

Code Coverage on small Targets

Even on embedded devices, software has to be tested sufficiently before the product can be launched on the market. However, such a monitoring in the testing progress, required by many safety standards, is not quite trivial at this point. Often, target resources are limited and memory and processor could easily reach their boundaries. But with some creativity it is even possible to capture and proof Code Coverage in the embedded area.

By Klaus Lambertz, CEO of Verifysoft Technology GmbH

Requirements to embedded devices are rising more and more. Above all, in relation to the keyword IoT (Internet of Things), embedded systems are taking over an important, if not even critical position within companies' business processes. Concerning their study from June 2018, the market research company IDC expects that the market for IoT technologies, will reach a worldwide volume of 1.2 trillion Dollar until the year 2020, with a growth rate of more than 13 percent. In that context, embedded devices are converting into a critical factor in every business sector. In addition to safety issues, security issues are becoming important.

In the medical sector, aerospace, automotive and railway industry, strict standards are regulating testing and certification of embedded systems to allow their general usage. Beyond the traditional safety-focused sectors, there is a lot of space to catch up. Regarding an inquiry from Gartner in spring 2018, nearly 20 percent of the asked companies have already suffered an IoT-based attack within the past 3 years. According to Gartner, one reason might be related to the slow development of security standards, caused by a shortage of Security by Design.

Code Coverage is often mandatory

To guarantee a safe embedded device, testing represents an important element. Therefore, within the safety-critical software development, standards stipulate for detailed requirements in terms of testing methods and Code Coverage. Normally, relevant standards like DO-178C in aerospace or ISO 26262 in the automotive sector assume a *Statement and Branch Coverage testing at least*. But also higher testing levels like Modified Condition/Decision Coverage (MC/DC) are often required. Generally it is said: the higher the software's safety requirements, the higher the required Code Coverage level. Nonetheless, not every Code Coverage level seems to be useful for every type of scenario, since each of them delivers different results and outcomes:

- Function Coverage ignores the internal steps of a function. Hence, its utility seems quite low.
- Statement Coverage detects which instructions are already executed by testing. This method is useful to identify dead code or instructions which have not been tested yet.

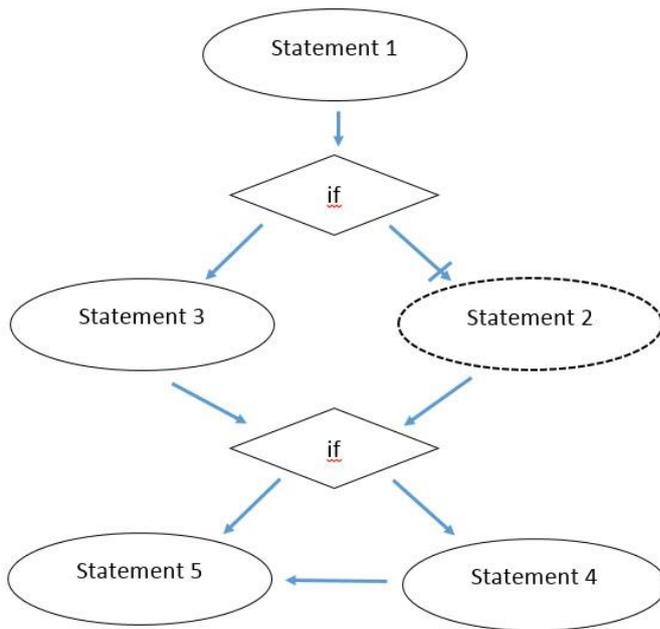


Figure. 1: Statement Coverage indicates how many instructions have already been executed. In the given example, six out of seven instructions are covered; the Code Coverage yields a result of 85 percent.

- Branch Coverage identifies if every branch within the program was taken. Thus, Branch Coverage is more accurate than Statement Coverage and represents a realistic minimum requirement for testing in comparison to its effort.
- MC/DC (Modified Condition/Decision Coverage) represents the highest Code Coverage level required by the standards. As testing every possible combination of available conditions would cause an enormous effort, MC/DC tries to minimize this problematic. In order to realize this objective, the testing method considers every atomic condition of a compounded condition. For each atomic condition, test cases are tested pairwise, leading to a change in the overall result of the compounded condition by just changing the boolean value of the considered atomic condition. In this case it is important that the boolean values of all other possible atomic conditions doesn't change.

In order to determine Code Coverage, special instruments are used in practice. Before the code is taken over to the compiler, a so-called Code Coverage Analyzer is used to complete the code with counters of the desired Coverage level. This procedure is called code instrumentation. An easy tool for such an instrumentation is for example the GNU testing tool gcov. Through the gcc-option `-ftest-coverage`, the code is instrumented in order to detect how often every line of code is executed. The option `-fprofile-arcs` instruments the branches. To gain some first insights within the area of Code Coverage or to manage small projects, the tool represent a suitable choice. Its disadvantage embodies - aside from a lack of Code Coverage levels which should be used for higher testing requirements - the preparation and interpretation of the gained information. In such a case, commercial tools are definitely the superior option.

CTC++ Coverage Report - Functions Summary #1/1

[Directory Summary](#) | [Files Summary](#) | [Functions Summary](#) | [Untested Code](#) | [Execution Profile](#)
 To directories: [First](#) | [Previous](#) | [Next](#) | [Last](#) | [Index](#) | [No Index](#)

Directory: C:\Projects\hcontrol

TER: 71 % (44/62) structural, 78 % (49/63) statement

Source file: C:\Projects\hcontrol\regulators.c

Instrumentation mode: multicondition **Reduced to:** MC/DC coverage

TER: 71 % (20/28) structural, 71 % (17/24) statement

To files: [Previous](#) | [Next](#)

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

Figure 2: Summary of a Coverage Report of the Testwell CTC++ tool.

During the function "lights()", 5 out of 6 statements were executed. Accordingly, the Statement Coverage (TER = Test Effectiveness Ratio) yields a result of 83%. The structural Code Coverage, in this case MC/DC, is 75%.

Hits/True False [Line](#) [Source](#)

Hits/True	False	Line	Source
		1	/* File io.c -----
		2	#include <stdio.h>
		3	#include "io.h"
		4	/* Prompt for an unsigned int value and return it */
Top			
10		5	unsigned io_ask()
		6	{
		7	unsigned val;
		8	int amount;
		9	
		10	printf("Enter a number (0 for stop program): ");
0	10	11	if ((amount = scanf("%u", &val)) <= 0) {
		12	val = 0; /* on 'non sense' input force 0 */
		13	}
10		14	return val;
		15	}

Figure 3: Testwell CTC++ Report with Coverage information within the source code. The condition in line 11 was executed 10 times as “false” and never as “true”.

To reach a complete Code Coverage, this should be done afterwards.

Instrumentation extends the code

Professional solutions are working according to the same principle like gcov by instrumenting the software code. Normally, the counters are organized as global arrays. When and how these counters are modified depends on the required Code Coverage level. The following example of a while-loop written in C illustrates the consequences of an instrumentation:

```
while (! b == 0 )
{
    r = a % b;
    a = b;
    b = r;
}
result = a;
```

Through the instrumentation – in this case implemented by the Code Coverage tool Testwell CTC++ - the following structure arises:

```
while ( (( ! b == 0 ) ? (ctc_t[23]++, 1) : (ctc_f[23]++, 0)) )
{
    r = a % b;
    a = b;
    b = r;
}
result = a ;
```

Resource shortage

Through the instrumentation process, the existing code is growing. Since the necessary arrays are located on the memory card, both on the RAM and on the ROM additional capacities are needed. Furthermore, the instrumentation influences the execution time. In case of server or computer applications, such an effect might be neglected. However, in relation to embedded devices, it becomes increasingly important, since due to cost issues hardware resources are calculated extremely scarce. Under such circumstances, it is recommendable to use a Code Coverage Analyzer with a comparatively low instrumentation overhead, as otherwise, counters may easily exceed the available memory. This issue is getting highly important if advanced Code Coverage levels like MC/DC are necessary.

7. Code Instrumentation

Verifysoft
TECHNOLOGY

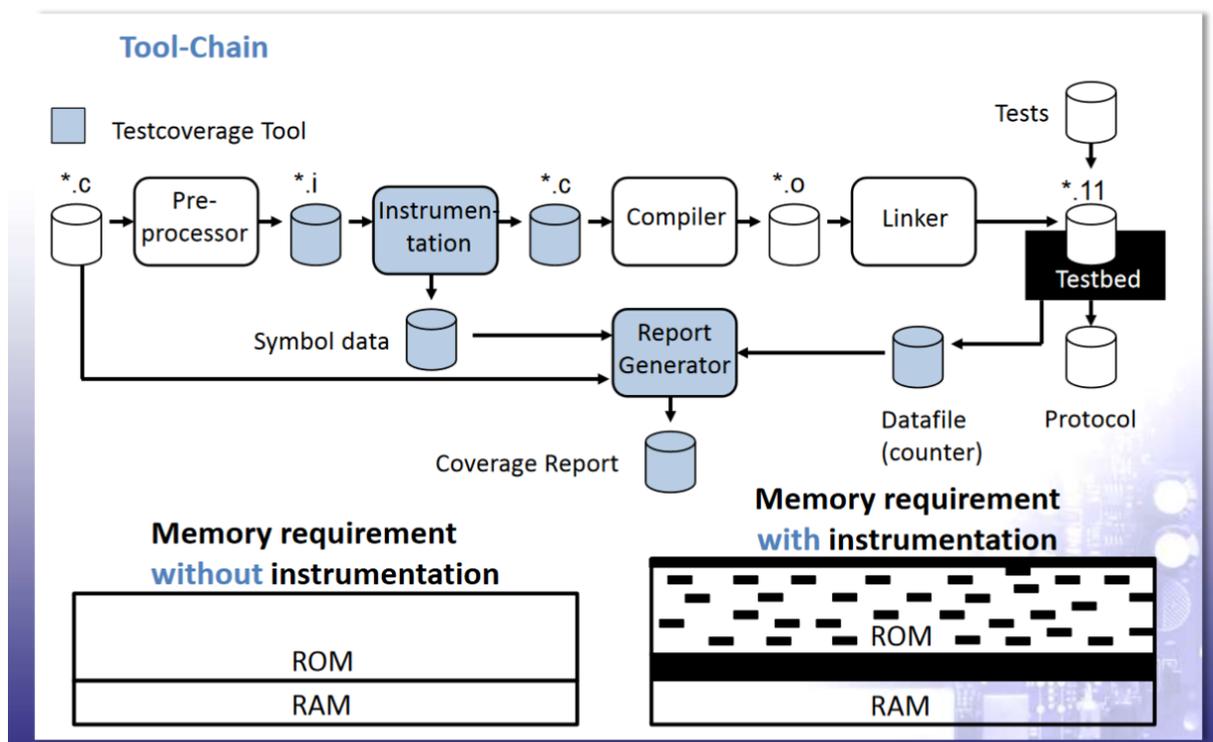


Figure 4: Due to the code instrumentation, resources requirements are growing.

If the Code Coverage tool delivers a too high instrumentation overhead, a partial RAM instrumentation might overcome such an obstacle. In this case, only small parts of the program are instrumented and tested. One after another, the test is repeated with every single part of the program and the obtained data is added to an overall picture afterwards. Thereby, the Code Coverage for the entire program could be identified. Another approach on small targets is to limit the size of the counters. Usually, Code Coverage tools are working with 32 Bit counters. These could be reduced -at least theoretically - to a size of 16 or 8 Bit. However, such a method has to be applied with caution, since under certain circumstances counters may run over. Thus, the gathered data has to be interpreted with highest carefulness. In extreme cases, counters could be even reduced to single Bits. This so-called Bit Coverage shows if a certain piece of code has been tested, but not how many times it has been visited.

ROM-Usage	
Without Instrumentation	60 Byte
Function Coverage	67 Byte
Branch Coverage	118 Byte
Condition Coverage	285 Byte
Additional RAM-Usage without Bit-Coverage (32- Bit-counter)	
Function Coverage	4 Byte
Branch Coverage	16 Byte
Condition Coverage	28 Byte
Additional RAM-Usage using Bit Coverage	
Function Coverage	1 Bit
Branch Coverage	4 Bit
Condition Coverage	7 Bit

Figure 5: RAM requirements can be minimized significantly due to Testwell's CTC++ offered Bit Coverage.

Also the chosen Code Coverage level affects the necessary RAM. On the other hand, the additional needed space on the ROM can be hardly limited. In order to capture the code coverage metrics, a small library is necessary, transferring the counter status to the host. In addition to its charge on the memory, the instrumentation is also affecting the processor within the target. Therefore it might be possible that a predefined time limit could not be maintained any longer. Especially if the CPU is working close to its limit, the possibility of faulty processes to occur is extremely high. Exclusively the bus communication is highly affected by such a problematic. Because of this, the tester should carefully monitor the whole testing procedure and check the obtained results. Nevertheless, advanced Code Coverage tools are able to keep both the instrumentation's memory requirements and changes of the runtime behavior relatively low.

Conclusion

The importance of software quality will further increase in the future. Consequently the significance of both testing and Code Coverage will rise as well. Even though not every standard would require a 100 percent MC/DC Coverage for each kind of software, it will be only a matter of time when standardization boards and industry associations will rise their requirements for non-critical applications as well. Furthermore, improved testing results are also in the interest of the producer itself, as faulty products may cause high follow-up costs or damages of the company's reputation. Consequently, the well-known "bananaware", including the idea that the software will just mature with the user, will hardly be accepted by clients in the embedded sector.