Solutions for Embedded Systems Development

100% Code Coverage is not equal 100% Code Coverage

How reliable is Code Coverage information shown in source code?

Code coverage is a method to assess the quality of test runs. Based on this evaluation new test cases are deduced, redundant test cases are eliminated and inefficient test cases are changed or replaced. At the same time, code coverage is a measure to find code that is never executed. This “dead code” represents undesired overhead when assigning memory resources and is also a potential hazard if this code was not executed during testing.

Code coverage measurement (mainly on source code level) is recommended for certification in avionic, medical, automotive and nuclear standards like DO-178B, DO248D, IEC 62304, ISO 26262 and SC45A.

According to Beizer [1] **100% statement coverage at source code level** might cover **75% or even fewer statements at object code level** [2].

In this article we will discuss where this gap comes from.

Basics

Programs written in high level programming languages like C, C++, ADA etc. are not running on a processor or controller without being converted to object code. It is important to keep the following in mind: starting from the source code, there is a chain of transformations that produces the object code running on the actual target. For C programs usually these steps are: preprocessing, compilation, optimization, assembly, linking and conversion. Details vary between systems (compilers, assemblers, linkers, converters). These steps are not bijective: information is added and removed in every step. For example, linking generates addresses for variables and throws away symbolic names.

These modifications directly affect code coverage analysis on source code level because

- often compiler or library calls add object code and
- it is not easy to detect when object code is added by a compiler or a library call and
- the added object code may not be 100% covered by your tests and
- it is very complex to establish which object code correlates to which source code line (usually code coverage analysis is shown in source code).

Reasons for object code coverage

The following examples show that every time you use *complex instructions* (e.g. a for loop), *complex conditions* (e.g. if statement) or *library/operating system functions* object code is added that is not directly traceable to source code statements. Sometimes also the *compiler generates object code*. So a lot of object code is not directly traceable to source code.

Consequently DO-178/ED-12B Section 6.4.4.2b states that if "... *the compiler generates object code that is not directly traceable to Source Code statements. Then, additional verification should be performed on the object code...*"

So if an application has to ***conform to standards*** like above then the ***object code has to be covered too***. To get an application approved following three steps must be addressed (cite from CAST-12 [3]):

- *detect and identify any added object code*
- *establish the functionality of that code; and*
- *verify the correctness of the added, untraceable code sequences*

You can easily achieve object code verification and coverage using iSYSTEM on-chip debug tools and analyzers. iSYSTEM tools measure code coverage by recording all addresses being executed *on the processor* – measurement is done on low level binary data. No code instrumentation is needed. Code coverage results are displayed on source code, object code and assembly level and are correlations of the low level binary data to the high level source/object/assembly code. Object code insertions are easily visible in iSYSTEM's code coverage analysis window.

<u>*Prerequisite of such an object level code coverage measurement:*</u> *Analysis on object code runs on the code loaded in the real target hardware. To measure code coverage a hardware tool connected to the target records the activities on the CPU buses (commands and data, external signals including time stamps). With this information it can be proved in object code whether code has been executed or not. This functionality is called tracing. Tracing is available through an In-Circuit-Emulator and also through an On-Chip Debugger (OCD), but the OCD interface has to have at least one code trace port to reconstruct and measure code execution. Perfer the usage of microcontrollers that implement on chip debug cells like NEXUS or ETM.*

Relation between Source and Object Code

When reading the following examples, please note: **Non-conditional instructions** are marked with a **colored box**. This box is ***green*** for ***executed*** instructions and ***red*** for ***non-executed*** instructions. Red and green **arrows** display the **object level condition coverage** information for **conditional branches and conditional instructions**. The arrow is colored according to the paths that were executed.

1. Complex Instructions

The object code may not correspond to the source code as seen from the perspective of a source level programmer. Take this example:

```
1 int i,x;
2 for (; x < 10; ++x)
3 {
4   ++i;
5 }
```

Line 2 is typically (no matter of the processor architecture used) splitted in two or three blocks: one to initialize the loop, another to check the condition and increment the loop variable, and a tail to cleanup everything after it exits. The instructions are not run through sequentially but in two or three blocks that may be located in non-contiguous memory locations. Further, you have one or more conditions (see assembler below). Every condition splits the execution path to (at least) two possible execution paths. Code coverage information for all three blocks has to be merged to one source code line for visualization to the user. iSYSTEM tools display code coverage results depending on the executed code on the processor. In the disassembly window you can see the instruction blocks described above. Note the branch instruction “b” at the very beginning and the conditional branch “bc” with jump at the end.

```
for (; x < 10; ++x)
{
 ++i;
}
```

```
                                        for (; x < 10; ++x)
40008260:40008260    4800001C           b              4000827C
                                        ++i;
40008264:40008264    801F000C           lwz            r0,0C(r31)
40008268:40008268    30000001           addic          r0,r0,01
4000826C:4000826C    901F000C           stw            r0,0C(r31)
                                        for (; x < 10; ++x)
40008270:40008270    801F0008           lwz            r0,08(r31)
40008274:40008274    30000001           addic          r0,r0,01
40008278:40008278    901F0008           stw            r0,08(r31)
4000827C:4000827C    801F0008           lwz            r0,08(r31)
40008280:40008280    2F800009           cmpi           7,0,r0,09
40008284:40008284    409DFFE0           bc             04,1D,"test.c"::58 (40008264)
```

!! When measuring code coverage watch out that all paths are covered.

2. Complex Conditions

For example:

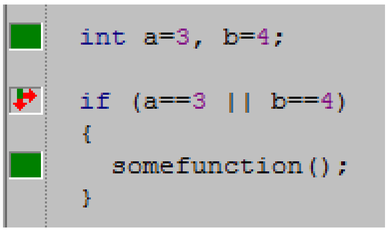
```
1 if (a == 3 || b ==4)
2 {
3   somefunction(c);
4 }
```

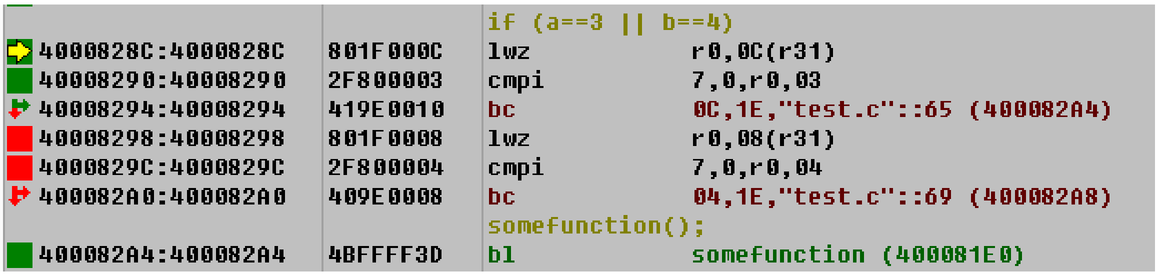
Line 1 of the example can be only partially tested when the first condition evaluates to true and the evaluation short-circuit logic* applies. In this case the second condition is not tested at all.

!! When measuring code coverage ensure that every condition is assessed.

iSYSTEM tools show this situation with the greatest possible amount of the detail. If the first condition is reached and the evaluation short-circuit logic applies, then the coverage information (in the source code window) will show the line as not executed. If you drill down on the assembler information for this line you will see that only part of the code was executed.

<u>This is shown when using iSYSTEM tools:</u> Note how the source code shows the "if" statement as executed (the arrowroot is green) but neither path taken (arrowheads are red). This is to signal a mixed case that has to be analyzed on the assembler level. In the disassembly window you see that the first condition was true, so the second one was never evaluated - therefore the coverage legend on these assembler lines shows red squares and arrows.





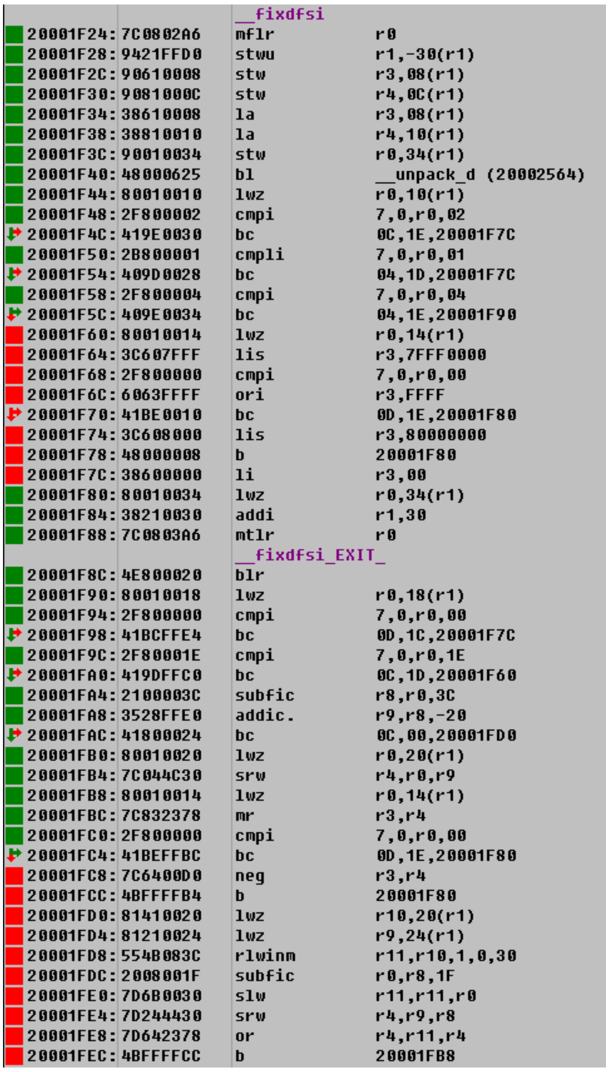
***Short-circuit evaluation**, **minimal evaluation**, or **McCarthy evaluation** denotes the semantics of some Boolean operators in some programming languages in which the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression: when the first argument of the AND function evaluates to false, the overall value must be false; and when the first argument of the OR function evaluates to true, the overall value must be true.

3. *Libraries*

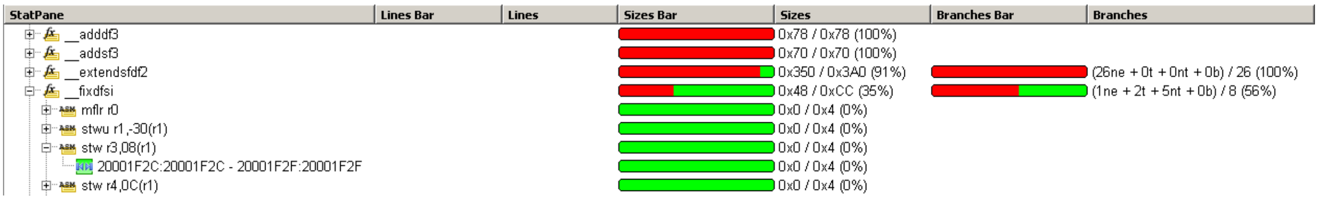
Using library functions, e.g. operating system functions, will add object code where no source code is available. These library functions have to be thoroughly tested because they could include dead code or never taken code paths.

iSYSTEM tools gather coverage information directly from the hardware. So the analysis of library code isn't different to analysis from code written by the user. If source code is not available (e.g. for an operating system function) the information is given on the object and assembler levels only. iSYSTEM tools allow to configure if library code is included in the overall coverage information and reports or not.

<u>This is shown when using iSYSTEM tools:</u> Note that not all instructions of the library function were tested.



Here you see iSYSTEM winIDEA’s hierarchical code coverage view for above library function. Because there is no source code available for these functions the "Lines" information columns are empty. Nevertheless you can view full code coverage information for each function, each object code and each assembler instruction, or you can collapse parts of the tree to have an overview.



4. Compiler Generated Code

Specially on smaller processors (e.g. Freescale MPC5604B), but not only, the compiler needs to introduce code to bridge the gap between what is possible in source code and on the processor. This is mostly the case for operations that don't translate directly to machine code. For example a conversion of a floating point variable to an integer is usually performed by code that is inserted inline or a library function call is introduced by the compiler at that point.

!! Compiler generated code should be thoroughly verified because it could include dead code or never taken code paths.

As iSYSTEM tools gather coverage information directly from the hardware, the analysis of compiler generated code isn't different as analysis from code written by the user. If source code is not available, the information is given on the object and assembler levels only.

Please check below example. See how the assignment statement "c=d;" uses code that isn't obviously part of the source. What is even more interesting is that this code includes conditionals and even function calls. In this particular case you see that part of the code wasn't executed, including a function call, because this is data dependent. This could lead to catastrophic failure if the function (that wasn´t called) is never tested and has bugs.

```
void Type_Simple()
  {
  otm(fn_Type_Simple);
  char c=0;
  unsigned char uc=0;
  int i=0;
  unsigned ui=0;
  long l=0;
  unsigned long ul=0;
  int iCount=0;
  long long ll=0x12345678;
  float f=(float)0.0;
  double d=1.583;

  for (iCount=-2;iCount<=2;++iCount)
    {
    c=iCount+3;
    d=c;
    c=d;
    uc=iCount;
    i=iCount;
    ui=iCount;
    l=iCount;
    ul=iCount;
    f=(float)iCount;
    }
```

```
                   d=c;
20000444 881F0035 lbz      r0,35(r31)
20000448 7C000774 extsb    r0,r0
2000044C 7C030378 mr       r3,r0
20000450 480019F9 bl       __floatsidf (20001E48)
20000454 907F0038 stw      r3,38(r31)
20000458 909F003C stw      r4,3C(r31)
2000045C 881F0035 lbz      r0,35(r31)
20000460 7C000774 extsb    r0,r0
20000464 2F800000 cmpi     7,0,r0,00
20000468 409C0028 bc       04,1C,20000490
2000046C 807F0038 lwz      r3,38(r31)
20000470 809F003C lwz      r4,3C(r31)
20000474 3CA04070 lis      r5,40700000
20000478 38C00000 li       r6,00
2000047C 480018BD bl       __adddf3 (20001D38)
20000480 7C691B78 mr       r9,r3
20000484 7C8A2378 mr       r10,r4
20000488 913F0038 stw      r9,38(r31)
2000048C 915F003C stw      r10,3C(r31)
20000490 813F0038 lwz      r9,38(r31)
20000494 815F003C lwz      r10,3C(r31)
20000498 913F0008 stw      r9,08(r31)
2000049C 915F000C stw      r10,0C(r31)
```

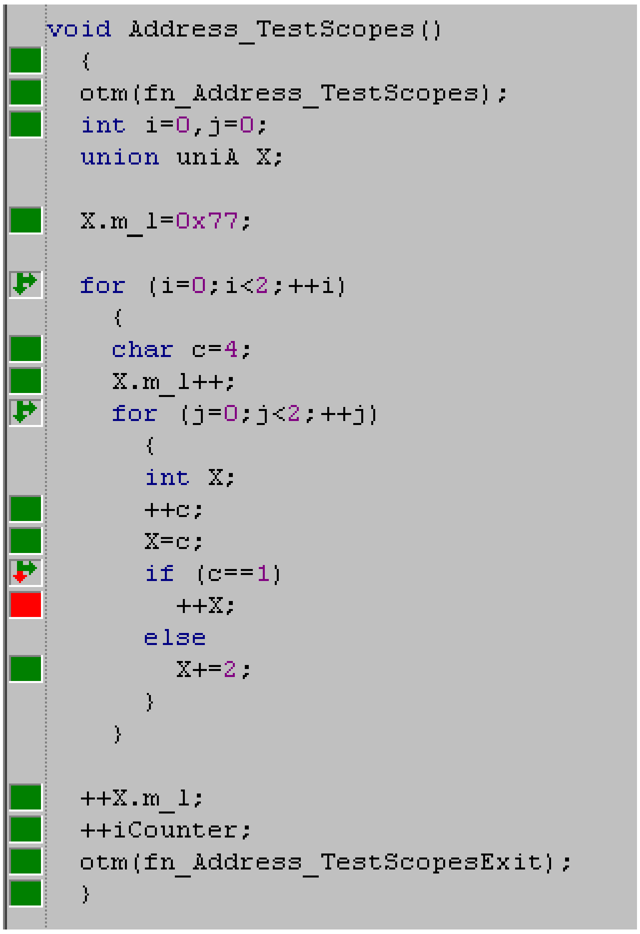
5. Compiler Optimizations

Another aspect is compiler optimizations. Compilers may

- merge several source code lines (e.g. merge string constant expressions, expression simplification,…),
- deem unnecessary code lines (e.g. dead code elimination, removal of unused symbols, common subexpression elimination,…) and/or
- reorder the execution flow (e.g. loop inversion, loop fusion,…).

!! When measuring code coverage, verify what source code lines generate object code and if not, check why. Any code change and recompilation may change the compiler optimization and could lead to untested code paths.

iSYSTEM tools take great care to show accurately what is actually going on. In iSYSTEM's winIDEA source code view every line that generates object code has a code coverage legend aligned to the left. Note that some "{" and "}" are not creating object code and therefore have no code coverage legend aligned.



6. Compiler Bugs

Seldom, but still, compiler can have bugs. These bugs are rare on the code that is generated, as compilers are usually tested in this respect. Most of the time bugs are in the debug information generated by the compiler and used by other tools. It is important to remark that these happen seldom, but shouldn't be ruled out.

Conclusion

The requirements for code coverage analysis are constantly increasing in almost every market. Code coverage based on object code is a convenient method to test as close as possible to the final product.

Evaluating code coverage results keep in mind what we have discussed in this article. To achieve a high level of code coverage (some claim 100% is the right figure), review the object code and find test cases to cover the (not yet) tested code.

iSYSTEM tools provide code coverage analysis within the same tool that is used for code development. This way development costs and time are optimally used to detect and fix software bugs as soon as possible within the development cycle.

Consequently, choosing a processor or controller for your product should also include the requirement of providing on-chip trace capabilities.

Links

Difference between Source and Object code Coverage

Offline versus Realtime Execution Coverage

Code Coverage: Relation between Source and Object Code

References

[1] Beizer, Boris: *Software Testing Techniques*, Second edition, Von Nostrand Reinhold Company, Inc., 1990
[2] Hayhurst, K. J., Veerhusen D. S., Chilenski J. J., Rierson, L. K.: *NASA / TM-2001-210876 A Practical Tutuorial on Modified Condition/Decsion Coverage*, NASA Center for Aerospace Information (CASI)
[3] Certification Authorities Software Team (CAST): *Guidelines for Approving Source Code to Object Code Traceability*, Completed December 2002