FROM UNIT TO INTEGRATION TO SYSTEM TESTING

Speeding Up Safety Certification with Trace-based Code Coverage

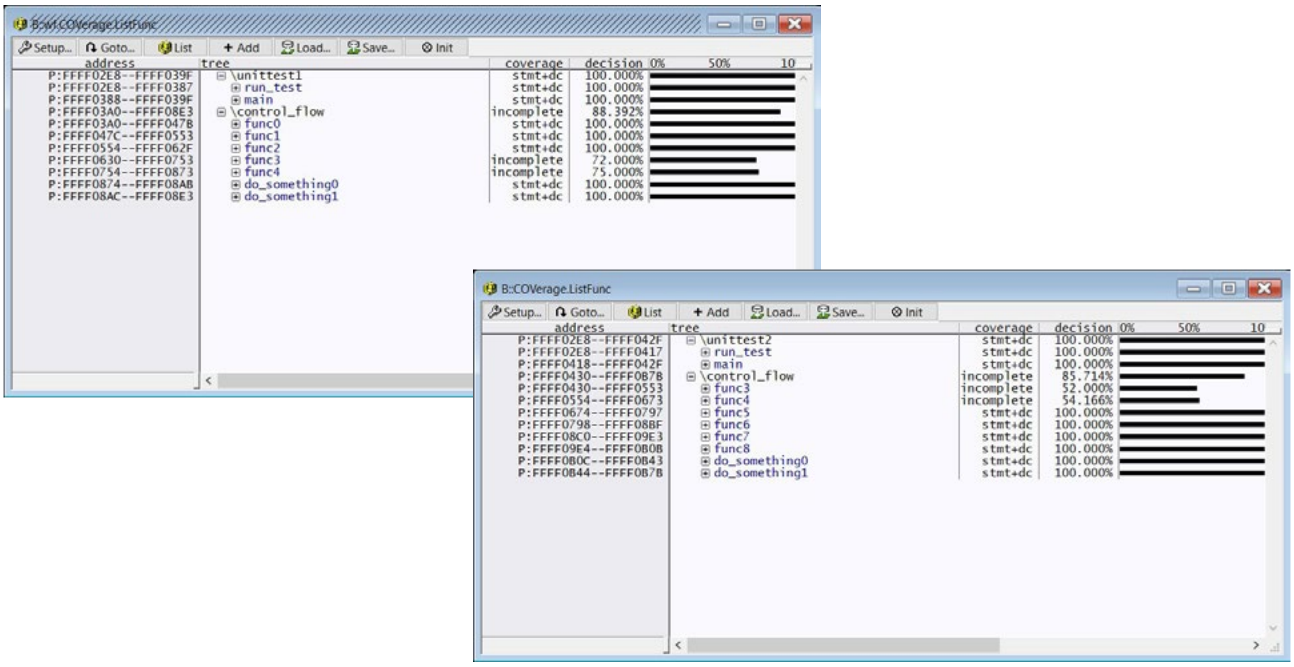
Many embedded systems must be developed according to an internationally recognized safety standard. Part of the data required to prove that a system meets these standards is a form of code coverage. In this article, we show how code coverage measurements can be made using TRACE32® trace tools with little or no code instrumentation required. This approach simplifies and accelerates integration and system tests in particular and can complement traditional unit testing tools for faster and more efficient safety certifications.

Introduction

Code coverage measurement is a requirement for certification to evaluate the completeness of test cases and to demonstrate that there is no un-intended functionality. Test cases for the verification of code coverage can be executed in different test phases, the unit test, the integration test and the system test.

When using traditional test tools, integration and system tests often generate considerable time and personnel expenditure due to the necessary code instrumentation, which can be reduced dramatically with trace-based code coverage measurements using TRACE32® tools.

The combination of traditional test tools and their code coverage measurement capabilities, which can play to their strengths particularly in unit testing, and TRACE32® with its advantages in integration and system testing, together offer optimum customer benefits.



address	tree	coverage	decision	0% 50% 10
P:FFFF02E8--FFFF039F	⊟ \unittest1	stmt+dc	100.000%	
P:FFFF02E8--FFFF0387	⊞ run_test	stmt+dc	100.000%	
P:FFFF0388--FFFF039F	⊞ main	stmt+dc	100.000%	
P:FFFF03A0--FFFF08E3	⊟ \control_flow	incomplete	88.392%	
P:FFFF03A0--FFFF047B	⊞ func0	stmt+dc	100.000%	
P:FFFF047C--FFFF0553	⊞ func1	stmt+dc	100.000%	
P:FFFF0554--FFFF062F	⊞ func2	stmt+dc	100.000%	
P:FFFF0630--FFFF0753	⊞ func3	incomplete	72.000%	
P:FFFF0754--FFFF0873	⊞ func4	incomplete	75.000%	
P:FFFF0874--FFFF08AB	⊞ do_something0	stmt+dc	100.000%	
P:FFFF08AC--FFFF08E3	⊞ do_something1	stmt+dc	100.000%	

address	tree	coverage	decision	0% 50% 10
P:FFFF02E8--FFFF042F	⊟ \unittest2	stmt+dc	100.000%	
P:FFFF02E8--FFFF0417	⊞ run_test	stmt+dc	100.000%	
P:FFFF0418--FFFF042F	⊞ main	stmt+dc	100.000%	
P:FFFF0430--FFFF0B7B	⊟ \control_flow	incomplete	85.714%	
P:FFFF0430--FFFF0553	⊞ func3	incomplete	52.000%	
P:FFFF0554--FFFF0673	⊞ func4	incomplete	54.166%	
P:FFFF0674--FFFF0797	⊞ func5	stmt+dc	100.000%	
P:FFFF0798--FFFF08BF	⊞ func6	stmt+dc	100.000%	
P:FFFF08C0--FFFF09E3	⊞ func7	stmt+dc	100.000%	
P:FFFF09E4--FFFF0B0B	⊞ func8	stmt+dc	100.000%	
P:FFFF0B0C--FFFF0B43	⊞ do_something0	stmt+dc	100.000%	
P:FFFF0B44--FFFF0B7B	⊞ do_something1	stmt+dc	100.000%	

Figure 1. Code Coverage Measurements with TRACE32®.

Challenges of Traditional Code Coverage Approaches

Traditionally, Code Coverage needs instrumentation of full source code or object code to capture the program flow. In the case of source code instrumentation, the instrumented code is compiled before running it on the target or in a simulator. In the case of object code instrumentation, the process is similar, but instrumentation is done after compilation. In both cases the program flow is acquired via a functional interface and analyzed in the Code Coverage Tool after that. Figure 2 shows the workflow.

While the described traditional approach works fine in unit testing, it brings up a couple of typical issues in integration and system tests: The first challenge is the larger code size caused by the instrumentation, which can lead to the executable no longer fitting into the target's memory. Furthermore, developers suffer from larger RAM consumption and a longer execution time due to the code overhead.

The biggest problem, however, is that this approach no longer works for real-time applications in integration testing and, at the latest, system testing, because the code instrumentation no longer provides real-time conditions.

To solve these problems, the code coverage measurements are typically divided into several parts, in each of which only a part of the code is instrumented. The results are then merged. This means that several build and test runs are necessary, which in practice can take days or even weeks depending on the complexity of the application.

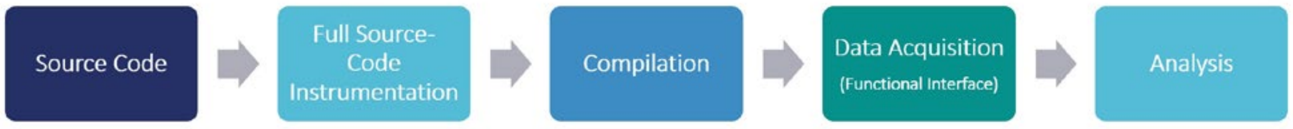

Source Code
Full Source-Code Instrumentation
Compilation
Data Acquisition (Functional Interface)
Analysis

Figure 2. Workflow of traditional Code Coverage Measurements.

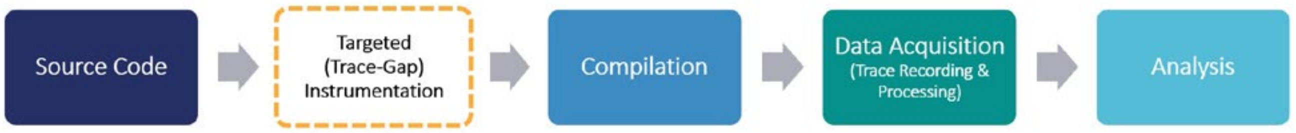

Source Code
Targeted (Trace-Gap) Instrumentation
Compilation
Data Acquisition (Trace Recording & Processing)
Analysis

Figure 3. Workflow of TRACE32® Trace-based Code Coverage Measurements.

Advantages of TRACE32® Code Coverage Approach in Integration and System Testing

When using Lauterbach‘s TRACE32® tools for realtime trace, the data required for code coverage measurements is obtained directly from the recorded trace data and analyzed in the TRACE32® PowerView software. Code instrumentation is either not necessary at all, or only to a very limited extent using so named targeted instrumentation.

Targeted instrumentation covers any trace gaps and is only necessary in the code coverage metrics MC/DC, decision coverage and condition coverage in connection with conditional instructions, for example. For other metrics such as statement coverage, function coverage and call coverage, no code instrumentation is necessary at all.

The uninstrumented or targeted instrumented code is then compiled and executed on the target or in the simulator. The program flow is recorded by the TRACE32® tools via the trace interface and then analyzed in the TRACE32® PowerView software or in external code coverage tools (Figure 3).

The advantages in integration and system testing are obvious: there is only a small increase in code size, if any, and additional data memory (RAM) is not required at all. The biggest advantage, however, is that code coverage measurements can also be carried out far more efficiently for real-time applications: You need fewer build and test runs for a complete code coverage measurement. This saves time and effort.

In practice, it is often the case that the code fully instrumented by traditional test tools no longer fits into the memory or no longer meets the time specifications, or both. This problem is then solved by alternately instrumenting only parts of the code, in fact such large parts that the time specifications (just now) are met and the code (just now) fits into the memory. In many practical examples, one half is instrumented alternately, which means, that two build and test runs must be carried out and at the end the two code coverage measurements have to be merged (Figure 4, above) .

With the TRACE32® trace-based code coverage measurement, one build and test run is usually sufficient due to the minimally invasive instrumentation, which leads to a saving of 50% of the total effort in our example (Figure 4, below). When you consider that these tests often take days or weeks to complete, it is easy to see the relevance of the savings.

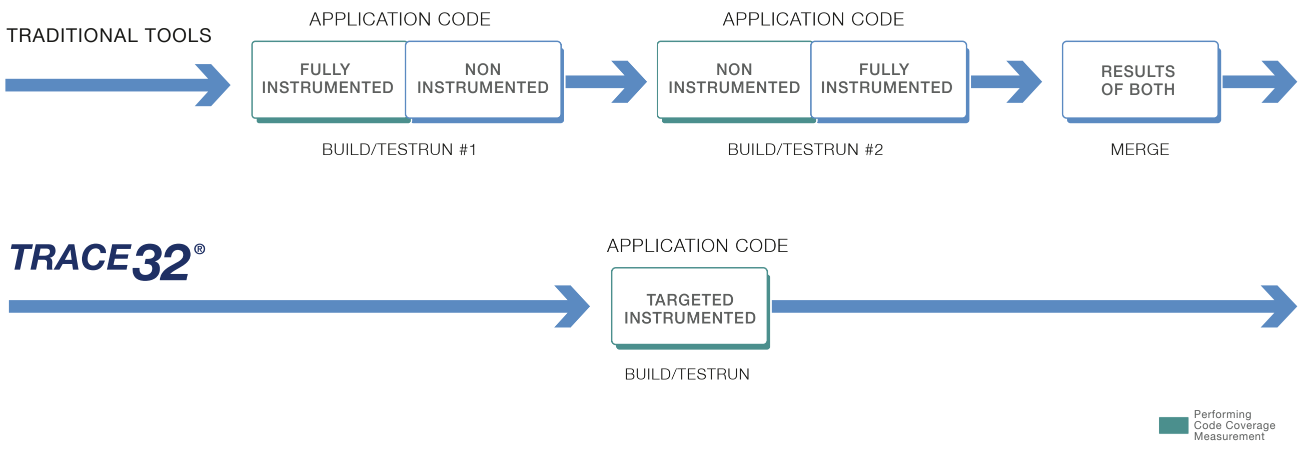


Figure 4. TRACE32® Trace-based Code Coverage Measurements vs. Use of Traditional Tools in System Tests (Example).

Merging Code Coverage Measurements

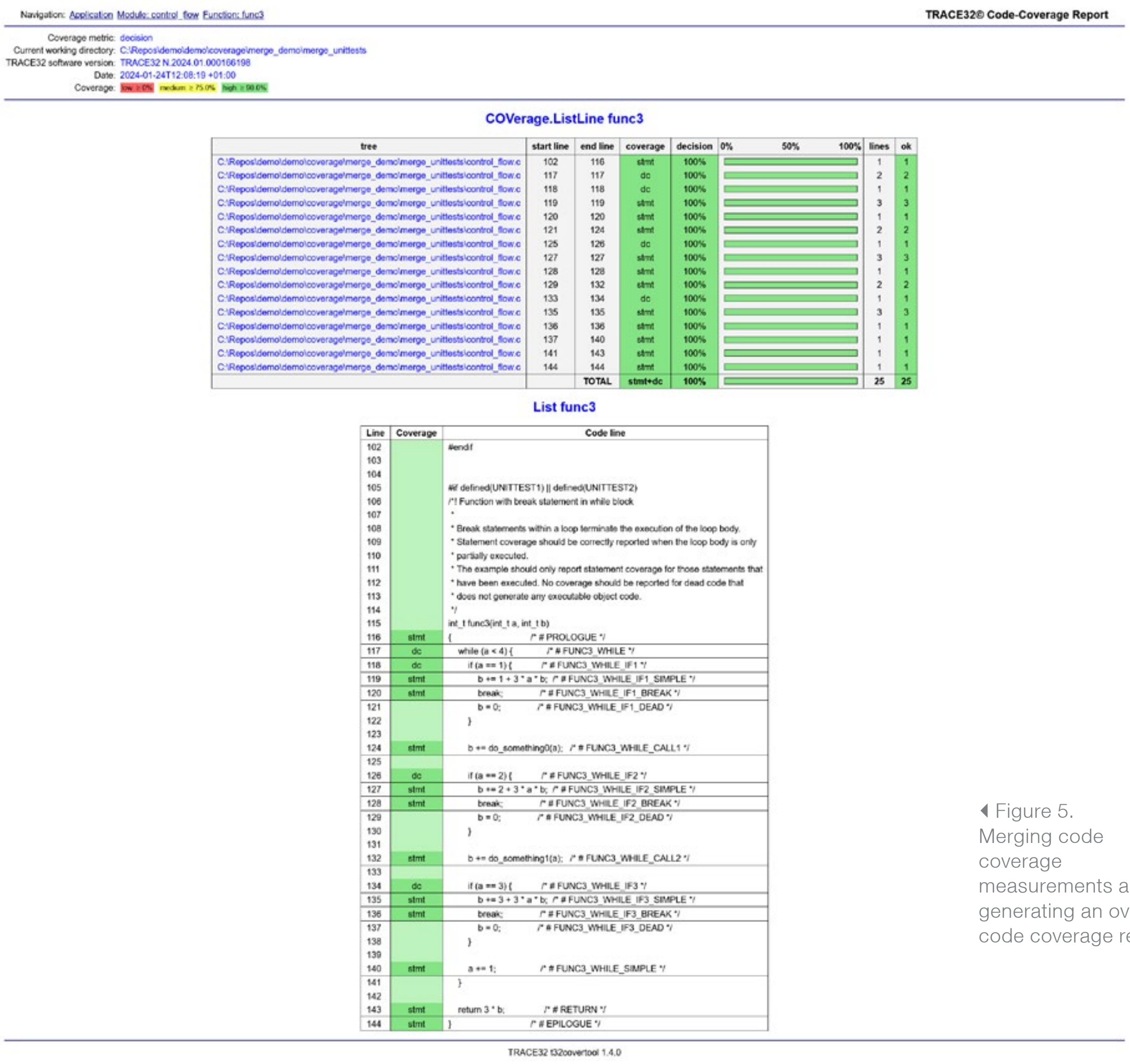
In the application example that runs on different MCUs and contains specific code parts for each of them, you cannot run both test scenarios at the same time. Each test scenario will therefore only show a certain amount of code coverage because the part for the other MCU will not be executed.

With our TRACE32® Merge Tool, several code coverage reports can be merged to generate an overall code coverage report (Figure 5). It consolidates the results of multiple code coverage test runs performed at different times, with different builds or – like in our example – different target configurations.

Conclusion

Lauterbach's TRACE32® debug and trace tools enable trace-based, minimally invasive code coverage measurements in integration and system testing. TRACE32® tools provide code coverage measurements that are closer to the production code than with traditional tools. Embedded test engineers therefore require fewer builds, fewer test runs and can significantly reduce their overall effort compared to using traditional methods only.

In conclusion, the combination of traditional test tools and their code coverage measurement capabilities for unit testing and TRACE32® for integration and system testing, provide the best possible customer experience for efficient and time saving code coverage measurements.



Navigation: Application Module: control_flow Function: func3

TRACE32® Code-Coverage Report

Coverage metric:	decision
Current working directory:	C:\Repos\demo\demo\coverage\merge_demo\merge_unittests
TRACE32 software version:	TRACE32 N.2024.01.000166198
Date:	2024-01-24T12:08:19 +01:00
Coverage:	low ≥ 0% medium ≥ 75.0% high ≥ 90.0%

COVerage.ListLine func3

tree	start line	end line	coverage	decision	0% 50% 100%	lines	ok
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	102	116	stmt	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	117	117	dc	100%	🟩	2	2
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	118	118	dc	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	119	119	stmt	100%	🟩	3	3
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	120	120	stmt	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	121	124	stmt	100%	🟩	2	2
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	125	126	dc	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	127	127	stmt	100%	🟩	3	3
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	128	128	stmt	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	129	132	stmt	100%	🟩	2	2
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	133	134	dc	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	135	135	stmt	100%	🟩	3	3
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	136	136	stmt	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	137	140	stmt	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	141	143	stmt	100%	🟩	1	1
C:\Repos\demo\demo\coverage\merge_demo\merge_unittests\control_flow.c	144	144	stmt	100%	🟩	1	1
		TOTAL	stmt+dc	100%	🟩	25	25

List func3

Line	Coverage	Code line
102		`#endif`
103		
104		
105		`#if defined(UNITTEST1) || defined(UNITTEST2)`
106		`/*! Function with break statement in while block`
107		` *`
108		` * Break statements within a loop terminate the execution of the loop body.`
109		` * Statement coverage should be correctly reported when the loop body is only`
110		` * partially executed.`
111		` * The example should only report statement coverage for those statements that`
112		` * have been executed. No coverage should be reported for dead code that`
113		` * does not generate any executable object code.`
114		` */`
115		`int_t func3(int_t a, int_t b)`
116	stmt	`{                         /* # PROLOGUE */`
117	dc	`  while (a < 4) {         /* # FUNC3_WHILE */`
118	dc	`    if (a == 1) {         /* # FUNC3_WHILE_IF1 */`
119	stmt	`      b += 1 + 3 * a * b; /* # FUNC3_WHILE_IF1_SIMPLE */`
120	stmt	`      break;              /* # FUNC3_WHILE_IF1_BREAK */`
121		`      b = 0;              /* # FUNC3_WHILE_IF1_DEAD */`
122		`    }`
123		
124	stmt	`    b += do_something0(a);  /* # FUNC3_WHILE_CALL1 */`
125		
126	dc	`    if (a == 2) {         /* # FUNC3_WHILE_IF2 */`
127	stmt	`      b += 2 + 3 * a * b; /* # FUNC3_WHILE_IF2_SIMPLE */`
128	stmt	`      break;              /* # FUNC3_WHILE_IF2_BREAK */`
129		`      b = 0;              /* # FUNC3_WHILE_IF2_DEAD */`
130		`    }`
131		
132	stmt	`    b += do_something1(a);  /* # FUNC3_WHILE_CALL2 */`
133		
134	dc	`    if (a == 3) {         /* # FUNC3_WHILE_IF3 */`
135	stmt	`      b += 3 + 3 * a * b; /* # FUNC3_WHILE_IF3_SIMPLE */`
136	stmt	`      break;              /* # FUNC3_WHILE_IF3_BREAK */`
137		`      b = 0;              /* # FUNC3_WHILE_IF3_DEAD */`
138		`    }`
139		
140	stmt	`    a += 1;               /* # FUNC3_WHILE_SIMPLE */`
141		`  }`
142		
143	stmt	`  return 3 * b;           /* # RETURN */`
144	stmt	`}                         /* # EPILOGUE */`

TRACE32 t32covertool 1.4.0

◂ Figure 5. Merging code coverage measurements and generating an overall code coverage report.