Source level testing and

OBJECT LEVEL TESTING

Objectives

At the end of this section, you will be able to

- Explain the advantages and disadvantages of both instrumented testing and object level testing
- Describe how iSYSTEM's object level testing implements its tests 'on target'

testIDEA

Contents

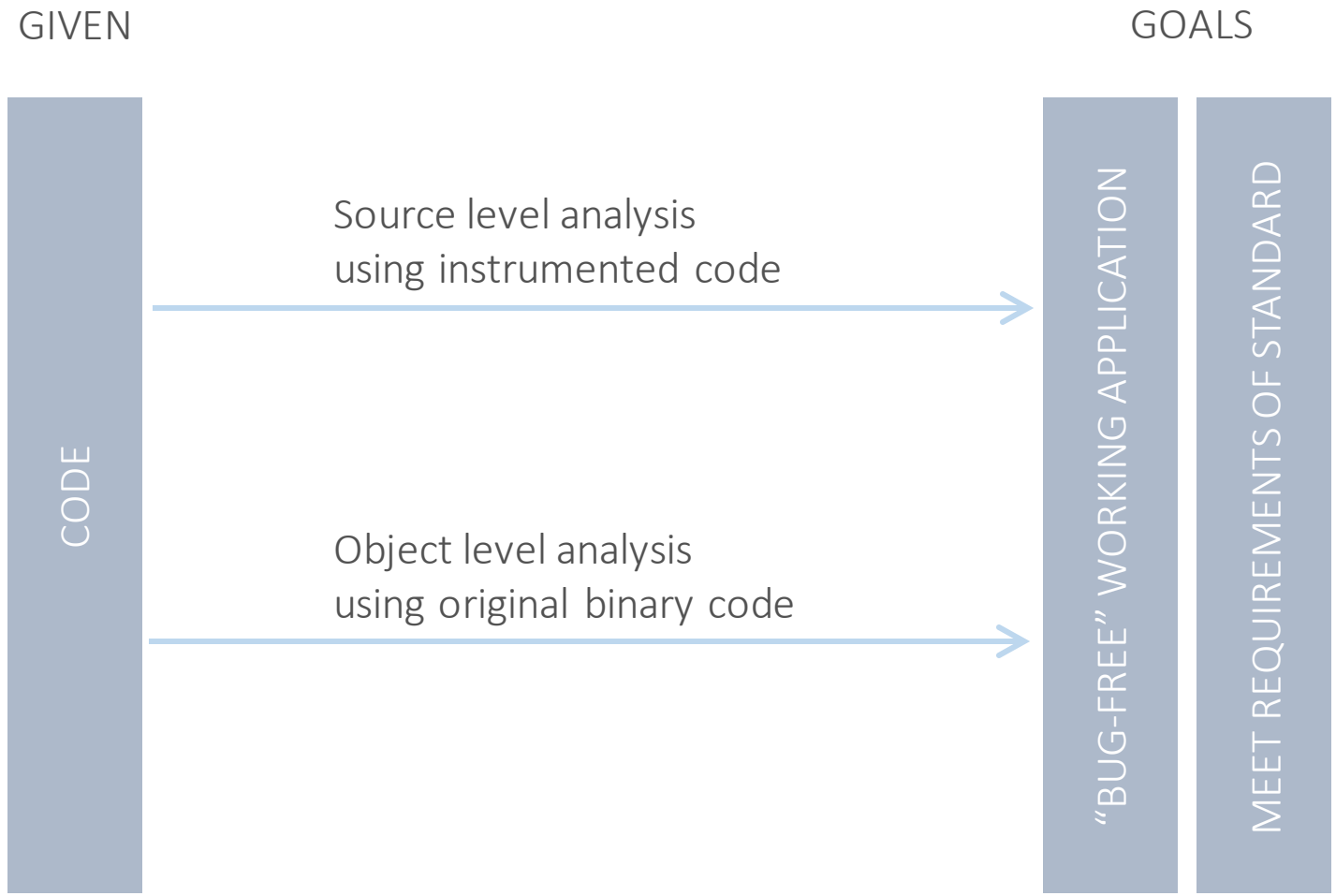
OBJECT LEVEL TESTING

testIDEA

1 METHODS FOR PROOF OF TESTING

Especially in the area of Functional Safety, it is necessary to provide proof of how much of your application's software was tested. The proof can be generated by executing a test suite of code that exercises, as far as possible, every function or method within your application and works through every decision outcome that is found. There are essentially two methods for creating this proof:

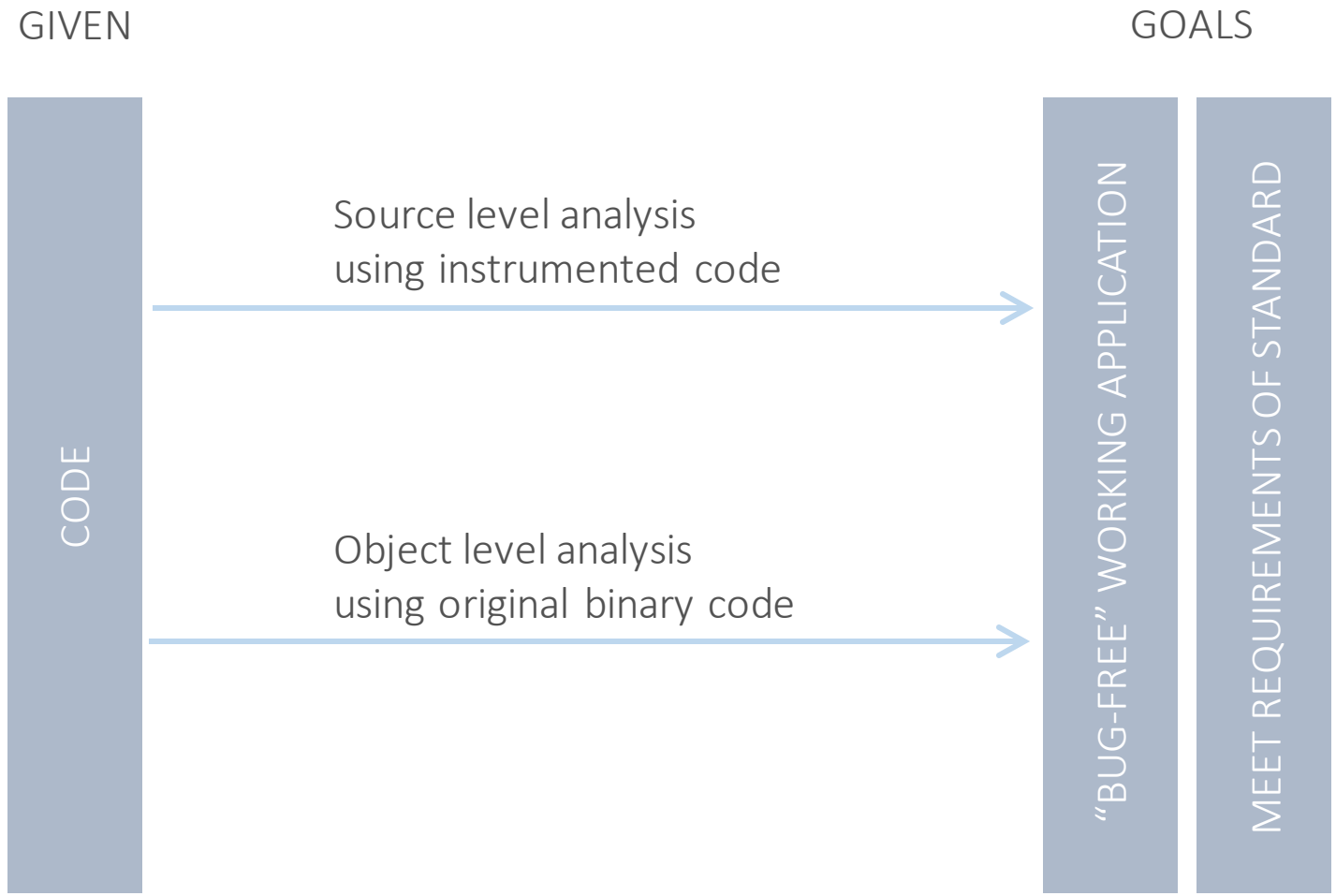
- Code instrumentation – essentially adding code that logs the outcome of decisional branch statements
- Object code trace – using the microprocessor's hardware trace output to log the outcome of decisional branch statements



1 METHODS FOR PROOF OF TESTING

Each method has its disadvantages and advantages, which will be covered here and in following units.

However, regardless of which method is used, most standards that cover functional safety will require that the code coverage for the application code can be determined through measurement and that the tests are generated using a proven methodology.



2 THE GOALS - WHAT STANDARDS EXPECT

As a result of these standards, the most common approach for proving that the code has been fully tested is by using code instrumentation. This involves using tools that add code to your application that log the path taken when decisions are made, such as the path taken through the code in an *if* statement.

This instrumented version of the code is then executed on a PC using a simulated version of the target processor. However, the PC simulation can rarely fully reflect all of the complexity of a microcontroller and its peripherals in a real-time application.

DO-178B/C and ISO26262 assume
source level code coverage analysis

Typical process

- Code is instrumented, then executed on PC using target processor simulator
- Test repeated on target with same test vectors - ***or*** -
- Instrumented code run on target - results transferred to PC for comparison

2 THE GOALS - WHAT STANDARDS EXPECT

Thus, the code is then executed again on the target microcontroller in one of two ways:

- The instrumented version of the application code is executed on the target, with the decision logic outcomes being logged.
- The non-instrumented version of the application code is executed with the results of the tests only being logged.

However, the instrumented version will be slowed down due to the extra code overhead – a huge issue in real-time applications – and the non-instrumented version is no longer the same code as the version tested on the PC, due to the missing instrumentation code.

DO-178B/C and ISO26262 assume
source level code coverage analysis

Typical process

- Code is instrumented, then executed on PC using target processor simulator

- Test repeated on target with same test vectors ***- or -***
- Instrumented code run on target - results transferred to PC for comparison

3 THE GOALS - SOURCE COVERAGE TYPES

To find a good balance between time spent generating and executing tests, different coverage levels are defined that depend on the risk to life the application poses. Achieving a certain level of code coverage requires analysis of the source code and writing tests that achieve that coverage level.

For example, statement coverage simply requires that the tester can prove that all source code statements (lines of code) in the application were, during all rounds of testing, executed at least once. Obviously this could leave some bugs unfound if the complexity of some decision making code isn't fully analyzed.

- Statement coverage
- Function coverage
- Call coverage
- Decision coverage
- Branch coverage
- MC/DC - Modified Condition/Decision Coverage

Least Risk to Life

Greatest Risk to Life

3 THE GOALS - SOURCE COVERAGE TYPES

As the risk to life increases, so more time must be invested in creating the tests and more time must be spent analysing the functionality of the code. As such, it is necessary to start to analyse the decision making elements of the application, developing enough tests that ensure enough of the logical outcomes have been proven to have been tested.

- Statement coverage
- Function coverage
- Call coverage
- Decision coverage
- Branch coverage
- MC/DC - Modified Condition/Decision Coverage

Least Risk to Life

Greatest Risk to Life

4 SOURCE LEVEL ANALYSIS - CHALLENGES SUMMARY

Due to the additional instrumentation code, recompilation of the application using different compiler switches, differences between simulated and real hardware, and the impact the new code and its size may have on the real-time behavior of the code when it is executed on the target microcontroller, it often becomes very challenging to prove the link between the instrumented and original code as well as the validity of the test results.

Instrumented Code ≠ Original Code

- Recompilation required
  - Increases code size
  - May require communication a resource (e.g. UART) to deliver results to host PC

- Are silicon differences reflected in simulation environment?

- Real-time behaviour changes
  - Extra cycles needed to execute test code
  - Tasks may not complete on time
  - Page boundaries change
  - Potential cache misses
  - System test not possible

- Changes in compiler behaviour
  - Different compiler switches used
  - Additional, new code
  - Evaluation of conditional code may change

5 SOURCE AND OBJECT LEVEL TESTING - THE DIFFERENCES

With the proliferation of hardware trace, even on low-cost microcontrollers, it has become easier to test 'on target', examining the collected program flow after testing on the PC. This enables the application code to be executed without the overhead of instrumentation, as well as allowing the code to interact in real-time with peripherals.

Also, since the code size remains the same and is compiled with the same switches, issues caused by cache misses and other run-time hardware-based differences, are avoided.

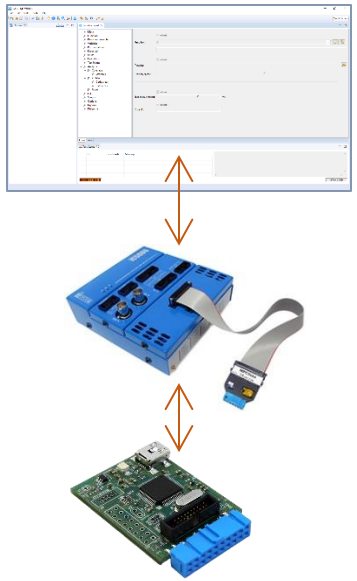
It is even possible to test optimized versions of code.

Source level testing

- Executed on host PC
- Source code executed
- On (instruction set) simulator for the target CPU

Object level testing

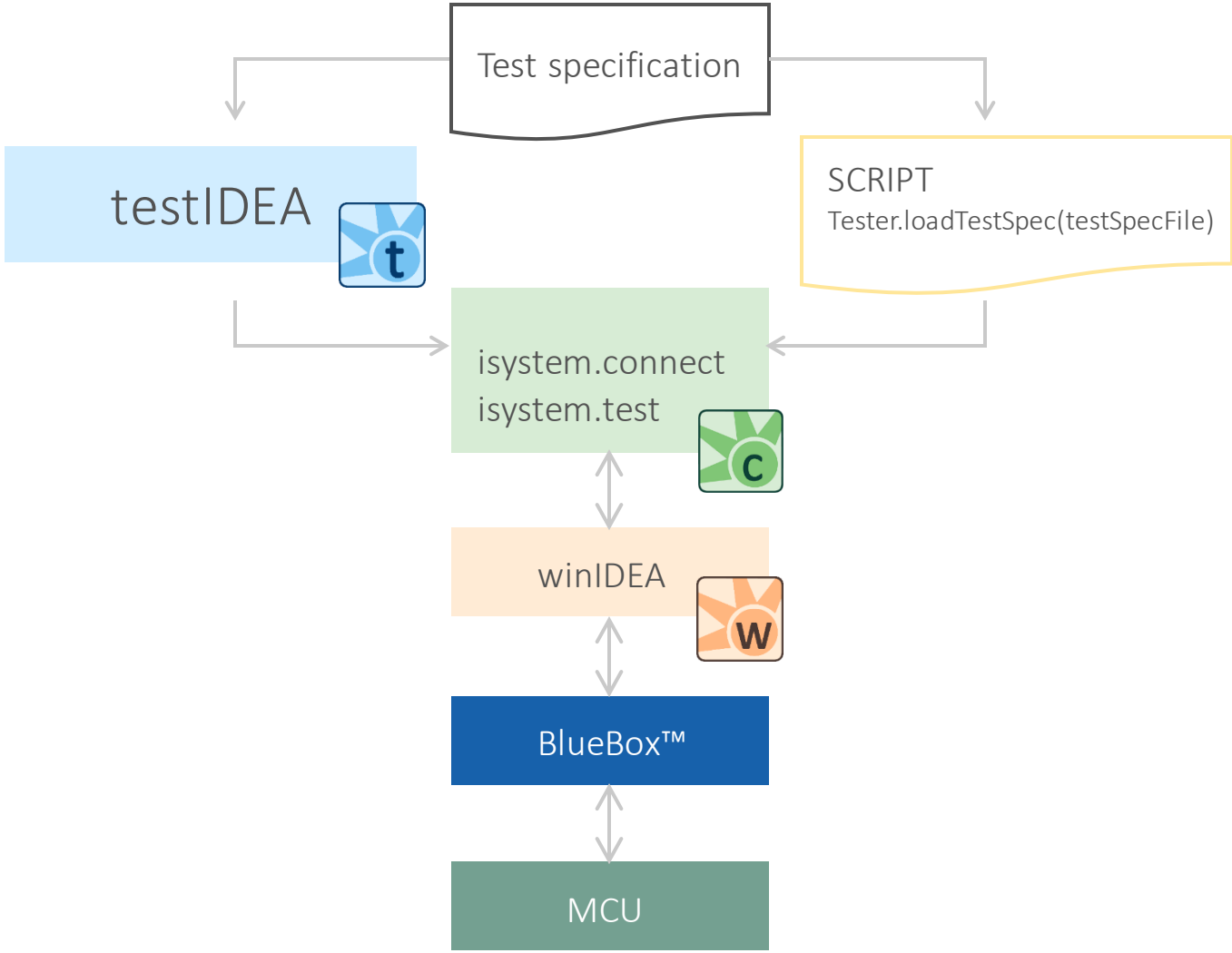
- Executed on real target
- Object code executed
- No test driver software running on target
- Integrated into the development environment (debugger)



6 NON-INSTRUMENTED OBJECT LEVEL TESTING

Non-instrumented object level testing requires a development environment that can communicate with the target hardware directly. The iSYSTEM development environment consists of a test creation and execution package (testIDEA) and a connection to the hardware target (winIDEA together with the chosen BlueBox™).

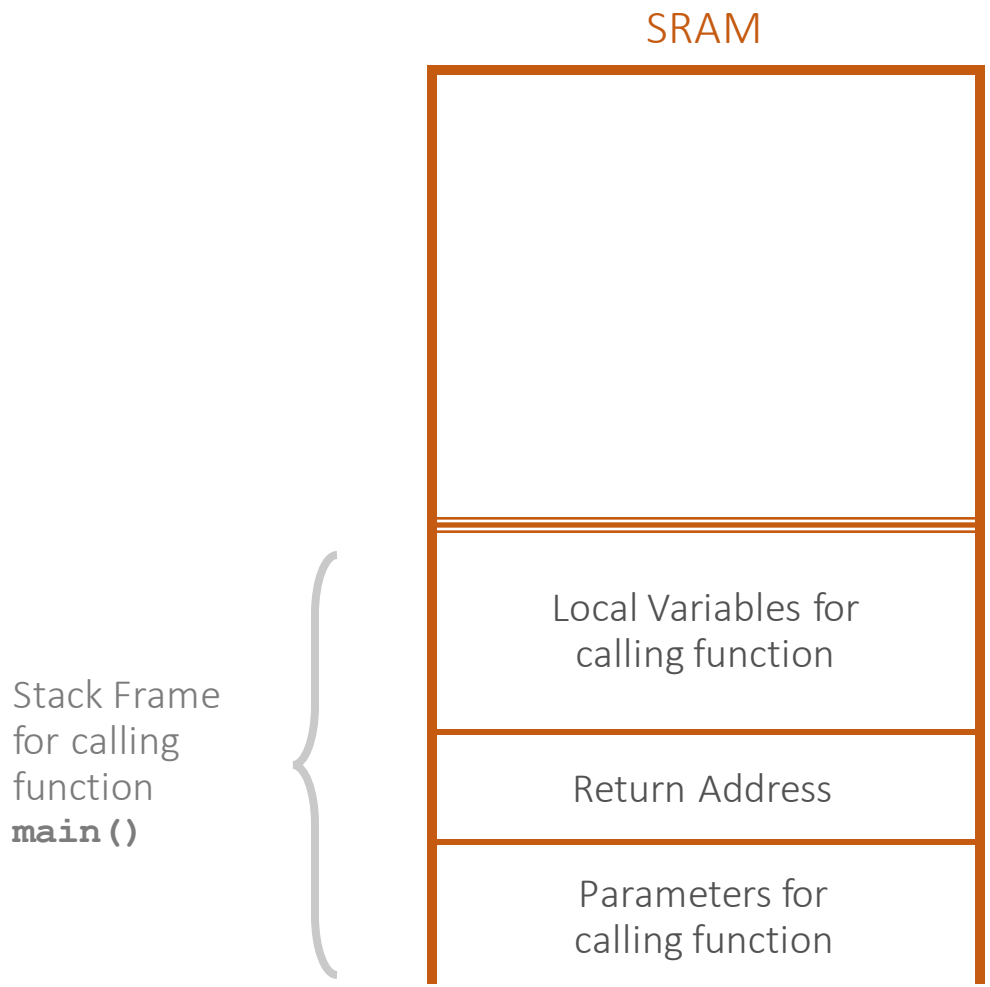
Using the isystem.connect SDK in the background, generated tests are executed directly on the target with the results being collected in real time. If required, code coverage can also be collected and the statistics turned into reports for documentation purposes.



6 NON-INSTRUMENTED OBJECT LEVEL TESTING

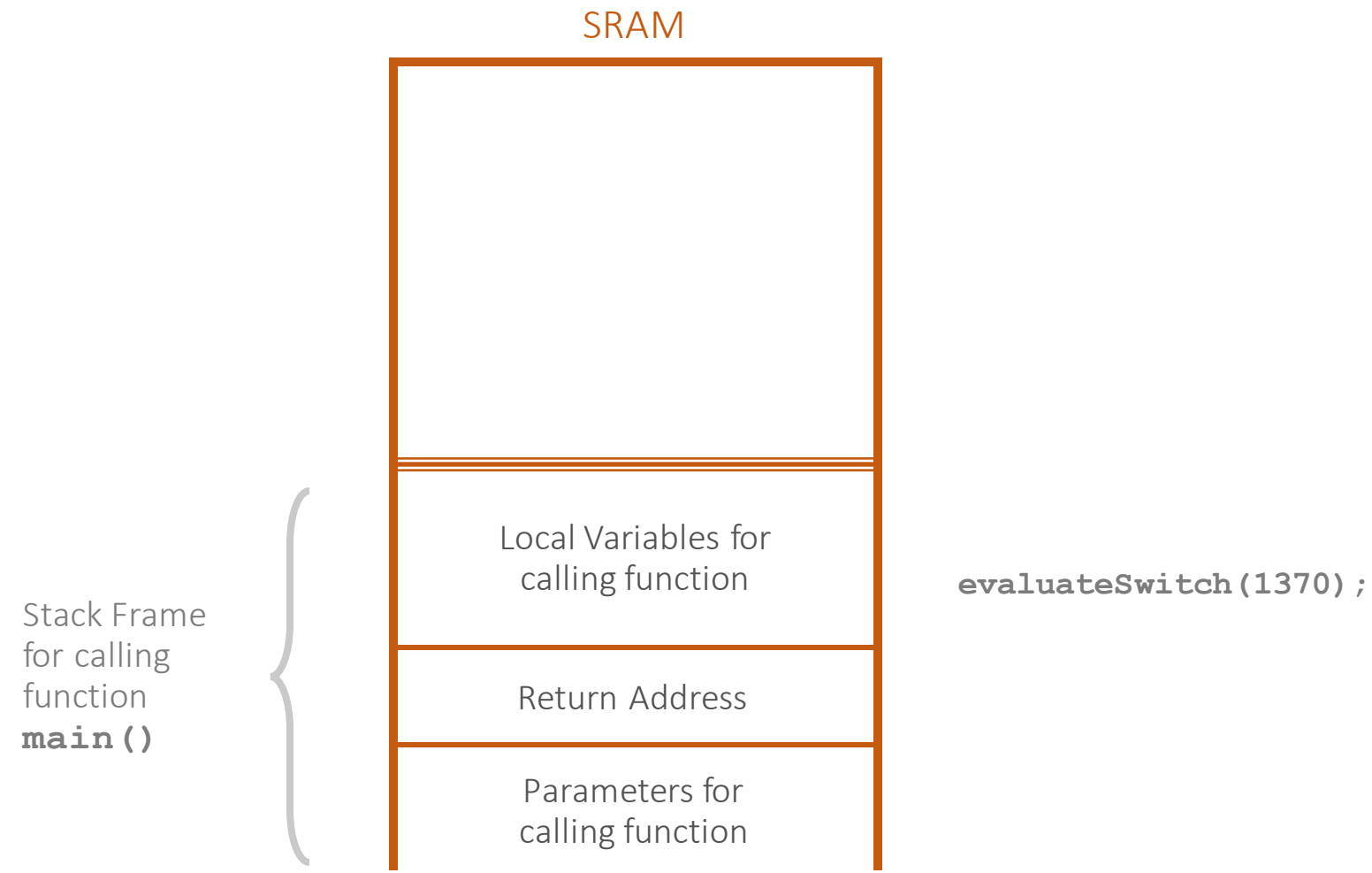
Object level testing relies upon an understanding of how the compiler used works – specifically, how it creates a stack frame when calling a function and how it acquires any return value. This know-how has been used by iSYSTEM to develop testIDEA. Thus, those developing the tests do not need to concern themselves with such details.

In this example, assume we are currently in the **`main()`** function. The compiler will have already ensured that code is in place to prepare the stack to support this function. This includes reserving space for local variables, return address, and so on.



6 NON-INSTRUMENTED OBJECT LEVEL TESTING

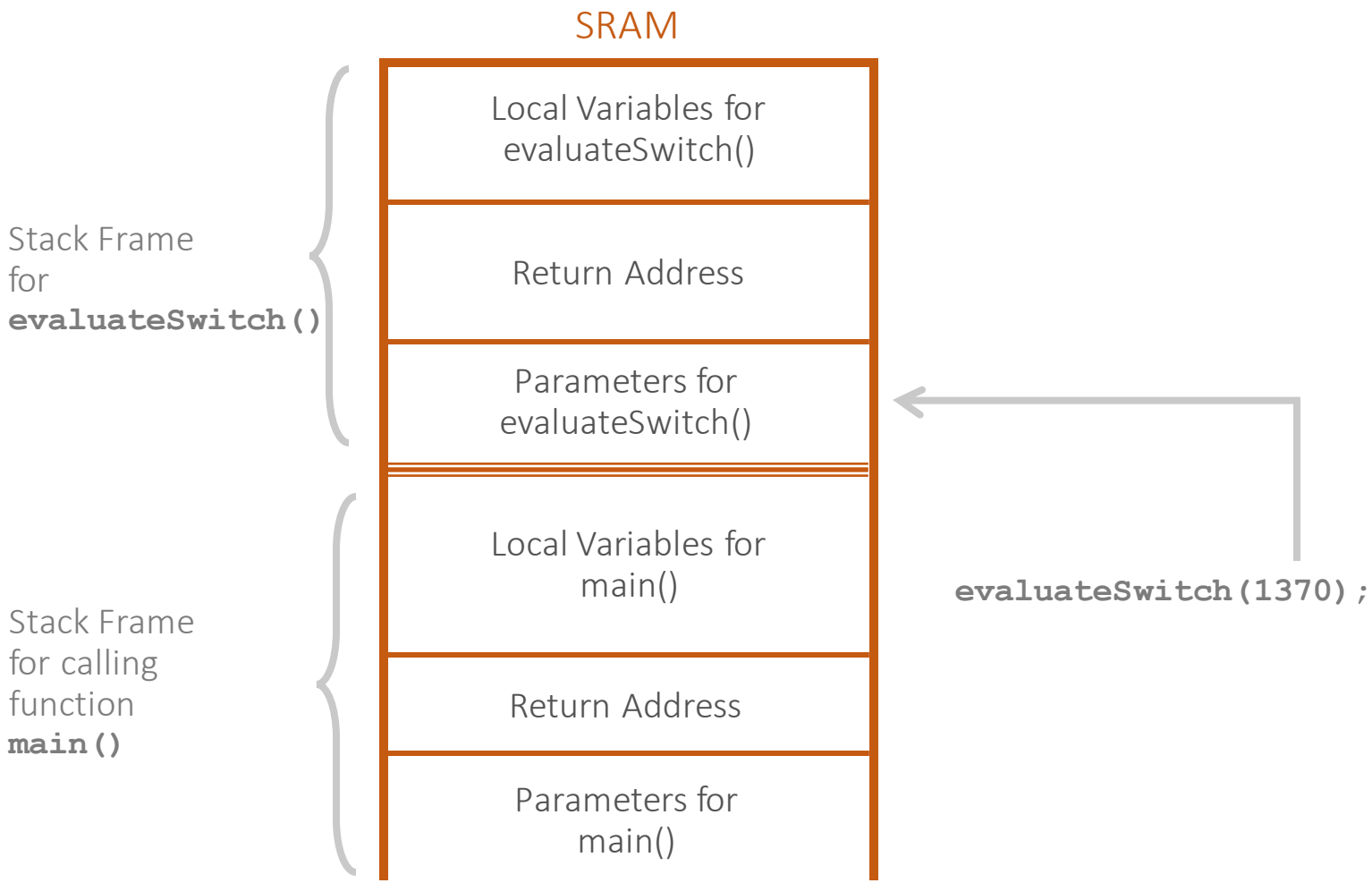
At some point in the code a function is called. In this case, it is the function **`evaluateSwitch()`**, passing in the parameter “1370”.



6 NON-INSTRUMENTED OBJECT LEVEL TESTING

At this point a new stack frame is created in RAM memory. Space on this frame is also reserved to pass the parameter (1370) to the function being called, as can be seen here inside the brown frame.

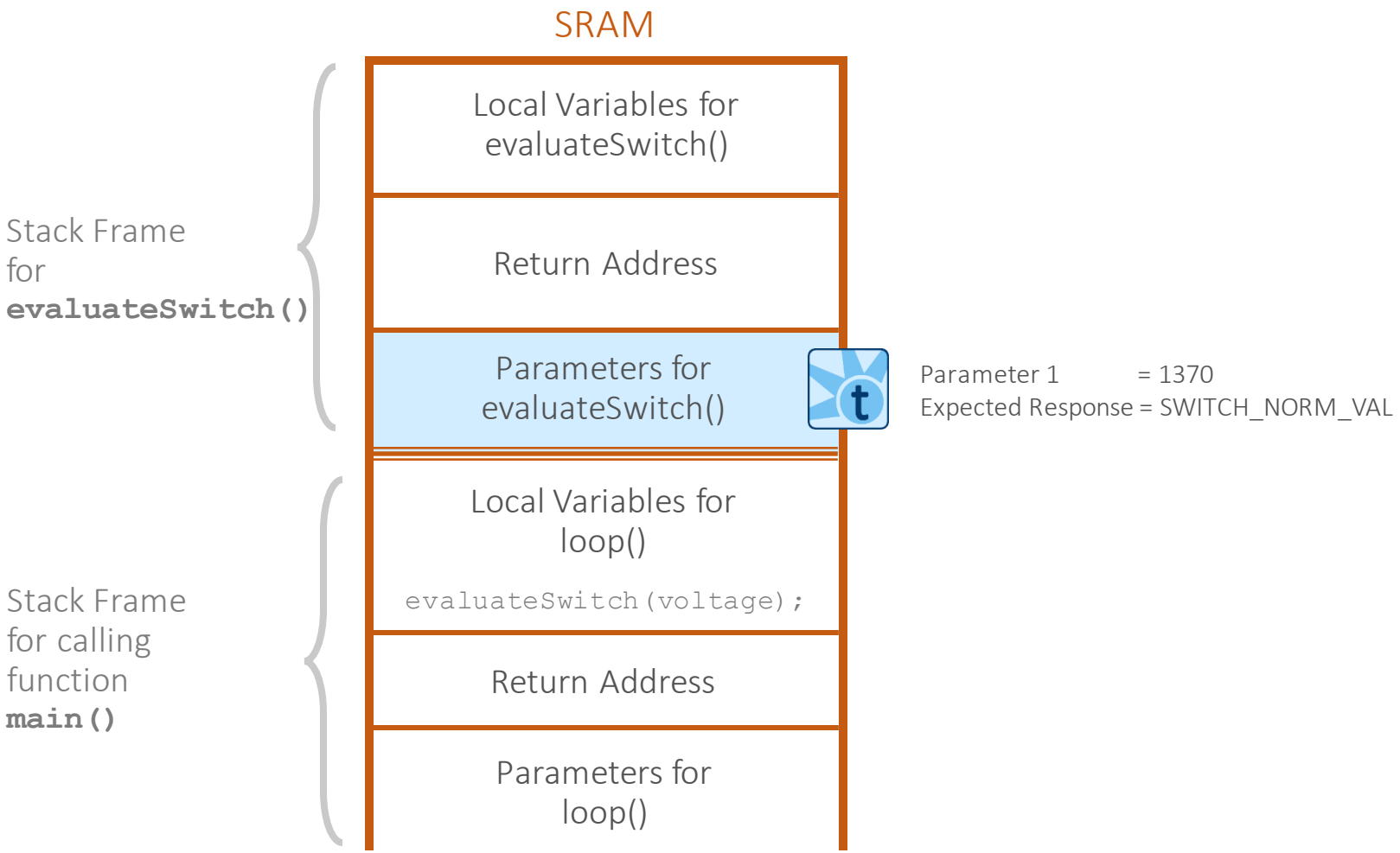
Upon completion of the execution of the function, the return value can be found on the stack.



6 NON-INSTRUMENTED OBJECT LEVEL TESTING

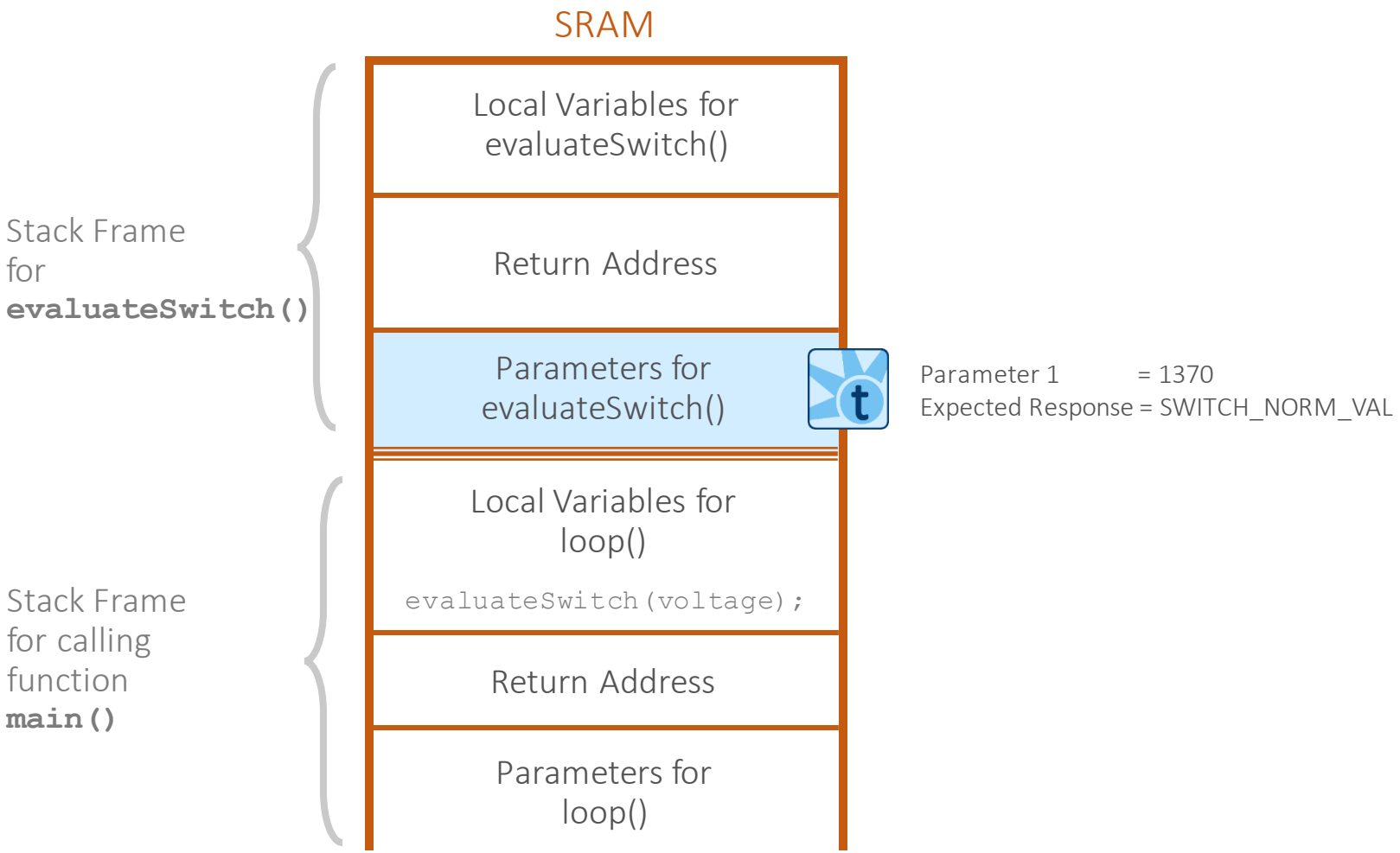
testIDEA utilizes this intimate understanding of stack frame creation to execute tests for functions.

After creating a test within testIDEA, where an expected response is linked to one or more passed parameters, the function to be tested can be executed in isolation. The desired parameters are passed in, the function is executed in place until it returns, whereupon the processor is halted again. Using the debug interface provided by the winIDEA development environment, the resulting return value can be retrieved from the stack of the microcontroller. If the value matches that expected, the test passes. Otherwise, the test fails.



6 NON-INSTRUMENTED OBJECT LEVEL TESTING

By making use of various features of the BlueBox™ development hardware, it is also possible to collect code coverage information during the execution of the function under test. This also includes microcontrollers that don't even offer any form of hardware trace interface. Such capabilities will be covered in this training together with how to generate suites of test for a variety of purposes and goals.



2 NON-INSTRUMENTED OBJECT LEVEL TESTING

To further underline the difference between instrumented and non-instrumented testing, we provide the example code opposite that was taken from a real application that has had code instrumentation inserted into it by a code instrumentation tool.

In this case, it is the exit of the function call that has been instrumented. In total, 12 additional assembler instructions have been inserted. At a single-cycle execution rate of 50MHz, this would equate to the exit of each function taking at least 240ns longer to complete.

In a real-time system, this change is likely to have a great impact on the functionality of the application.

```
Address   Disassembly
          IsOdd
          }
4000223C  rlwinm        r3,r3,0,31,31
          IsOdd_EXIT_
40002240  blr
```

- 12 extra instructions
- > 240ns @ 50MHz

```
Address   Disassembly
          IsOdd_instr
          g_cov = 0;
40002204  li            r0,00
40002208  lis           r9,40010000
4000220C  stb           r0,-0FDC(r9)
          if (a & 1)
40002210  andi.         r0,r3,01
40002214  bc            0D,02,"main.c"::59 (40002228)
          g_cov = 1;
40002218  li            r0,01
4000221C  stb           r0,-0FDC(r9)
40002220  li            r3,01
          IsOdd_instr_EXIT_
          return 1;
40002224  blr
          g_cov = 2;
40002228  li            r0,02
4000222C  lis           r9,40010000
40002230  stb           r0,-0FDC(r9)
40002234  li            r3,00
          IsOdd_instr_EXIT_2
          }
40002238  blr
```

2 NON-INSTRUMENTED OBJECT LEVEL TESTING

This disconnect between instrumented and non-instrumented code, or results from a simulated and a real processor, will cause challenges either in the testing itself or in trying to convince a certification authority of the reliability of the resulting evidence.

Address	Disassembly
	IsOdd
	}
4000223C	rlwinm r3,r3,0,31,31
	IsOdd_EXIT_
40002240	blr

- 12 extra instructions
- > 240ns @ 50MHz

Address	Disassembly
	IsOdd_instr
	g_cov = 0;
40002204	li r0,00
40002208	lis r9,40010000
4000220C	stb r0,-0FDC(r9)
	if (a & 1)
40002210	andi. r0,r3,01
40002214	bc 0D,02,"main.c"::59 (40002228)
	g_cov = 1;
40002218	li r0,01
4000221C	stb r0,-0FDC(r9)
40002220	li r3,01
	IsOdd_instr_EXIT_
	return 1;
40002224	blr
	g_cov = 2;
40002228	li r0,02
4000222C	lis r9,40010000
40002230	stb r0,-0FDC(r9)
40002234	li r3,00
	IsOdd_instr_EXIT_2
	}
40002238	blr

7 ORIGINAL BINARY CODE (OBC) TESTING

Object level testing has been compared with instrumented testing methods by various organizations, including CAST, an international team of certification and regulatory authority representatives from North and South America, Europe, and Asia.

Overall, binary code testing is considered equal in its purpose to instrumented methods of testing. However, simply proving that all branch possibilities have occurred at binary level is not equivalent to all test strategies that can be undertaken that have knowledge of the source code. Specifically, this is the case with MC/DC testing, which does still require source code analysis.

- Considered equal to source code coverage analysis
  - Reviewed by CAST (Certification Authorities Software Team)
- Tests should be developed with knowledge of the source code (note comments on MC/DC)
- May need to show that object level code coverage is equivalent to source level analysis

For more details on the CAST findings, take a look at this link

7 ORIGINAL BINARY CODE (OBC) TESTING

Due to the fact that instrumented approaches to testing are more established than object level testing, there may be some issues when submitting results to a certifying body. In such cases, you may be required to show that the object level testing is equivalent to an instrumented, source-level testing methodology. The best advice here is to communicate with such bodies early in the test design process to ensure that everyone is satisfied with the approach chosen.

If you are undertaking testing merely to improve software quality and have no certification pressures, the testIDEA approach and tools will be more than adequate in most cases.

- Considered equal to source code coverage analysis
  - Reviewed by CAST (Certification Authorities Software Team)
- Tests should be developed with knowledge of the source code (note comments on MC/DC)
- May need to show that object level code coverage is equivalent to source level analysis

For more details on the CAST findings, take a look at this link

SUMMARY

testIDEA

8 SUMMARY

- Code instrumentation is the incumbent method for testing
- Original Binary Code (OBC) testing is, in most cases, more than adequate
- The iSYSTEM OBC method also allows code coverage to be generated even when the microcontroller is missing the necessary hardware trace interface