# Testing of Embedded Software Products

**Abstract**

*The paper is devoted to organizing testing process for the embedded software products, and suggesting useful techniques in this area. Some details discussed in the presentation are specific to offshore development and testing teams*

***Keywords:*** *Testing, embedded software.*

## Introduction

This paper is devoted to the questions of organizing the testing process for the embedded software products. Embedded software domain has its own specifics that influence all components of the development process. Testing is not an exception. To understand what practices and techniques work best for this domain, we should first understand what the main differences of the embedded software from other software types are. This analysis is done from the point of view of the offshore team. For in-house team some details differ, still the overall approach remains the same.

## Embedded software domain specifics

The first eye-catching thing is that the embedded software is significantly less visible to the end users. 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. 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. Thus, unlike application development, in the embedded world we can't rely on the fact that the operating system is already tested on that hardware platform, or that the ability of the hardware itself to execute various software is already 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 the 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 embedded software to a much higher degree dependent on the particular hardware unit and on the behavior of the other units in the same bus or network. Compared to the conventional case, 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 are used to see for conventional application-level software, with its plug-n-play concept, installation wizards, ability to attach a convenient debugger from one of the IDEs, or at list dump all debug output lines to a large file on disk. In the embedded world we often need to put the software in the 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. Besides, it might happen that the device that stores your software supports only a limited number of re-write cycles.

Thus even during the active development phases the software versions tend to be updated less frequently than for the other forms of software. The new revision would typically be 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.

Talking about the difficulties with debugging and gathering additional information for defects, embedded software development process differs in those aspects as well. No or limited non-volatile storage, inability to run debugger on the same system, often reduced version of underlying operating system or no OS at all what leads to the absence of remote debugging methods, that makes the debugging task much harder than for the conventional application-level software.

As the last but not least characteristic of the embedded software domain, I should mention a high level of robustness typically required from that software. Serving as a base for higher level application software, and working in the environment that doesn't allow for high maintainability, embedded software is very sensitive for the robustness level.

## Embedded software testing challenges

The specifics of the embedded software domain imply certain requirements for the organization of the testing process. Let's quickly reiterate through the specifics and understand what challenges they mean for the testing process.

Focus on non-human interfaces leads to the fact that regardless of our attitude to that, we can't use manual interface testing approach. To test the developed embedded software we need first to develop more software. Special applications, test agents, need to be created to provide stimulus and capture response through the non-human interfaces. It is also often required 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 and that also implies having a special test agent to control that complex.

High level of hardware dependency and the fact that the embedded software is often developed in parallel with the hardware lead 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 our 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, that means that the testing team has no physical access to the hardware at all.

Another aspect in 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 the hardware, not only software. Always keeping that in mind is especially important for embedded software projects. What's worse the software may work just fine with one revision of the hardware, and doesn't work with another. That's definitely a challenge for the testing process.

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 to simplify looking for the root of the defect. Combined with the very limited debug capabilities of the embedded products, that gives us another challenge.

Limitations related to the software updates make the testing process very persistent in discovering as many bugs as possible for a given software revision. It also increases importance of build and deployment procedure organization.

High level of requirements on the robustness/availability front leads to the need in very thorough stress testing. Another consequence of that fact is the need to emulate the sequences of rapidly following events to check for race conditions under those circumstances.

## Embedded software testing approach

As it is clear from the previous paragraphs, testing process in the embedded software case faces a number of specific challenges. This section suggests an approach that satisfies all the needs listed above and that has proven its viability based on the results of implementing it in the real-life embedded software projects performed by Auriga.

## Automated vs. manual testing

First of all, it is obvious that using manual testing as the main method for such projects is very difficult, if not impossible. Routine, time-consuming, repetitive stress testing, work with non-human interfaces, need to discover race conditions for the fast sequences of events, and some other factors—all stand against that. Thus automated testing is the first cornerstone of the approach.

Of course, manual testing as always has its percentage of tests that is more cost-effective to run manually than automate. But this percentage is smaller than usually forced 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 this chapter. In any case, automation is done for more than 95% of the test cases, and we'll mostly concentrate on it in this section.

Having said that, it must be mentioned 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 straightforward 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 consisting 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 agents is often impossible without having more-or-less working version of the target process. So, passing the 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. That 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 that can be done only once 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, the test support infrastructure. As one of the examples of the practical consequences of that, it 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 engineer skills and skills in using various hardware debugging tools mentioned above. This person is also included in the triage committee, examines each defect from the point of view of probability for it to be of the hardware origin, provides guidance to the team regarding the suspicious signs in hardware behavior and gathers additional data for analysis if hardware defect is suspected.

## Hardware coverage matrix

A higher probability of the hardware defect doesn't lead just to the need to specify hardware ID in the defect record and having a hardware engineer in 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, the hardware coverage matrix is created. All "test case—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. As it was mentioned, 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 from it 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. The value of a defect that was seen by the testing team, but then could not be reproduced by the development team, and thus could not be fixed due to insufficient information, is low. As we discussed, 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 for the other version. Thus it is important to keep track of the software revision, for which the defect was discovered by testing. This topic is discussed separately in this paper.

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.

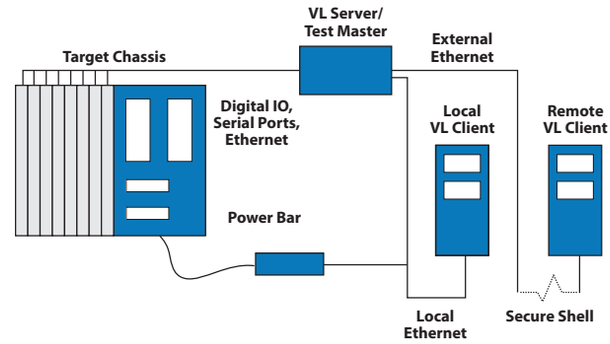
As it was said, it is often beneficial to reveal as many defects as possible for the deployed version before updating the software revision. An ideal tool for that is batch run of test cases. All selected test cases are run according to the hardware coverage matrix, and the results are stored in the test log. If an error is detected, the test run doesn't stop, but rather captures all possible information about the system state at the time when the defect was discovered (and all debug support techniques are important here) and 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 hardware reboot after a failure.

Such test runs are lengthy, the required time further increased by the need to clean up, and 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, the second type of the test run is used. The tests are run until the first failure, and if a failure occurs, the test run is stopped, the system state is preserved, and a developer is notified and allowed to examine the system state in details to reveal the root cause of the failure. It is also possible to create an automation framework that would break the test run only of the failure occur in a particular test case (or a set of test cases). Such test runs are useful for hunting down the 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 the 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 that means that by a unique unit ID passed as a parameter, it can restore all other parameters required for communicating with this board and providing complete defect reports.

Second, 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 units 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, VL provides means for ensuring exclusive access to the shared hardware. Before accessing a unit's console, toggling the power or some other relay for the unit, 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.

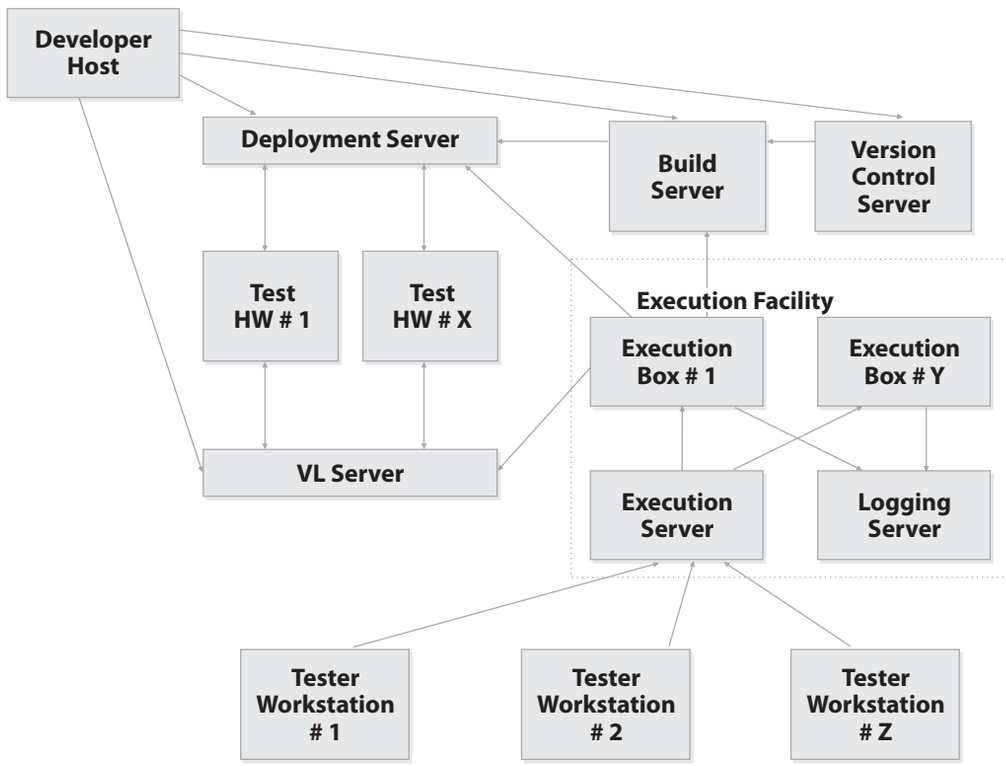
VL provides human-friendly command-line interface over secured connection, and can be used both by test scripts and human test operators.

VL serves the base for executing all automated and manual tests for the target software.

## Automation framework

The description of the test support infrastructure would be incomplete without a brief description of the automation framework. As it was already said, besides the test agents the test support infrastructure contains also the automation framework. This framework contains a library of methods called by the test scripts. All typical, project-independent operations performed by the test scripts should be moved there. Re-using this framework for subsequent projects allows saving a lot of effort on automation and validation of typical operations.

The majority of those typical operations were already



discussed in one or another form in the previous sections. Here we provide a compilation of the highlights. The framework includes methods for performing the following key tasks:

- Defining the set of test cases and hardware targets for a particular test run
- Obtaining unit parameters based on unit ID
- Accessing serial console of the specific unit, performing login with correct credentials if applicable
- Controlling the power and relay channels for a particular target
- Invoking test agents with certain parameters to send raw or predefined messages through different protocol channels
- Obtaining parameters from the target unit—software revision, debug state, etc
- Obtaining debug logs from the board
- Clean reset of the target
- Recording the results of the test case execution
- Generating test results report
- Controlling the test run execution—breaking the test run on test failure or continuing with further test cases depending on the settings
- Building tagging, archiving software
- Deploying software on the target board

## Test environment

The diagram on this page represents the overall architecture of the test environment. It consists of the boxes representing logical functions of various test environment units and interfaces between them. It must be noted that logical functions are not always directly mapped to the physical components. E.g. a single physical server can play the role of build server and version control server simultaneously.

According to this diagram (we start from the bottom), testers connect from their personal workstations to the execution server.

Execution server is a dispatcher. Its main goal is to distribute testing jobs between the execution boxes. Very often it serves as the authorization gate to the whole testing facility.

Logging server is the centralized data storage center, all test logs are dumped there for analysis.

Execution boxes are the systems which execute automatic tests. The main reasons for separating execution boxes from tester workstations are the following:

- Execution box on which the tests are run may have performance requirements different from the tester's workstation configuration. Also, once run on the execution box the test scripts are not affected by the further activities on the tester's workstation. Thus, recording a DVD with the new build on the tester's workstation, or shutting down the workstation for the night, won't affect the test script in any way.
- In case of the remote access to the hardware, execution boxes reside in that remote lab along with VL server and target hardware units, and thus its communication with them is not a subject to Internet-caused delays, which otherwise could affect the test results.

Build server and version control server are used for making the product builds and storing the version control repository respectively.

The role of the deployment server was already discussed. It allows uploading the new version of the software to the target hardware units, using a product-specific interface (e.g. TFTP).

VL server is used as a gate. It contains the hardware units database, and is the only entity that can directly access the hardware units. Other entities use its exclusive locking mechanism to ensure proper usage of shared resources.

Development host depicted on the diagram is used as an example of access provided to other members of the development team, involved in ad hoc testing and debugging activities. It communicates with build, version control, and deployment servers to put a new software version on the target board, and with the VL server to control the board operation.

## Summary

The test approach presented in this section has the following main components:

- 95+% automated testing
- Virtual laboratory solution
- Re-usable automation framework simplifying typical operations
- Practices and techniques
  - Traceability of the target software requirements to the test cases and test support software requirements
  - Validation of the test support infrastructure
  - Tight integration of the target software and test support software development plans
  - Usage of hardware debug tools, especially during early stages
  - Analysis of 3 origins for every defect: target software, hardware, test support infrastructure
  - Specifying 3 IDs for every defect: target software, hardware, test support infrastructure
  - Hardware engineer in the team
  - Defining hardware coverage matrix
  - Double-checking target software version by reading it from the unit
  - Automatic software build and/or deployment as a part of test execution (optional)
  - Support for extensive debug builds/modes in automation framework
  - Two types of test runs: non-stop batch, defect hunting