PARASOFT®

Practical Unit Testing for Embedded Systems

Part Two

Table of Contents

In part one of this article, we provided a brief introduction to unit testing. We also discussed the challenges embedded developers face when doing unit testing. We used a specific example: A simplified ASR (Acceleration Slip Regulation) system running on a Keil evaluation board MVBSTM32E. In this part we will show what it mean to obtain particular goals with unit testing, such as achieving a target level of code coverage. We will also discuss the safety relevance of unit testing.

Achieving actual results with Unit Testing

All of the required setup, as described in the previous sections, took some time. We also learned about some challenges to keep in mind for unit testing code for embedded software. Now, we can start enjoying the fruits of our labor. It was not particularly hard labor since C++test automated a lot of the initial set up tasks, but some work was required. For an experienced user, setting up a moderate project for unit testing with C++test takes anywhere from several minutes up to an hour, assuming there are no extra challenges to overcome.

Coverage goals

An international standard IEC-61508-3 edition 2 highly recommends explicit types of the code coverage that need to be achieved for a given Safety Integrity Level. For SIL 2 you need to show 100% statement coverage; SIL 3 requires 100% branch coverage and SIL 4 requires 100% MC/DC coverage (Table B-2). Without unit testing, achieving 100% coverage—even in terms of line coverage—is nearly impossible. But now, once we have our project set up, there are several techniques that will significantly reduce the effort required.

Coverage metrics

First of all, notice that C++test actually shows us coverage: line, statement, branch or MC/DC coverage. You can easily navigate directly to the functions/methods where extra work is needed to achieve the desired coverage level. Without this capability it would be impossible direct effort where it is needed. You would just not know which part of code needs additional coverage, and which doesn't. So ensure that whatever framework you want to use provides the correct coverage metrics.

Generating test cases

Let's say that our initial goal is to get 100% statement coverage with our unit tests (SIL 2). Let's first focus on one file, "proc.c", which contains functions to calculate the wheels' rotation speed ("update_speed"), calculate braking power ("update_brake_signal", "brake_control") and some additional smoothing function ("average"). For the sake of keeping this article a reasonable length, we will not discuss exactly how those functions work and whether they are good or bad, optimal or not optimal etc. That is not the point here. Our focus is on how to unit test it.

To get started, we could just start typing test cases. However, a better approach is to generate some tests automatically. All you need to do is to highlight the file (or the directory or the entire project) that you want to generate test cases for, and then choose the built-in "Generate Unit Tests" test configuration from the C++test menu. In our example, all functions are in one file, "proc.c", so we highlight it and run generation. A "tests" directory is created in the project, along with a "autogenerated" subdirectory. C++test then analyzes code, generates unit test cases, and stores them in a file within that directory. In this case, the generation took only 30 seconds and it resulted in 38 test cases being generated. Remember that generation time depends on the code: it may be longer or shorter.

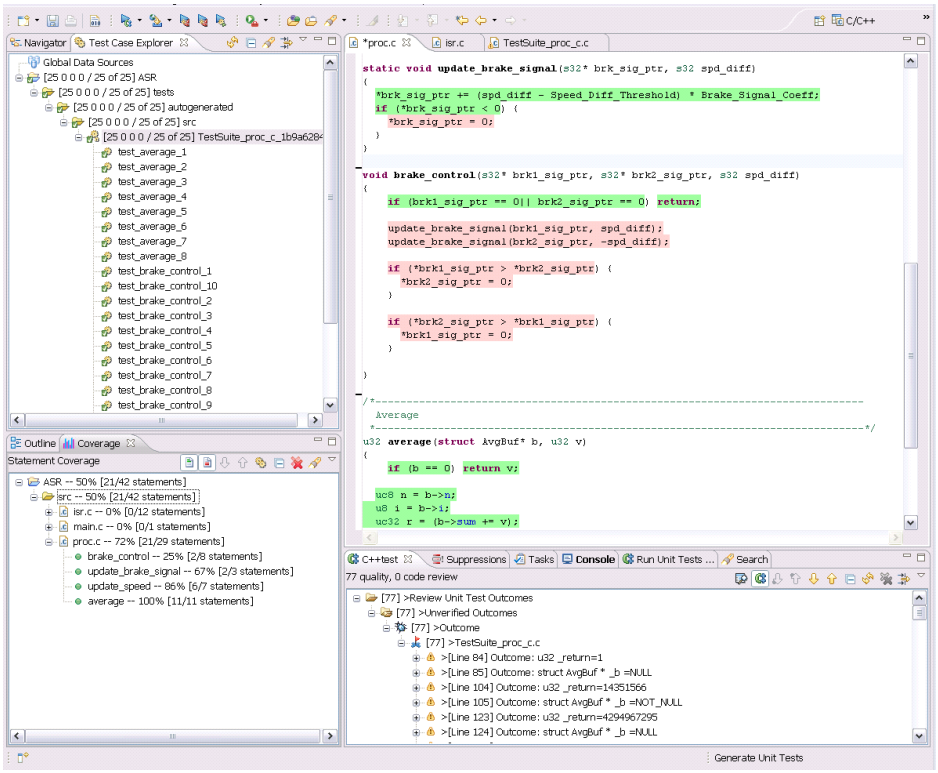
When test cases are first generated, they do not yet contain assertions. Instead, they have statements that are supposed to catch the actual value after the first execution, and report a "Verify Outcome" task. Once these values are verified, C++test will transform those statements into actual assertions. If outcomes are found to be incorrect, the user determines how to proceed: remove the test case, correct the code, enter the assertion manually etc.

Verifying outcomes

The next step is to run the test cases and review the outcomes as well as the actual test cases. If you want, you can review them before running them. However, from practical point of view, it is good to run them first. To do this, we highlight the same file ("proc.c") and choose "Run Unit Tests" from the configurations menu. After less than a minute (including download, execution and upload), we get results: 11 test cases passed and 27 failed with runtime exceptions. When we examine the failures more carefully, we see that they are caused by three functions: "average", "update_brake_signal" and "brake_control." They all take pointers as arguments (as shown in the screenshot below) and some generated test cases are actually passing nulls to them.

Now we have a decision to make. If our code should be super-hyper-safe, then it should not crash when a null value is passed. Otherwise, we can just assume that such a condition would never occur and delete these test cases. We take a hybrid approach. Since the "average" and "brake_control" functions are meant to be called from other functions, we put an appropriate guarding "if" statement. The function "update_brake_signal" is used only from "brake control," so we do not need a redundant safeguard here. Instead, we just delete those test cases by right-clicking on them and selecting the appropriate C++test action.

After running again our test cases, we end up with 25 test cases, which give us 72% statement coverage and 77 outcomes to verify. The screen shot shows a piece of code with our "brake_control" and "average" functions. Statement coverage is highlighted. It is also displayed in a small window in the lower left corner. Test cases are shown on the left side, within the test case explorer. And the outcomes that need to be verified are shown in the lower window panel.



All the work we've performed up to this point has taken just minutes. Now we need to go test case by test case and verify the outcomes. This is unavoidable if you want to have meaningful assertions. The amount of work required is reduced by C++test's convenient views and actions; it can be further reduced if you decide to keep only the assertions for the most vital variables. Moreover, if you know that your code is working well you could verify all of the outcomes en masse, which essentially "freezes" the current state. Then, you carefully examine only most important parts of your code. This all depends on how critical the code under test is. If it affects the safety of your system, you need to check everything.

Extending coverage

After automated generation we are at 72% statement coverage of our “proc.c” file. Our target is 100%, so let’s look at one of the functions and try to increase coverage. “brake_control” has only 25% statement coverage. We can see that it basically exits on the first “if” statement, which we added as a safeguard against null pointers. Obviously, we need to provide a test case that passes non-null pointers to the brake signal variables. We can do this by copying, pasting, and modifying one of the existing test cases, or by using the test case wizard. With the test case wizard, you indicate which function you want to test and the graphical editor guides the set up of pre-conditions, arguments, post conditions and expected values. The method for defining test cases—through the wizard or by copying and modifying—is really a matter of personal preference. With simple types, the wizard tends to be very efficient. With more complex types without factory functions, copy and modify may be more efficient. One thing worth noting about the wizard is that it automatically identifies all variables that are used inside the function and asks for initialization. If you are preparing test case manually, you may initially overlook this. In that case, additional debugging would be needed to figure out why your test case is not behaving as expected.

In our case, it took only a a minute or two to use the wizard to create test cases that pass pointers to the allocated s32 types (“spd_diff” equal to +100 for one and -100 for the other) and set up both the variables “Speed_Diff_Threshold” and “Brake_Signal_Coeff” to a value of 10. After running both added test cases, our coverage for “proc.c” increased to 97%, with both “update_brake_signal” and “brake_control” fully covered. Achieving 100% required one more wizard usage, this time on the “update_speed” function. This also took no more than 1-2 minutes; the function is not particularly complicated and it was easy to figure out a set of variables values that would lead to the uncovered if statement. After that, we achieved 100% statement coverage.

Stubs and drivers

When referring to something that replaces a call to an original method/function with a call to different one, we use the term “stubs”, or sometimes “drivers” (because it drives execution to the place you want).. This is helpful whenever you want to avoid interaction with a real-world world network, database, hardware—or when you want to force execution to go into a specified path. C++test allows you to use the original symbols, generate automated stubs, or provide your own stubs. If you choose to provide your own stubs, it generates a file with appropriate function names, then you can modify this file to provide the stub implementation. From the user stub, you can use the C++test API to figure out where the given stub was called from and alter the behavior accordingly. You can also put assertions to the stub and call the original symbol from with the stub. In our simplified ASR example, we were able to achieve 100% statement coverage without using stubs to drive execution. However, stubs are necessary in many cases. Thus, a convenient stubbing mechanism is indispensible.

Functional testing goals – faults injections

Even though stubs were not required for achieving our coverage goal, they can be helpful in this example: Not to drive execution in unit testing, but to drive execution for the entire application. We could inject a fault into the working system to actually see what this would cause. Imagine that the photo element responsible for measuring the wheel speed suddenly malfunctions. How can we simulate this situation during system testing? We certainly do not want to physically break the photo element because that would be very costly... and it still would not allow us to emulate all the possible types of defects. Instead, we could simulate the faulty behavior by stubbing the functions that are responsible for providing wheel speed data. In our implementation, speed is obtained from the interrupt handler reading from the appropriate port. Stubbing the interrupt handler is not particularly convenient; it would require modifying the vector map. But there is easier way – we can stub functions that use data already obtained by the interrupt handler. In this example, this is the “update_speed” function, which is called from the main loop to calculate the speed from raw data. Then, depending on the actual data obtained from the properly-functioning photo element, we can emulate the faulty behavior: return a very high speed, or zero, or both interleaved.

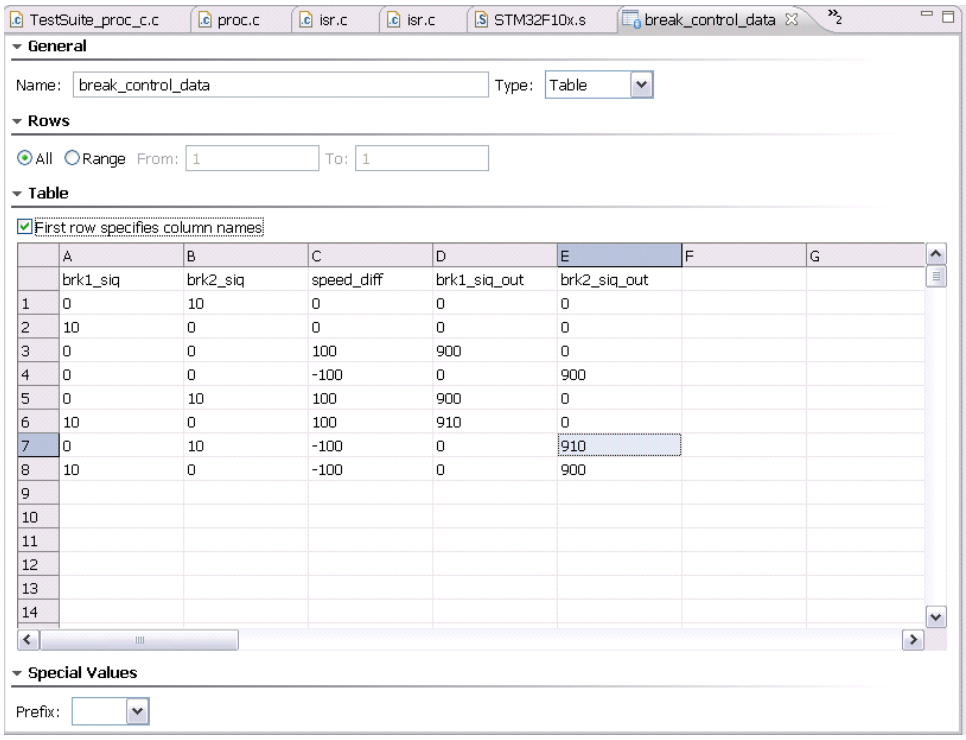
Then, the entire application can be uploaded to the board and run through regular functional, physical tests. The tester can then see , with own eyes, how the system behaves under the simulated faulty conditions. And the code coverage achieved is reported automatically.

Data sources

It is quite common to have a scenario where your code should be tested with many combinations of input parameters. They may be numbers, byte streams, character buffers etc. Instead of preparing many test cases (one for each combination), it is much more convenient to prepare one test case and then feed it values from external data sources. This gives you huge benefits in terms of maintainability. C++test supports several types of data sources: built in tables, CSV files, Excel spread sheets, database connections, and a combination of the above. You just configure the data source of your choice and then create a test case using the test case wizard. Or, you can directly call the C++test API to access values from the data source. In our case, we needed to add two additional test cases to obtain 100% statement coverage on our function brake control. Not bad...but if we had to test more combinations, the number of required test cases would start growing. For path coverage or MC/DC coverage, quote a few more test cases would be needed. Thus, it would be better to prepare one test case and parameterize it with data source values. The test case could might look as follows:

```
CPPTEST_TEST_SUITE(TestSuite_proc_c_1b9a6284);
(…)
CPPTEST_TEST_DS(TestSuite_proc_c_1b9a6284_test_brake_control_ds,
CPPTEST_DS("brake_control_data"));
CPPTEST_TEST_SUITE_END();
(…)
/* CPPTEST_TEST_CASE_BEGIN test_brake_control_ds */
/* CPPTEST_TEST_CASE_CONTEXT void brake_control(s32 *, s32 *, s32) */
void TestSuite_proc_c_1b9a6284_test_brake_control_ds()
{
  /* Pre-condition initialization */
  /* Initializing argument 1 (brk1_sig_ptr) */
  s32 _brk1_sig_ptr_6 = CPPTEST_DS_GET_INTEGER("brk1_sig");
  s32 * _brk1_sig_ptr = & _brk1_sig_ptr_6;
  /* Initializing argument 2 (brk2_sig_ptr) */
  s32 _brk2_sig_ptr_7 = CPPTEST_DS_GET_INTEGER("brk2_sig");
  s32 * _brk2_sig_ptr = & _brk2_sig_ptr_7;
  /* Initializing argument 3 (spd_diff) */
  s32 _spd_diff = CPPTEST_DS_GET_INTEGER("speed_diff");
  /* Initializing global variable Speed_Diff_Threshold */
  {
     Speed_Diff_Threshold = 10;
  }
  /* Initializing global variable Brake_Signal_Coeff */
  {
     Brake_Signal_Coeff = 10;
  }
  {
    /* Tested function call */
    brake_control(_brk1_sig_ptr, _brk2_sig_ptr, _spd_diff);
    /* Post-condition check */
    CPPTEST_ASSERT_INTEGER_EQUAL(CPPTEST_DS_GET_INTEGER("brk1_sig_out"),
*_brk1_sig_ptr );
    CPPTEST_ASSERT_INTEGER_EQUAL(CPPTEST_DS_GET_INTEGER(„brk2_sig_out"),
*_brk2_sig_ptr);
  }
}
/* CPPTEST_TEST_CASE_END test_brake_control_ds */
```

In above example, C++test iterates over rows from the “brake _control_data” data source and calls the “brake_control” function with values from the columns “brk1_sig”, “brk2_sig” and “speed_diff.” Then, outcomes are compared against values from the columns “brk1_sig_out” and “brk2_sig_out.” It’s very simple to create such test case with the test case wizard. One thing which might take time is populating the data source with the correct values. Either you know them off hand, or you have to execute a test case, watch and validate outcomes, then copy them into the data source.



	A	B	C	D	E	F	G
	brk1_sig	brk2_sig	speed_diff	brk1_sig_out	brk2_sig_out		
1	0	10	0	0	0		
2	10	0	0	0	0		
3	0	0	100	900	0		
4	0	0	-100	0	900		
5	0	10	100	900	0		
6	10	0	100	910	0		
7	0	10	-100	0	910		
8	10	0	-100	0	900		
9							
10							
11							
12							
13							
14							

Additional considerations

Associated cost

For the sake of this article, we measured the time required to achieve 100% statement coverage with unit tests on our ASR demo code. From the time the project was configured, it took us less than 10 minutes. Then, the initial generation of test cases took 30 seconds. Reviewing generated test cases took an hour, but that included the analysis and fixes involved in finding several serious problems:

- Some functions were not guarded against passing null pointers (described earlier).
- One function (“average”) was not prepared for receiving very large data. An overflow was detected.
- The same function (“average”) was not guarded against improperl- initialized structures.

Then another hour was required to create test cases for the remaining code in order to achieve 100% statement coverage. The test case wizard was used here, as were data sources. No stubbing was necessary.

Within less than 3 hours of total time, our code was fully covered with 100% statement coverage (SIL 2), 79% path coverage (SIL 3), and 59% MC/DC coverage (SIL 4). In addition, problems in the code were detected and fixed. 25 quality test cases were created—two of which were iterating over easy-to-populate data sources.

Memory monitoring

C++test provides additional benefits beyond efficiency gains. First, all the test cases prepared can be run under memory monitoring. To do this, C++test instruments code for monitoring its execution and then runs it. If it detects any reading/writing out of bounds, using uninitialized variables, leaks, or similar runtime defects, they will be reported. You don't have to perform any extra work, and you get this runtime error detection as the code executes on your target or simulator!

Regression test suit

Another bonus is that we built a regression test suite for our simplified ASR code. A regression test suite provides a safety net for whenever you want to modify your code. You can think of it as a large number of sensors planted into your code. When something goes wrong, some sensors should be able to catch it, and you will be alerted to the problem. Having a large regression test suit gives you much more freedom to modify your code. This becomes particularly valuable when you need to fix code that is already in production. In this case, the last thing you want to do is cause more harm than good when providing fixes. When you do unit testing, you are establishing such regression test suite at the same time. This is truly an extra bonus!

You need to keep in mind, though, that unit test cases are usually on a very low level. They are not meant to replace regular functional testing, but rather to extend it. And the more "functional" your test cases are, the better they are. In other words, you should strive to create test cases that actually emulate real life situation, not emulate artificial ones just to get some extra code coverage. Creating such test cases requires slightly more effort, but this pays off when it comes to maintaining the test suite. Robust test suites tend to catch real, significant problems. Just imagine how much less scary it would be to introduce a fix into your code if you knew that you could instantly run hundreds, if not thousands, of test cases to ensure that your fix did not introduce any undesirable side effects. That confidence is invaluable.

Reporting

Last but not least is the final benefit: reporting. It is nice to know that you achieved 100% statement coverage and that all test cases passed...but if you want to certify your code, you need proof demonstrating that you achieved this. For this purpose, C++test can generate reports (including coverage details_ that you can provide to the auditor for certification purposes. Or, you can just keep these records on hand to prove that you did all you could to deliver good, quality software.

Functional Safety relevance of Unit Testing

In the previous paragraphs we made several references to the issue of certification in relation to functional safety. Indeed, this is one of the key issues of today's and tomorrow's electrical/electronic/programmable electronic systems. New functionalities increasingly touch the domain of safety engineering. Each function that is required to keep a risk at an accepted level is called a *safety function*. To achieve functional safety, these functions need to fulfill safety function requirements (what the function does) and safety integrity requirements (the likelihood of a function behaving in a satisfactory manner). Future development and integration of the functionalities containing safety functions will further strengthen the need to have safe system development processes and to provide evidence that all reasonable safety objectives are satisfied.

With the trend of increasing complexity, software content, and mechatronic implementation, there are rising risks of systematic failures and random hardware failures. An international standard, IEC-61508, includes guidance to reduce these risks to a tolerable level by providing feasible requirements and processes.

Safety Integrity Levels

Safety Integrity Level (SIL)—as defined by the IEC-61508 standard—is one of the four levels (SIL1-SIL4) corresponding to the range of a given safety function's target likelihood of dangerous failures. Each safety function in a safety-related system needs to have the appropriate safety integrity level assigned. An E/E/PE safety-related system will usually implement more than one safety function. If the safety integrity requirements for these safety functions differ, unless there is sufficient independence of implementation between them, the requirements applicable to the highest relevant safety integrity level shall apply to the entire E/E/PE safety-related system.

According to IEC-61508, the safety integrity level for a given function is evaluated based on either the average probability of failure to perform its design function on demand (for a low demand mode of operation) or on the probability of a dangerous failure per hour (for a high demand or continuous mode of operation).

The IEC-61508 standard specifies the requirements for achieving each safety integrity level. These requirements are more rigorous at higher levels of safety integrity in order to achieve the required lower likelihood of dangerous failures.

Unit testing and Safety Integrity Levels

The relevance of unit testing, or more precise, C++test unit testing related functionality, to particular Software Integrity Levels as defined by IEC-61508-3 is summarized in the following table. The following markers are used in the tables presented below:

For the sake of this article, we measured the time required to achieve 100% statement coverage with unit tests

The following markers are used in the tables presented below:

- R – Indicates functionalities matching methods recommended by the IEC-61508 standard
- HR – Indicates functionalities matching methods highly recommended by the IEC-61508 standard

The descriptions reference the appropriate techniques/measures as defined by the IEC-61508-3, Annex A. For example, (Table A.3: 1) references IEC-61508-3, Table A.3, Technique 1.

C++test functionality	SIL			
	A	B	C	D
Unit testing module – general				
Unit test execution (Table A.5: 4, Table A.7: 3)	HR	HR	HR	HR
Automatic unit tests generation module				
Automatic unit test generation using boundary values (Table B.2: 1, Table B.3: 3)	R	HR	HR	HR
Using factory functions to prepare sets of input parameter values for automatic unit test generation (Table B.2: 5)	R	R	R	HR
Automatic unit test generation using random input combinations (Table A.5: 1)		R	R	HR
Test management module				
Using user-defined test cases – both manually-written and created using Test Case Wizard – to test specific atomic cases of the given requirement (Table A.5: 4, Table A.7: 3)	HR	HR	HR	HR
Using Data Sources to efficiently provide multiple inputs for functionally equivalent atomic cases of the given requirement (Table A.5: 4, Table A.7: 3)	HR	HR	HR	HR
Using Test Case Explorer for managing test cases and reviewing test case status (Table A.5: 2)	R	HR	HR	HR

C++test functionality	SIL			
	A	B	C	D
Function stubs				
Using stubs to control the flow of the executed tests as specified in the given requirement (Table A.5: 4)	HR	HR	HR	HR
Using function stubs to substitute user interface for automatic unit test execution (Table A.5: 6)	R	R	HR	HR
Using stubs to provide fault conditions in tests (Table B.2: 2)	R	R	R	R
Coverage module				
Analyzing statement, branch, and MC/DC code coverage for structure testing (Table B.2: 6)	R	R	HR	HR

Note that C++test can:

- Run the unit tests in both instrumented and non-instrumented mode—for example to show that coverage instrumentation does not impact the test results.
- Execute unit tests in the production environment on a target device or on a simulator.

It is also worth noting that the second edition of IEC-61508 goes further with testing requirements than first edition. For example, in table B.2 in the first edition, technique 6 references structure-based testing in general. In the second edition, table B.2 has new rows (7a to 7d) that reference specific structural test coverage types and demand 100% coverage, depending on the SIL number.

Other Safety Standards

IEC-61508 is not the only functional safety related standard. Some of the others are derived from it to address particular industry specifics, while others were developed independently. Some are more strict (for instance, those related to airborne systems) while others are more relaxed. The underlying concepts, though, are similar, so unit testing would prove indispensible almost everywhere. Discussing all functional safety related standards is far beyond the scope of this article, but we briefly mention few below, just for reference. For more details on a particular standard, see the related reference documents or contact specialists in that domain.

ISO/DIS-26262 – This is the adaptation of IEC-61508 to comply with needs specific to the application sector of E/E systems within road vehicles. The Draft International Standard (DIS) is the latest version; it has been publicly available since June 2009. The International Standard (IS) is planned for June 2011.

ASIL (Automotive Safety Integrity Levels) – This is an equivalence of SIL defined by the ISO/DIS-26262 standard. It specifies the necessary safety measures for avoiding an unreasonable residual risk, with D representing the most stringent level and A representing the least stringent level.

DO-178B/C – "Software Considerations in Airborne Systems and Equipment Certification" is a standard for software in airborne systems and equipment used on aircraft and engines. It is an industry-accepted guidance for satisfying airworthiness requirements.

IEC-60880-2 – This is the adaptation of IEC-61508 used in safety systems of nuclear power plants.

EN-5012X/EN-50128/EN-50129 – This is the adaptation of IEC-61508 used for rail transportation.

Wrap up

Admittedly, unit testing is not free. Work is required to set it up properly, and time and effort are required to maintain it effectively. For embedded systems software development, unit testing presents additional challenges, which can be overcome in the ways discussed in this article. You need to understand this before you start; otherwise, you're likely to be disappointment.

On the other hand, unit testing can give you huge benefits, such as helping you to create better code, build a regression test suite, achieve a desired Safety Integrity Level, or obtain DO-178B certification.

Fortunately development platforms vendors such as Keil recognize the growing demand for good software testing and co-operate with testing tool vendors such as Parasoft. Together, we provide a solution that significantly reduces the burden of unit testing: both in terms of preparing, extending, and maintaining test suites and in terms of running the tests and collecting results. To read more about the Parasoft/Keil solution, please see http://www.keil.com/ulinkpro and http://www.parasoft.com/product/cpptest/

PARASOFT®

USA PARASOFT HEADQUARTERS / 101 E. Huntington Drive, Monrovia, CA 91016
Phone: (888) 305-0041 / Email: info@parasoft.com