Nine Tips for the right Code Coverage Measurement

By Klaus Lambertz

Measuring code coverage is increasingly important for embedded systems but requires some experience. This is because there are a few hurdles to overcome, especially with small targets. However, with the right approaches and suitable tools, measuring test coverage is possible without excessive effort. Nine practical tips help you get started.

Measuring test coverage, also known as code coverage, is becoming increasingly important for embedded systems. In many cases, these devices are critical to safety or business. Processes are based on IoT devices, patients rely on working pacemakers and intelligent insulin pumps, automotive and aviation is no longer conceivable without embedded software. This list could be continued almost endlessly. As the criticality of the various devices increases, so do the requirements that must be met in terms of safety, security, and functionality. Safety standards take this into account and explicitly require test coverage to be recorded as part of product verification.
Especially for beginners, measuring code coverage seems extremely complex and time-consuming. However, if you pay attention to a few basic aspects, things quickly become easier.

1. Set expectations

Often it is not entirely clear what to expect from code coverage. Coverage measurements are not intended to directly improve the code. Their purpose is to determine whether the code was fully tested and whether the test cases were complete. Thus, the code coverage rather serves the improvement of the testing and finally for proving that sufficient tests have been performed.

2. Determination of the required test coverage level

There are numerous different test coverage levels. It can be stated: the more rigorous and detailed a test coverage level is, the greater is the required effort and the costs to achieve it.
Unfortunately, only few standards, like DO-178C in aviation, IEC 61508 for the functional safety of electrical/electronic, programmable electronic safety-related systems, or ISO 26262 for automotive, give specific guidance on the required code coverage level.
Depending on the Safety Integrity Level (SIL) for IEC 61508 or Automotive Safety Level (ASIL) for ISO 26262, statement coverage, branch coverage or MC/DC (Modified Condition/Decision Coverage) is required (for more information about Code Coverage Levels see separate explanation). The following applies to all standards: the more dangerous the effects of possible errors, the stricter the required coverage level.

This can be seen in the following table from the ISO 26262 standard. The highest safety level ASIL D requires the highest coverage level MC/DC. In the table ++ stands for highly recommended, + stands for recommended. Here we can say that complete Modified Condition/Decision Coverage automatically implies complete Branch Coverage, and complete Branch Coverage automatically implies complete Statement Coverage.

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

Figure 1: ISO 26262 Table 12 - Structural coverage metrics at the software unit level (Source: INTERNATIONAL STANDARD ISO 26262-6)

For software for which there are no regulations through standards, it is also important to determine in advance which code need to be tested according to which criteria. It may be helpful to use the above standards as a guide and to align the scope of testing with the criticality of the system.

3. Determination of the scope of tests

Complete code coverage initially means that 100 percent of the specified test level has been achieved. However, this is usually only necessary in the safety-critical area.
Aiming for 100 percent code coverage just to meet this criterion is often not helpful. It is important to know why you are running certain test cases. If you don't know, there is high risk of redundant tests that run through already tested codes again and again. This higher effort does not pay off, because these additional tests do not provide any new insights. By the way, using code coverage analyzers is a good way to avoid such redundant tests.
In non-critical application areas, testing should be neither "too little" nor "complete", but "sufficient".

4. Determination of supported programming languages

There are numerous code coverage analyzers: from free tools such as the GNU Coverage Testing Tool Gcov, which is part of the GNU Compiler Collection (GCC), to comprehensive solutions such as Testwell CTC++ from Verifysoft Technology, which offers besides support for all code coverage levels also support for various languages. If possible, all languages used in the company should be able to be processed with a single coverage tool. This way, developers and testers can work in a uniform interface and do not have to familiarize themselves with different tools. For embedded software development, the tool should cover at least C, C++, and Java.

5. Save memory space

When measuring code coverage on small targets the limited memory space is often a hurdle. The reason is the need of instrumentation of the code: in order to measure the coverage, the code coverage tool inserts counters into the code. In addition, a library must be implemented on the target to be tested, which, among other things, handles data transfer to a host.

The counters are usually stored as global arrays in the data memory. Especially with very tightly dimensioned targets this can lead to problems. The remedy is a code coverage analyzer specialized for embedded devices. Such tools instrument as sparingly as possible. In the case that the instrumentation overhead is still to high, the code coverage can be analyzed separately for individual code parts.

A third method is to reduce the size of the counters. Usually, the size of the counters are 32 bits. Special arrangements reduce the counter size to 16 or 8 bits. However, care must be taken to avoid overflows. The bit-cov-measure of Testwell CTC++ is such a special arrangement for small embedded targets and microprocessors.

6. Check possible effects on the processor

To save costs, not only the memory of many embedded devices is small, but also the processor has often limitations. However, instrumentation also affects the processor. Therefore, it can happen that the defined timing is exceeded, which can lead to faulty program runs under certain circumstances.

Unfortunately, it cannot be reliably predicted in advance whether problems will occur due to the slightly higher processor load. Possible effects will only become apparent in the respective project. For this reason, code coverage analyzers should also be selected with regard to their impact on timing. If necessary, the use of smaller counters (as mentioned above) or partial instrumentation can remedy the situation.

It is also possible to run the tests once with coverage analysis and once without it to determine whether the coverage analysis has led to changes of the program behavior.

7. Check the possibility of integration into your toolchain

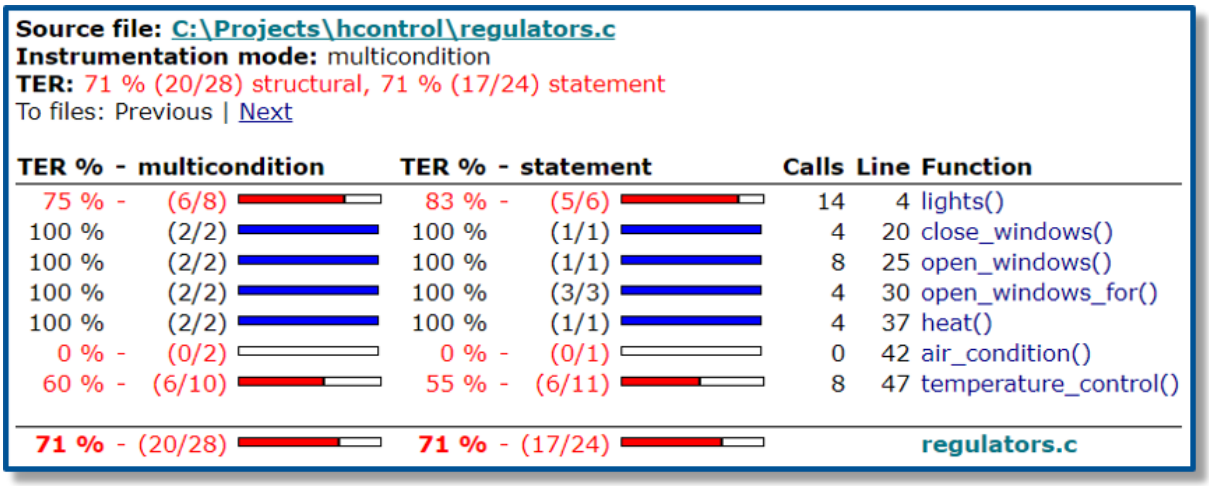
More and more tests are being automated. This makes sense, because manual interventions always represent a potential source of errors and also cause significant costs. The automation of the test on the build system is required especially in agile development with Continuous Integration/Continuous Deployment (CI/CD). However, this requires that the code coverage analyzer used can be integrated into the tool chain and the build system. In addition, the coverage tool should also be independent of the compiler used.

Especially in large development projects, automated tests and the automatic capture of code coverage can hardly be dispensed with. Therefore, it is important that code coverage tools can be easily and smoothly integrated into toolchains. On the other hand, sophisticated dashboards and frontends are rather negligible for automated testing.

8. Verification of the comprehensibility of reports

Code coverage generates large amounts of data. Each area of testing - unit tests, module tests, functional tests, etc. - provides its own data on code coverage. To obtain a complete picture, this information should be aggregated. It is up to the Code Coverage Analyzer to prepare this information in a meaningful and interpretable way. In addition, this data is important in the context of certifications or audits that may be required.

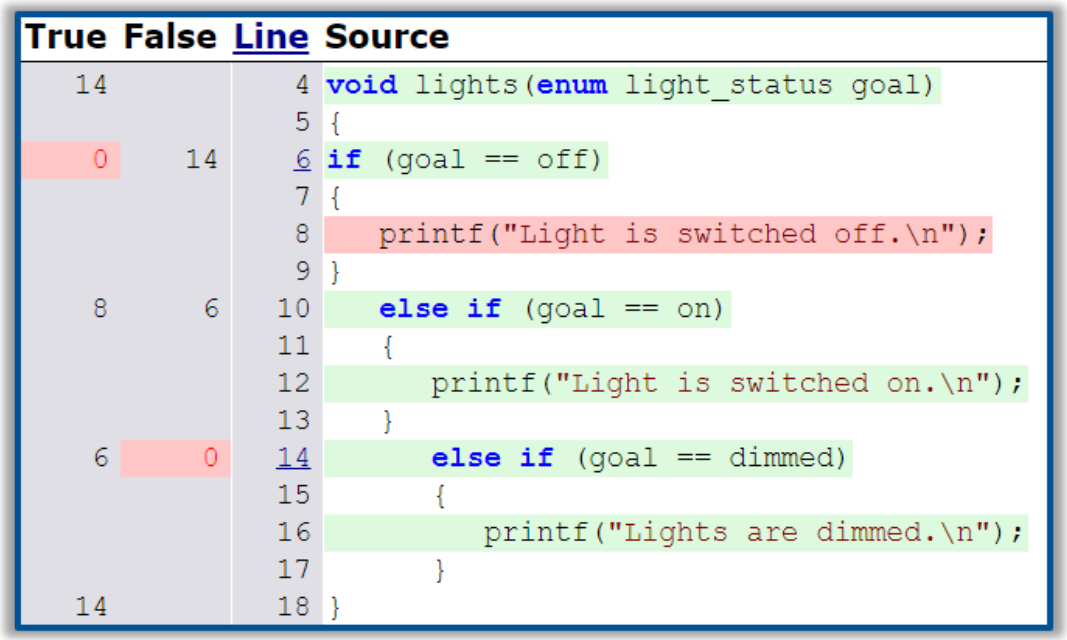
It is important that the information provided by the coverage tool is available in both machine-readable form for further automatic processing (e.g. XML), as well as in human readable form (e.g. HTML) for a quick and good overview.



Source file: C:\Projects\hcontrol\regulators.c
Instrumentation mode: multicondition
TER: 71 % (20/28) structural, 71 % (17/24) statement
To files: Previous | Next

TER % - multicondition		TER % - statement		Calls	Line	Function
75 % -	(6/8)	83 % -	(5/6)	14	4	lights()
100 %	(2/2)	100 %	(1/1)	4	20	close_windows()
100 %	(2/2)	100 %	(1/1)	8	25	open_windows()
100 %	(2/2)	100 %	(3/3)	4	30	open_windows_for()
100 %	(2/2)	100 %	(1/1)	4	37	heat()
0 % -	(0/2)	0 % -	(0/1)	0	42	air_condition()
60 % -	(6/10)	55 % -	(6/11)	8	47	temperature_control()
71 % -	(20/28)	**71 %** -	(17/24)			**regulators.c**

Figure 2: The HTML-report (function summary) of the code coverage analyzer Testwell CTC++ shows at one glance which functions are covered by tests. Entirely covered functions are shown in blue. For functions shown in red color, coverage is less than 100%. For example the function “lights()” has 75% multicondition coverage and 83% statement coverage (TER stands for “test effectiveness ratio”).



True	False	Line	Source
14		4	`void lights(enum light_status goal)`
		5	`{`
0	14	6	`if (goal == off)`
		7	`{`
		8	`printf("Light is switched off.\n");`
		9	`}`
8	6	10	`else if (goal == on)`
		11	`{`
		12	`printf("Light is switched on.\n");`
		13	`}`
6	0	14	`else if (goal == dimmed)`
		15	`{`
		16	`printf("Lights are dimmed.\n");`
		17	`}`
14		18	`}`

Figure 3: The Execution Profile of Testwell CTC++ shows detailed information of the covered items within the source code

9. Check for suitability of the code coverage tool for the development of safety-critical software

For the development of safety-critical software, it is self-evident that the code coverage tool must support the coverage levels required by the respective standard.
Furthermore, standards require the qualification of the entire toolchain. The availability of a tool qualification kit for the code coverage analyzer simplifies this work considerably. In this context, it is also important whether the tool provider offers appropriate advice here.

Conclusion

Measuring code coverage is not trivial, but it is not too complex either. Thus, there is hardly any good reason to renounce code coverage - even in the non-critical area. Test coverage helps on several levels: tests and test procedures can be optimized, resulting in significant cost and time benefits. Furthermore, code coverage ensures that the products have been adequately tested and that they reach the customers in a certain minimum quality.
Embedded systems providers should not repeat a serious mistake made by the IT industry: deliver “banana software” that matures at customer’s site. Users of embedded devices will not accept this - especially not when it comes “critical” products like medical devices or ECUs in the automotive sector.

Further Explanations about Code Coverage Levels

Function Coverage

Function Coverage measures whether all functions of the program were called. The Function Coverage is the “weakest” of the usual test coverage levels. As it ignores the inner working of the software, its usefulness is quite low.

Statement Coverage

Statement coverage determines which statements were executed by tests. This can be used, for example, to detect dead code. It also shows whether tests are available for all statements.

Decision Coverage / Branch Coverage

At this coverage level, each decision must be tested at least once as true and once as false. For normal if statements, this corresponds to branch coverage, where each branch must have been executed.

Condition Coverage

Condition coverage considers compound decisions in detail. For decisions that consist of multiple atomic conditions composed via Boolean operators, each of these conditions must be tested individually as “true” and as “false”.

Multicondition Coverage and Modified Condition/Decision Coverage (MC/DC)

For multicondition coverage, all possible true-false combinations must be checked for composite decisions. In the case of multiple conditions within a decision, this requires a mostly impracticably high number of test cases. Therefore, in practice and in standards, Modified Condition/Decision Coverage (MC/DC) is relevant, where the number of test cases is reduced, and the informative value of the test coverage remains sufficiently high. For MC/DC all atomic conditions of a composite condition are used. For each of the atomic conditions a test case pair is tested, which leads to the change of the overall result of the composite condition, but only the truth value of the considered atomic condition changes, whereas the truth value of the other atomic conditions must remain constant.

About the Author

Klaus Lambertz is Chief Executive Officer of Verifysoft Technology GmbH www.verifysoft.com in Offenburg/Germany. Before he founded the company in 2003, had sales and management positions with different software testing solution providers. Klaus graduated in studies of Economics, Marketing and Foreign Trade in Cologne (Germany) and Paris (France).