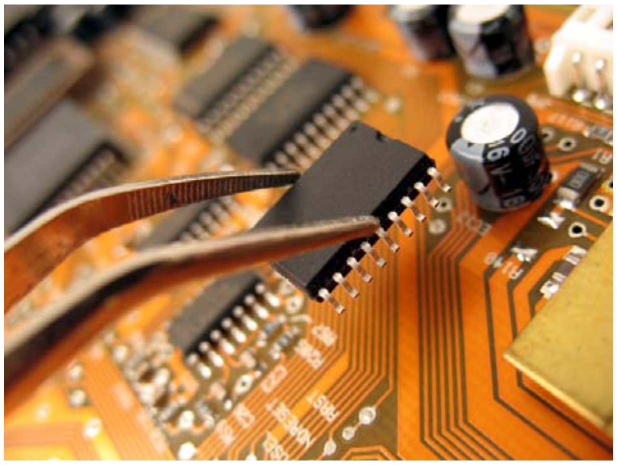
Feature

Proven Practices for Testing Embedded Software

Build the right test platform including infrastructure, virtual lab and process

By Andrey Pronin, Auriga



Testing embedded software is both similar and dissimilar to application software testing. The first eye-catching thing is that embedded software is significantly less visible to the end user. User interfaces are limited; there may be a console-based text menu, a simple command line interface, a set of digital inputs of outputs, or something similar, but rarely do we get more than that. On the other hand the inter-component interfaces can be very rich and complex—including APIs to the higher-level software, implementations of various communication, data exchange, control, and other standards, etc. Thus the main focus of embedded software testing is not on testing the user interfaces, but on testing the components not visible to the end users.

The second major difference is the level of the dependence on the hardware specifics. Embedded software is the level of the software closest to the hardware. Other software types such as operating systems and applications may be built upon the interfaces provided by the embedded software such as BIOS or boot loader. The embedded software itself, even if it uses some more or less standard framework underneath, needs to care more about hardware details. Embedded software by definition is designed for a particular hardware unit (or a set of hardware units in common case). Often, those hardware units are developed in parallel with the embedded software. The created software is the first to run on it.

Unlike application development, in the embedded world we can't rely on the fact that the operating system is already tested on the hardware platform, or that the ability of the hardware itself to execute various software has already been thoroughly tested. As a result, the developed software may have solutions and workarounds specific for particular hardware revisions.

Operation of the embedded software may depend on such things that we usually don't care about for application-level software, like the length of the cable, type of the mouse, serial port frequency, or type of the devices connected to the same bus that makes the successful execution of embedded software degrees more dependent on the particular hardware unit and on the behavior of the other units in the same bus or network.

Compared to conventional cases, race conditions are mostly caused not by the interaction of the internal software components, but rather by the interactions of the software with the environment. So, the number of factors and parameters that can influence the operation is bigger than for the average application. And reproduction of a defect is more difficult. Support operations, such as software deployment, upgrade, getting debug information, also differ from what we usually see in conventional application-level software with plug-n-play concept, installation wizards, ability to attach a convenient debugger from one of the IDEs, or at least dump all debug output lines to a large file on disk.

In the embedded world we often need to put the software in a special mode, disable EEPROM write -protection, attach to some file-distribution (like TFTP) server, reboot a couple of times, and care about other similar things. That makes the software update process lengthy and inconvenient. And, it might be that the device that stores your software supports only a limited number of rewrite cycles.

During active development phase, software versions tend to be updated less frequently than for the other forms of software. New revisions are typically deployed only after a significant number of defects are resolved. Thus, the testing process should attempt to find as many defects as possible, and not stop after the first one, even if it makes the product crash.

Embedded Software Testing Challenges

The specifics of the embedded software domain imply certain requirements for the organization of the testing process. The focus on non-human interfaces leads to the fact that, we can't use a manual interface testing approach.

To test the developed embedded software, we first need to develop special applications—test agents—need that provides stimulus and captures response through the non-human interfaces. It is also often necessary to emulate particular electrical signal patterns on various data lines to test the behavior of the embedded software for such inputs. It can be done using special hardware/software complex along with a built-in special test agent to control that complex.

A high level of hardware dependency and the fact that the embedded software is often developed in parallel with the hardware leads to several important consequences. First, there may be only few samples of the newly developed hardware. Second, the range of the hardware unit types to test the software on can be quite wide. Thus, typically the testing team has to share a very limited set of hardware units among its members and/or organize remote access to the hardware. In the second case, this means that the testing team has no physical access to the hardware at all.

Another aspect of having the software developed for a freshly created hardware is a high ratio of hardware defects that can be discovered during the testing process. Any discovered defect may be related to either the hardware or the software. Always keeping that in mind is especially important for embedded software projects. What's worse, the software may work just fine with one version of the hardware, but not so well with another.

We have already mentioned that defects are harder to reproduce in the embedded case. That forces the embedded testing process to value each defect occurrence much higher than in a conventional case and attempt to gather as much information as possible rather than simplify looking for the root of the defect. That, combined with the very limited debug capabilities of the embedded products, gives us another challenge.

Limitations related to software updates requires persistence with the testing process to discovering as many bugs as possible for a given software revision. It also increases the importance of build and deployment process.

A high level of requirements on the robustness/availability front leads to the need for very thorough stress testing. A consequence of that fact is the need to emulate the sequences of rapid-follow events to check for race conditions under those circumstances.

Automated vs. Manual Testing

First of all, it is obvious that using manual testing as the main method for embedded testing projects is very difficult, if not impossible. Routine, time-consuming, repetitive stress testing, working with non-human interfaces, the need to discover race conditions for fast-sequence events, and a host of other factors all complicate the task. Thus automated testing is a cornerstone approach.

Of course there will always be a percentage of tests that is more cost-effective to run manually than automate. But that percentage is smaller than usual, dictated by higher relative efficiency of automation in remote access environment (the alternative to which is organizing a trip to the remote lab) and special supporting means described later. In any case, automation is typically done for more than 95% of the test cases. Having stated that, it important to understand that automation and usage of test agents doesn't simply change the way of executing the test cases and presenting results, it affects all aspects of the testing process.

Test Design and Tracing Requirements

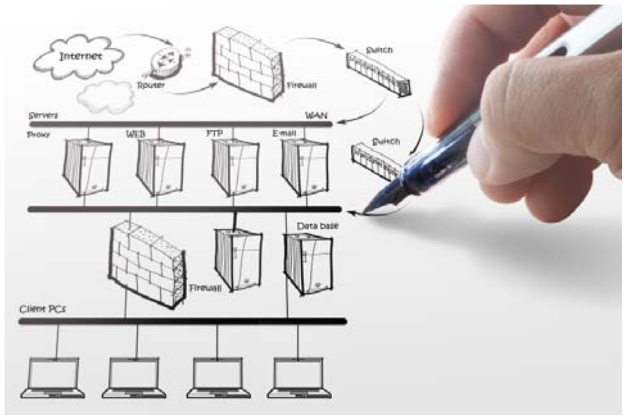
Two things must be understood. First, a great number of the test cases created for the embedded software simply cannot be executed manually. Thus a straight forward test design approach—get requirements; design test cases; run manually; optimize; fix; detail; create script based on the manual case—doesn't work here. Second, unlike regular methodology, the software requirements specification does not lead to, and is not traced to, just the set of the test cases.

Instead, based on the software requirements of the embedded software, two artifacts are created—the set of the test cases and the requirements for the test support infrastructure which consists of the automation framework and test agents. In the formal sense, the embedded software requirements are traced to the test cases, which in turn are traced to the software requirements for the agents and framework. But from the practical perspective, test cases and support software requirements cannot be separated.

Validation of The Test Support Infrastructure

The second influence on the testing process is in the fact that the support software must itself be validated. Basically, that means that first, the test agents and the automation framework must be tested themselves—test design, execution, coverage analysis, and all other activities are performed for them as well. Test agents are typically relatively simple software entities with a limited set of requirements, so testing them is significantly simpler than testing the original software product. Still, they often need to implement complex data exchange protocols (including encryption, authentication, compression, connection establishment, and what not), so testing them is not at all simple.

Complete testing of the test agent is often impossible without having more-or-less, a working version of the target process. So, passing tests for a test agent also means passing basic functionality tests in a particular area for the target software.



During this testing, previously verified test agents and hardware debugging tools—bus analyzers, network sniffers, JTAG probes, and oscilloscopes—are extensively used. The hardware debugging tools are especially useful at this stage of achieving a basically functional application. This has another natural implication on the embedded software development process. The design of the test support tools is done parallel with the target embedded software design, and the development plans for the target software and test agents are highly dependent.

The second component of the test support infrastructure, automation framework, also obviously requires validation. However, unlike the test agents, which perform functions specific to a particular embedded product, it can, and should be, designed and implemented as project independent, at least inside some wide technological or organizational segment. That saves a great amount of testing effort and doesn't need to be repeated for every next project.

Defect Tracking and Analysis

Besides the direct verification and validation effort, the need to validate the test support infrastructure also influences the defect lifecycle and defect tracking repository setup. For embedded software several possible origins should be considered for each defect: the target software, the underlying hardware and the test support infrastructure. One example of the practical consequences of that leads to specifying target software, hardware, and test support suite IDs in every discovered defect record. Another example is including the representative of the test support infrastructure development team in the triage committee for the project.

For hardware-caused defects, the testing team must include a person with hardware engineering skills along with knowledge of using various hardware debugging tools mentioned above. This person should also be included in the triage committee to examine each defect from the point of view of probability for it to be of hardware origin. This person will also provide guidance to the team regarding the suspicious signs in hardware behavior and gather additional data for analysis if a hardware defect is suspected.

Hardware Coverage Matrix

A higher probability of a hardware defect doesn't lead just to the need to specify hardware ID in the defect record and having a hardware engineer on the team. The target software must also be tested on the range of the possible target hardware types and revisions. That doesn't mean that each test case must be run on all possible hardware units/types/revisions. A conscious choice between the coverage and cost/time must be made. It is often possible to combine hardware units in groups for testing each functionality area, or at least perform random selection for regression testing purposes. The test strategies defined for different projects may vary in this aspect based on the project constraints and requirements.

In any case, a hardware coverage matrix is required. All "test case and hardware unit" combinations that should be verified are marked in this matrix. For obvious reasons, automation framework should allow for specifying and changing the matrix without affecting the bodies of the individual test cases.

Software Builds

Establishing the right build and deployment process is also essential for the success of the embedded software testing task. It is important to correctly identify the target software revision, for which a defect is revealed. Several techniques are used to address the issues related to the software build identification.

One of the useful practices is obtaining the build number from the running target software at the beginning of the test suite execution—the embedded software that has some user interface often allows getting that information. Using this practice prevents incorrect identification of the version in defect records, if a test suite was run against the wrong version by mistake.

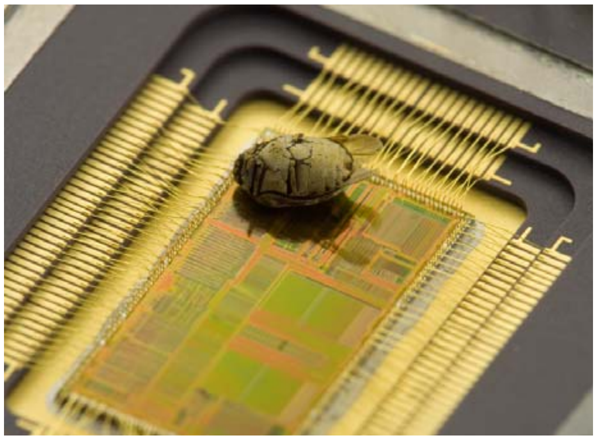
Another practice is used for the smoke tests of regular software releases. According to the practice, the test support infrastructure contains all necessary tools for making the build, assigning it a unique number, tagging the source tree, archiving the binaries, transferring the binaries to the deployment server (e.g. TFTP server) and then to the target board, and updating the software on the board. Such operations may be performed at the beginning of the overnight smoke test for a regular build. For the projects with no limitations on the number of software updates for the target hardware unit, this operation can be performed completely (build and deploy on the board) or partly (deploy only) before every build to ensure the right version to be used during the testing.

Debug Support

One of the goals of the good testing process, besides revealing as many defects as possible, should be assistance to the developers in resolving the defects. A defect found by the testing team that can't be reproduced by the development team and thus can't be fixed due to insufficient information provides little value.

As stated, in the embedded world the defects are harder to reproduce, thus as much information is possible should be gathered on the first occurrence. Due to the fact that debugging is also more difficult for the embedded software, the development team often uses special debug builds or special debug modes of the target software with increased logging capabilities.

There are two implications of this situation for the testing process. First, the timing and other characteristics of the debug and release versions of the target software may differ, and the defect seen on one version may never be seen in a different version. Thus it is important to keep track of the software revision, for which the defect was discovered by testing.



Second, the test cases should be designed to allow using these extended capabilities of the debug version or mode. When a defect is revealed the test case should store the debug output of the software in the test log tied to the test result, so that a developer assigned to resolving the defect can use this data during the analysis. The test case should also be able to detect the type of version of the target software—debug or release, or switch between the modes. The details of that are highly project-specific and are usually implemented either through the parameters passed to the test case, or by employing a specialized test agent.

Test Runs

Due to the contradicting characteristics of the embedded software product, there are two types of test runs employed for it.

An ideal method is to batch-run test cases. All selected test cases are run according to the hardware coverage matrix, and the results are stored in the test log. If an error is detected, the test run doesn't stop, but rather all possible information about the system state at the time the defect was discovered (and all debug support techniques are important here) is captured, and testing continues with the next test case. Needless to say, the test support framework should perform a complete clean-up after each test case to avoid influence between the test cases in general, and a series of failed cases after the first crash in particular. Such clean-ups often include system reboot, typically software reboot after a successfully completed test case, and a hardware reboot after a failure.

Such test runs are lengthy; the required time further increases the need to clean up. Due to the length of time these runs are typically scheduled to be performed in automatic mode overnight. Such batch runs are especially useful as smoke/ regression tests for new builds.

In certain cases tests are run until the first failure. Then the test run is stopped and the system state is preserved. A developer is then notified and allowed to examine the system state in detail to reveal the root cause of the failure. It is also possible to create an automation framework that would break the test run only if the failure occurred in a particular test case (or a set of test cases). Such test runs are useful for hunting down defects, for which information gathered in the batch mode is insufficient and a developer needs to get access to the system at the moment of defect to investigate it.

Virtual Laboratory

The methodological approaches described in the previous sections allow forming the testing process relevant to the specifics of the embedded software testing. However, there is another important part of the approach—a software and hardware solution, called Virtual Laboratory, or VL. This solution provides the means for solving several technically complex problems faced during testing.

First, it contains a database of the existing hardware units. The units are identified by simple string IDs. For each unit, it is possible to specify several properties, such as hardware revision, communication IDs—IP address, MAC address, login credentials, etc. For a test script this means that by passing a unique unit ID as a parameter, it can restore all other parameters that are required to communicate with this board and provide complete defect reports.

Second, a VL supports serial consoles, power bars (devices allowing switching the power on and off for the target units), and dry contact controllers (relays). Console/relay/ power bar lines are associated with a particular unit in the hardware unit's database and as a result, all operations with a particular unit are performed by the test scripts based on the name of that unit.

Third, a VL provides a means for ensuring exclusive access to the shared hardware. Before accessing a unit's console the test script must first 'lock' that unit using a special command. While the unit is locked, no other entity can 'lock' it. After all testing actions are performed, the test scripts 'unlocks' the unit, allowing others to control it. Such exclusive locking mechanism prevents interference of different test scripts and human operators attempting to run test on the same board simultaneously.

A VL provides human-friendly command-line interface over secured connection, and can be used both by test scripts and human test operators. A VL serves the base for executing all automated and manual tests for the target software.

Summary

Testing of embedded systems can be quite complex and defects can be quite hard to produce. Taking time to create the right test environment and then creating the right level of testing will help produce the very best test environment.

About Andrey

Andrey has a Ph.D. in applied mathematics and has spent over 20 years in various roles related to software development. He is currently SVP of Strategy, Technology and Marketing at Auriga, and a board member of RUSSOFT, association of software development companies. Auriga is a leading software engineering services provider (www.auriga.com)

The Top 10

Automation Issues to Solve for Embedded Software Testing

1. **Physical access to the embedded system to play tests or get results.** Sensor and other hardware interfaces may need special case access.
2. **Support for test automation in the product itself.** Hooks or callable APIs may need to be added to the code.
3. **Behavioral verification.** Recompiling embedded code on a PC often affects the runtime behavior.
4. **Hardware availability.** At least run verification tests directly on hardware. Automation on a simulator does not replace testing directly on the device.
5. **Security**. Any testing (or agent) code in the embedded system, in particular in a production version, should not open a gate for hacking.
6. **Check for timing issues**. Some devices have time dependent controller software.
7. **Availability of expert team members**. Teams need people who both understand the technologies and are versed in testing.
8. **Automating multi-media aspects**. Sounds being played or LED flashing needs to be tested.
9. **Memory constraints**. There are situations where available RAM may be too low causing system failure.
10. **Non-determinism of the system under test**. Due to any number of reasons (race conditions, random number generators, state) a system may behave differently in different runs making pass/fail criteria and test coverage difficult.

LogiGear