read and write accesses that have taken place is not further specified.
- **readwrite**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been read and written at least once.
- **write**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been written at least once and not read.
- **read**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been read at least once and not written.
- **p-rd write**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been written at least once. In addition at least one data value has been read.
- **p-wr read**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and all of the data values have been read at least once. In addition at least one data value has been written.
- **p-rd p-wr**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and at least one of the data values has been read and one written.
- **p-write**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and at least one of the data values has been written.
- **p-read**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and at least one of the data values has been read.
- **never**: The HLL source code statement(s) have never been executed. None of the associated assembly instructions has been executed and neither read nor write accesses to the data values have been recorded.