Measuring Code Coverage for Embedded Software

By Klaus Lambertz

For a long time already, embedded software is used for critical applications where safety is highly important. As nowadays, embedded devices are often clients which are connected with other devices on the Internet of Things (IoT), security aspects need to be considered as well. This means that the quality of embedded devices is extremely important - both from a security point of view and from a functional safety point of view.

For safe and reliable embedded devices, testing is an indispensable part of quality assurance. It is not without reason that the standards for safety-critical software development set precise requirements for test methods and test coverage. As a rule, the more critical the application, the higher are the requirement concerning the code coverage.

The most important code coverage levels are:

Statement Coverage determines which instructions were executed by the tests. Dead code can be detected as well as instructions for which no suitable test has been created yet.

Branch Coverage records whether all program branches have been tested. This is the minimum requirement that should be placed on testing. Branch coverage can be implemented with a reasonable amount of effort.

MC/DC (Modified Condition/Decision Coverage) is the highest level required by standards and rather complex. To minimize the testing effort, all atomic conditions of a composite condition are used. For each of the atomic conditions, a test case pair is tested that leads to the change of the overall result of the composite condition, but only the truth value of the atomic condition under consideration changes. Here the truth value of the other atomic conditions must remain constant.

Code size increases because of code instrumentation

For measuring which parts of the software has been tested, code coverage analyzers are used. Most coverage analyzers work according to the same principle: they instrument the code before passing it to the compiler. That means they add counters to the code which count whether the relevant code part has been executed. These counters are usually stored as global arrays. The side effect of this instrumentation is that the code becomes more voluminous. This places an additional load on both RAM and ROM.

The process is shown in Fig. 1.: The Code Coverage Analyzer Testwell CTC++ adds counters to the source code ("Instrumentation"). The information about the counters is stored in a file named "Symbol data". During the tests (right side of the figure) the number of executions is counted and stored in "Datafile". At the end of the process the "Report Generator" of Testwell CTC++ combines the information from "Symbol data" and "Datafile" to generate a "Coverage Report". The side effect of the coverage analysis is the higher memory consumption (shown at the bottom by the comparison of needed memory without and with instrumentation).

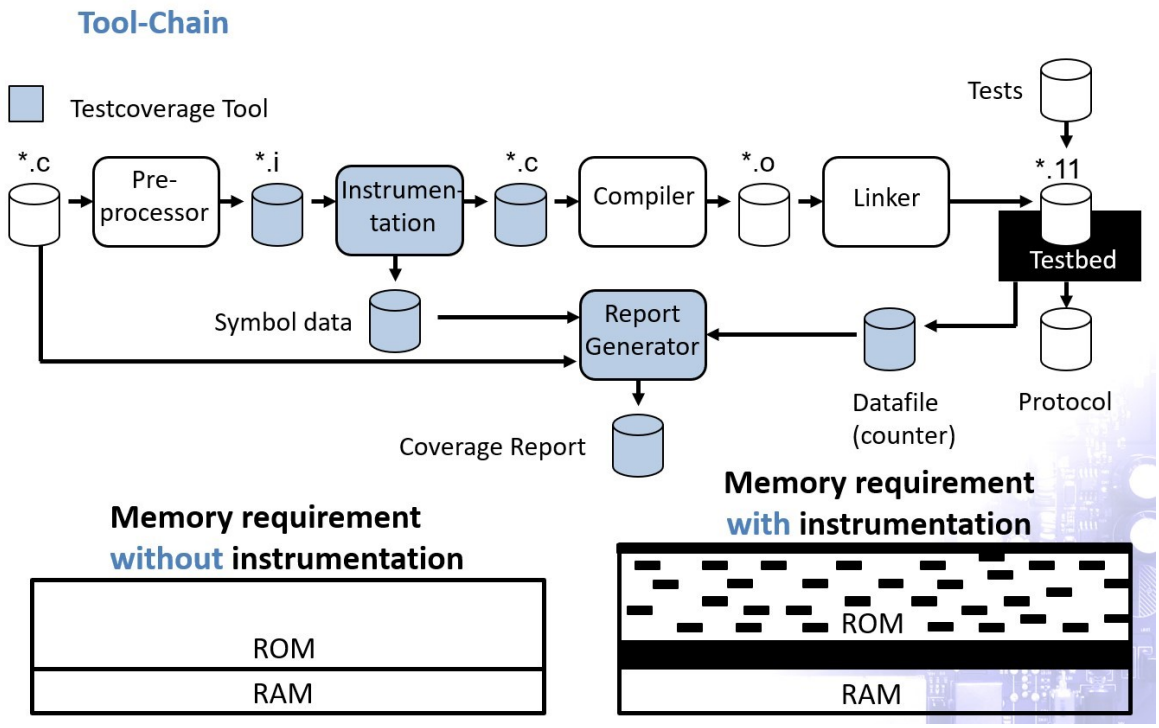

Fig. 1: Code instrumentation and coverage report generation with Testwell CTC++ Test Coverage Analyzer and its effect on needed memory space.

Even a small While condition written in C can significantly grow this way.

The initial structure

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

becomes the following through instrumentation with the code coverage analyzer Testwell CTC++:

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

For server or PC applications, this effect can be neglected. For embedded devices, on the other hand, the instrumentation overhead can lead to challenges, as hardware resources are often very tightly calculated for cost reasons. Here, care must be taken to use a code coverage analyzer with a comparatively low instrumentation overhead, otherwise the counters will quickly exceed the limits of the available memory. This is especially true when very demanding test coverage levels such as MC/DC are required. Special analyzers optimized for embedded systems, such as Testwell CTC++ from Verifysoft Technology, are the right choice.

Partial instrumentation

If the code coverage tool has a too high instrumentation overhead, this hurdle can be circumvented in RAM with partial instrumentation. With partial instrumentation, only small sections of the program under test are instrumented and tested. The test is repeated one after the other with all program parts, and the resulting data is combined to form an overall picture. This allows to determine the test coverage for the complete program.

Another possible solution for measuring code coverage on small targets is to limit the size of the counters. Normally, code coverage tools work with 32-bit counters. These counters can be reduced to 16 or 8 bits. However, caution should be exercised here, because under certain circumstances the counters can then overflow. The data obtained must therefore be interpreted with great care. In extreme cases, the counters can also be lowered to single bits. This bit coverage (provided by Testwell CTC++) can be useful, for example, if it is not relevant how often a program section has been run through.

Unfortunately, the additional space required in ROM can hardly be limited. A small library is required to capture the code coverage, which is responsible, among other things, for transmitting the counter readings to a host.

Not to be forgotten: in addition to the memory, instrumentation also places a load on the processor in the target. As a result, it can happen that a defined timing is no longer adhered to. Especially if the CPU is already working close to the limit, faulty processes can occur. Bus communication is particularly susceptible to this. Here, the tester should carefully monitor the process and carefully check the results. However, powerful code coverage tools can keep instrumentation memory requirements and runtime behavior changes relatively low.

Conclusion

Safety and security play an important role in the long-term success of IoT initiatives. In addition to applications for industry, IoT programs for the private sector must also be developed and tested in such a way that the risks for users and manufacturers are manageable. While MC/DC coverage is mandatory for safety-critical applications in cars and aircraft, at least branch coverage should be standard in all other areas. Currently,

only a few standards require proof of test coverage for software that is not safety-critical, but it is only a matter of time and market penetration before standardization bodies and industry associations increase requirements beyond safety-critical applications. Better tests are also in the interest of the manufacturers themselves, since faulty products cause high follow-up costs and can significantly damage the company's reputation. Customers in the embedded sector will hardly want to accept the “banana software” known from the PC, which only matures after delivery.

Author:

Klaus Lambertz is founder and CEO of Verifysoft Technology GmbH in Germany. Verifysoft provides Testwell CTC++, a leading code coverage analyzer for embedded software.

More information: www.verifysoft.com