Code coverage in automated test of embedded systems

By Jens Braunes, PLS

By means of code coverage, it is not complicated to determine test quality in parallel to software and system tests on real hardware. In any case, a continuous and seamless workflow for the software and system testing, which logically requires open interfaces for efficient tool coupling, is essential for correct and reliable results.

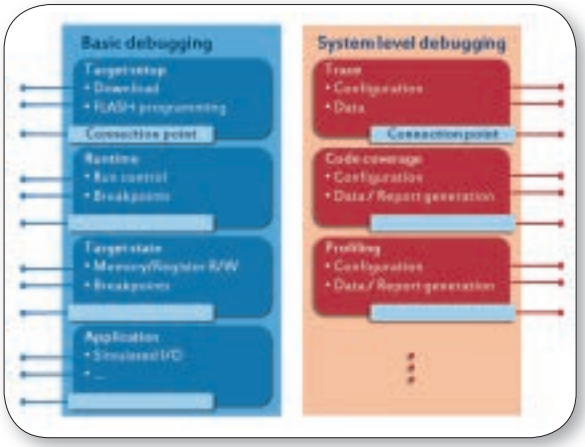


Figure 1. The UDE object model provides access to almost all debugger functions.

■ Today the complexity of SoCs presents many new challenges for developers, especially with regard to software quality and software security. This means for effective software testing, it is not only automatic generation of test results and their evaluation that are essential. To reliably determine the quality of the tests, it is also necessary to measure and document the achieved code coverage.

Software errors can never be completely avoided during the development of complex applications with megabytes of source code. It is therefore all the more important, especially with safety-critical applications, to detect and eliminate these bugs as early as possible using modern methods. Testing is one of the most crucial parts of this process. With good reason, a great deal of attention is given to testing in relevant standards such as ISO 26262 for the automotive industry, EN/IEC 62061 for safety of machinery or DO-178 for the aviation industry.

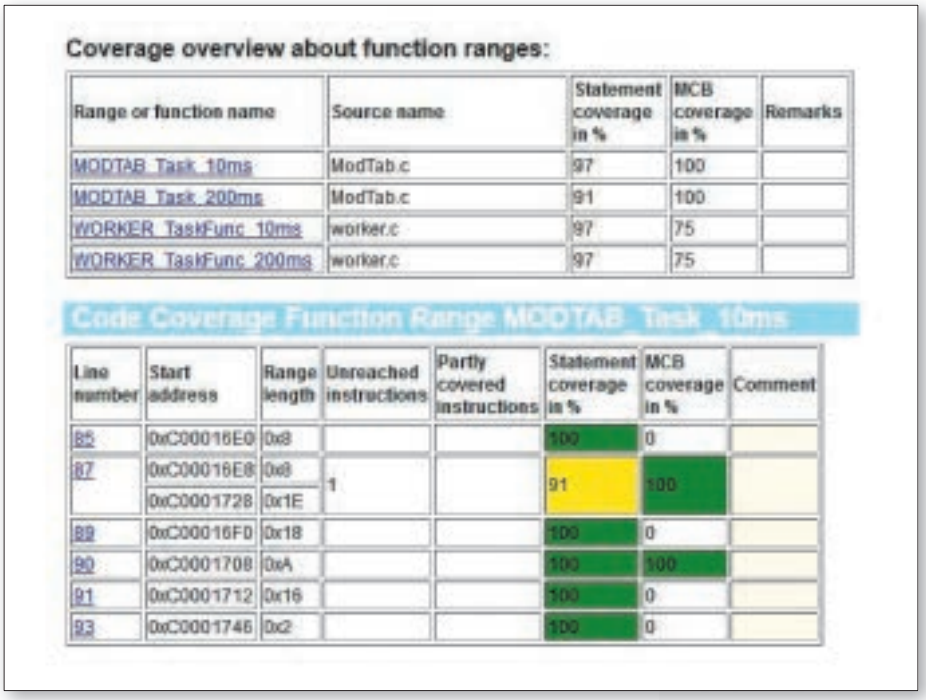
Whether a software solution is finally released for use by the customer or needs reworking by the development department, however, does not only depend on the successful completion of testing. In addition, the quality of the testing performed has to be right. The decisive factor here is test coverage. What is the proportion of the application, module or individual functions, which the respective test has stressed, compared to the proportion, which could not be stressed at all due to an unfavourable choice of test cases? To explore this question more thoroughly, which is extremely important for software quality and security nowadays, in the area of software development code coverage is generally used to assess test quality. Since the standards mentioned at the beginning not only demand the documentation of test results but also the degree of coverage achieved, it is of course sensible to determine the code coverage for the function or module under test in parallel to the test execution.

To calculate code coverage, for example of a function, information is required that can only be gathered from executing the code. In the simplest case, only those pieces of code that are actually executed and those which are not must be recorded. The latter is of course implicitly given if the sources or at least the binary are available. In turn, the statement coverage - meaning how much code is tested by the test cases in relation to the total code - can be directly derived from this information. Program code that is not executed (dead code) can therefore be detected with this comparatively simple method; however, the current test quality requirements of the standards ISO 26262, EN/IEC 62061 or DO-178 are generally not fulfilled.

Branch coverage is significantly more meaningful with regard to test quality. The execution of all possible program branches is used for this. In the case of a simple IF statement, both the TRUE and the FALSE branch have to be executed. Because all pieces of code are implicitly reached, the statement coverage can be directly derived from the result of the branch coverage.

Modified condition/decision coverage (MC/DC) goes another step further in looking at branches. In simple terms, for MC/DC, all individual conditions contained in each composed condition must take every possible outcome in order to be regarded as tested. This ensures that each individual condition affects the overall result, independently of the other individual conditions involved. However, this method proves to be extremely cumbersome in practical use, because an extremely high number of paths have to be considered, especially within loops containing conditional code sequences. In practice, there are now a number of possibilities to obtain code coverage, as follows.

Simulation of executable code on a virtual platform and determining the necessary information for the calculation. Fairly simple data about runtime behaviour can of course be determined from simulations. For this



Coverage overview about function ranges:

Range or function name	Source name	Statement coverage in %	MCB coverage in %	Remarks
MODTAB_Task_10ms	ModTab.c	97	100	
MODTAB_Task_200ms	ModTab.c	91	100	
WORKER_TaskFunc_10ms	worker.c	97	75	
WORKER_TaskFunc_200ms	worker.c	97	75	

Code Coverage Function Range MODTAB_Task_10ms

Line number	Start address	Range length	Unreached instructions	Partly covered instructions	Statement coverage in %	MCB coverage in %	Comment
85	0xC00016E0	0x8			100	0	
87	0xC00016E8	0x8	1		91	100	
	0xC0001728	0x1E					
89	0xC00016F0	0x18			100	0	
90	0xC0001708	0xA			100	100	
91	0xC0001712	0x16			100	0	
93	0xC0001746	0xc2			100	0	

Figure 2. By means of trace, the code coverage achieved can be determined in parallel with the actual testing. The reports generated from this serve as proof of the test quality.

reason, this method is also very widespread with many test tools. A disadvantage, however, is that the actual test is not carried out on the real embedded system; the actual timing behaviour is therefore not taken into consideration.

Instrumentation of the code to be tested and execution on real hardware. Additional test code, which collects information necessary for calculating the code coverage and stores this in the target memory, is inserted in the software for this. However, as a consequence the test code not only influences the runtime behaviour and code size but possibly even the memory layout. This is certainly a crucial point in safety-critical applications or systems with hard real-time requirements.

Execution of original code on real hardware with simultaneous program trace. The target system must provide suitable trace hardware for this including a trace interface to the coverage tool. This method requires absolutely no instrumentation and the timing behaviour of the application to be tested also remains unchanged.

However, suitable high-performance debugger solutions, such as for example the Universal Debug Engine (UDE) from PLS, are crucial for successful test execution and determining the code coverage on real target hardware by means of instrumentation or trace. This is because in the instrumented case, the target memory, which contains the data for the coverage calculation, must at least be read out by the debugger. For the trace-based solution, the debugger even uses the entire trace infrastructure on the chip.

For tests on real hardware, tight coupling between the debugger, which of course enables the actual access to the target system, and the test tool, which takes over test case management and the test documentation, is of essential importance. Only in rare cases do debuggers also offer complete project management for the test. Test tools, on the other hand, often lack suitable possibilities to communicate directly with the target system via the debug interface. Modern high-end debuggers, like the UDE, therefore provide an automation interface for the tool coupling. That interface allows test tools to utilize the debugger functions for controlling the target system and for reading out the target state as well as manipulating it.

In the case of the UDE, this interface is based on the Component Object Model (COM) from Microsoft. For a long period of time, COM has been established as the de facto standard in the world of Windows. Even Microsoft itself offers a large part of its newly added Windows functions via COM interface. The object model of the UDE encompasses almost all functions of the debugger such as flash programming, run control, reading and writing of the target system memory contents, trace data acquisition and analysis, and of course also code coverage. Therefore, via the said interface, with the UDE it is possible to establish a tight coupling to test systems of different vendors and thus build a complete tool chain for testing on real hardware. Furthermore, COM offers the major advantage that it can be used with a very large number of different languages. These include C, C++, C# and other .NET languages as well as scripting languages such as JavaScript, Python, Perl and VB Script or Windows PowerShell. Therefore, the UDE can also be automated by the users very easily and controlled remotely via own scripts without the aid of a test tool.

As is already known, instrumentation of the executed code is not really advisable for obtaining code coverage in safety-critical applications or systems with high real-time requirements. For most of these cases already mentioned trace-based solutions are inevitably used. Also for this the automation interface of UDE offers appropriate functions. For example, if UDE is controlled by a test system the recording of trace for code coverage can be enabled already during the configuration of test tasks within the test tool itself. The other settings for code coverage are subsequently made in the debugger itself, as follows.

Used coverage level: here it is possible to choose between statement coverage and branch coverage. For both levels, the coverage can be calculated solely based on program trace and from debug information of the application.

Accumulation of sequentially performed coverage measurements: if multiple runs are necessary for testing all functions of an application, for example because the trace memory is not sufficiently large for the coverage, an accumulated calculation of the code coverage for all runs makes sense. In this case, the results of the individual runs are then combined to form an overall result.

Settings for report generation: depending on the user needs, separate reports for all functions tested or an overall report can be generated. Furthermore, it is possible to define basic things such as file name conventions or locations for the respective reports.

All settings can be done by the user directly via the user interface of the debugger. In addition, they are also accessible via functions of the COM-based automation interface. The code coverage can thus also be used by third party tools or by own scripts. Modern high-end debuggers, like the Universal Debug Engine (UDE) from PLS, play an increasingly key role in this close interaction of different components and tools. Thanks to a wide variety of COM-based internal functions of UDE for example, virtually all settings can optionally be made via third party tools or via own scripts and there is nothing else standing in the way of fully automated testing including documented test quality. ■