

Advantages and Aspects of Embedded Software Testing

Anurag Kashyap, Sujay Mishra

MIT College of Engineering & Management, Hamirpur, HP

Corresponding author's Email id: -anurag_kash1000@rediffmail.com

Abstract

This paper offers a general introduction to testing embedded systems, including a discussion of how embedded systems issues affect testing process and technologies, and how Rational Test Real Time provides solutions for these issues.

Keywords: *Embedded Software Testing, Software Testing, Testing Tools*

INTRODUCTION

Embedded systems today represent some of the most exciting pieces of technology in the overall technological market. They represent a different way of thinking from conventional technology and therefore have an efficiency rating that can end up allowing companies all over the world to do a lot more with a lot less. For this reason, embedded systems are in high demand all over world, and embedded technology companies have increased dramatically in order to meet this new demand. However, one thing that embedded systems need in order to perform well is testing. Embedded software testing is a discipline that is both easier and more efficient if it can be conducted concurrently with software

development, and preferably with the hardware. The extent to which this can be done varies.

Testing embedded systems, or closed hardware environments where users don't realize the embedded interface exists, is challenging and different because there is no user interface (UI). Instead, the tests involve digital output, interfaces, APIs, data exchange messaging or other forms of communication. It's also challenging because the testing is not within a UI; instead, it involves components not visible to the end user. Another difference from UI testing is the general difficulty of reproducing defects, especially the longer the time between when a defect is reported and the time it is fixed.

Embedded systems software testing shares much in common with application software testing. However, some important differences exist between application testing and embedded systems testing. Embedded developers often have access to hardware-based test tools that are generally not used in application development.

Also, embedded systems often have unique characteristics that should be reflected in the test plan. These differences tend to give embedded systems testing its own distinctive flavor. This paper covers the basics of testing and test case development and points out details unique to embedded systems work along the way.

ADVANTAGES OF EMBEDDED SYSTEM TESTING

Before we begin designing tests, it's important to have a clear understanding of why we are testing. This understanding influences which tests us stress and (more importantly) how early we begin testing. In general, we test for four reasons:

- To find bugs in software (testing is the only way to do this)
- To reduce risk to both users and the company

- To reduce development and maintenance costs
- To improve performance

To Find the Bugs

One of the earliest important results from theoretical computer science is a proof (known as the Halting Theorem) that it's impossible to prove that an arbitrary program is correct.

To Reduce Costs

The classic argument for testing comes from *Quality Wars* by Jeremy Main. In 1990, HP sampled the cost of errors in software development during the year. The answer, \$400 million, shocked HP into a completely new effort to eliminate mistakes in writing software.

The \$400M waste, half of it spent in the labs on rework and half in the field to fix the mistakes that escaped from the labs, amounted to one-third of the company's total R&D budget and could have increased earnings by almost 67%.[5]

The earlier a bug is found, the less expensive it is to fix. The cost of finding errors and bugs in a released product is significantly higher than during unit testing, for example (Figure 1 below).

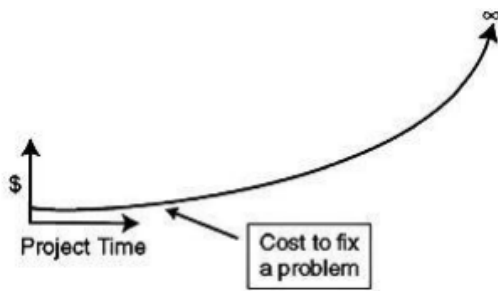


Figure 1: The Cost to Fix a Problem. Simplified graph showing the cost to fix a problem as a function of the time in the product life cycle when the defect is found. The costs associated with finding and fixing the Y2K problem in embedded systems is a close approximation to an infinite cost model.

To Improve Performance

Testing maximizes the performance of the system. Finding and eliminating dead code and inefficient code can help ensure that the software uses the full potential of the hardware and thus avoids the dreaded “hardware re-spin.”

ASPECTS OF TESTING COMPLEX EMBEDDED SYSTEMS

Depending on the type of the granule, and according to common usage in the industry (we will discuss this usage later in this section), it is common to consider testing from six different aspects to evaluate whether the application's behavior, performance, and robustness match expected criteria. These aspects are:

- Software unit testing Software integration testing
- Software validation testing
- System unit testing
- System integration testing
- System validation testing

What follows is a discussion of each of these aspects in relation to testing complex embedded systems.

1. Software Unit Testing

The GuT is either an isolated C function or a C++ class. Depending on the purpose of the GuT, the test case consists of either: Data-intensive testing: applying a large range of data variation for function parameter values, or

Scenario-based testing: exercising different C++ method invocation sequences to perform all possible use cases as found in the requirements.

Points of Observation are returned value parameters, object property assessments, and source code coverage. White-box testing is used for testing units, meaning that the tester must be familiar with the content of the GuT. Unit testing is thus the responsibility of the developer.

Since it is not easy to track down trivial errors in a complex embedded system, every effort should be made to locate and remove them at the unit-test level.

2. Software Integration Testing

The GuT is now a set of functions or a cluster of classes. The essence of integration testing is the validation of the interface. The same type of Points of Control applies as for unit testing (dataintensive main function call or methodinvocation sequences), while Points of Observation focus on interactions between lower-level granules using information flow diagrams.

As soon as the GuT starts to be meaningful, that is when an end-to-end test scenario can be applied to the GuT. First, performance tests can be run that should provide a good indication about the validity of the architecture. As for functional testing, the earlier the better. Each forthcoming step will then include performance testing. White-box testing is also the method used during that step. Software integration testing is the responsibility of the developer.

3. Software Validation Testing

The GuT is all the user code inside a component. This includes all the code

developed by the team, but only the code itself. This aspect of testing can be considered one of the activities that occur toward the end of each software integration or build cycle. Partial use-case instances -- also called partial scenarios -- begin to drive the test implementation. The test implementation is less aware of and influenced by the implementation details of the GuT. Points of Observation include resource usage evaluation since the GuT is a significant part of the overall system. Again (and finally), we consider this step as white-box testing. Software validation testing is still the responsibility of the developer.

4. System Unit Testing

The GuT is now a full system component - - that is, the user code as tested during software validation testing plus all RTOSand platform-related pieces that allow the code to run at minimum on a simulated target: tasking mechanisms, communications, interrupts, and so on. The Point of Control protocol is no longer a call to a function or a method invocation, but rather a message sent/received using the RTOS message queues, for example.

The similarities in their message-passing paradigms implies that the distinction between test drivers and stubs can, from

the perspective of the GuT, be considered irrelevant at this stage. We will call them Virtual Testers because each one can replace and act as another system component vis-à-vis the GuT. "Simulator" or "tester" are synonyms for "virtual tester." Virtual tester technology should be versatile enough to adapt to a large number of RTOS and networking protocols. From now on, test scripts usually: bring the GuT into the desired initial state; then generate ordered sequences of samples of messages; and validate messages received by comparing (1) message content against expected messages and (2) date of reception against timing constraints. The test script is distributed and deployed over the various virtual testers. System resources are monitored to assess the system's ability to sustain embedded system execution. For this aspect, greybox testing is the preferred testing method. In most cases, only a knowledge of the interface (the API) to the GuT is required to implement and execute appropriate tests. Depending on the organization, system unit testing is either the responsibility of the developer or of a dedicated system integration team.

5. System Integration Testing

The GuT starts from a set of components within a single node and eventually

encompasses all system nodes up to a set of distributed nodes. The PCOs are a mix of RTOS- and network-related communication protocols, such as RTOS events and network messages. In addition to a component, a Virtual Tester can also play the role of a node. As for software integration, the focus is on validating the various interfaces. Grey-box testing is the preferred testing method. System integration testing is typically the responsibility of the system integration team.

6. System Validation Testing

The GuT is now a complete implementation subsystem or the complete embedded system. The objectives of this final aspect are several:

Meet external-actor functional requirements. Note that an external-actor might either be a device in a telecom network (say if our embedded system is an Internet Router), or a person (if the system is a consumer device), or both (an Internet Router that can be administered by an end user).

Perform final non-functional testing such as load and robustness testing. Virtual testers can be duplicated to simulate load, and be programmed to generate failures in

the system. Ensure interoperability with other connected equipment. Check conformance to applicable interconnection standards. Going into details for these objectives is not in the scope of this article. Black-box testing is the preferred method: The tester typically concentrates on both frequently used and potentially risky or dangerous use-case instances.

CONCLUSION

Like other forms of software testing, embedded software testing can be performed either manually or by using automated software testing tools. Unlike other software, embedded software is often written for machines that are not primarily used as computers. Modern industrial controls, airplanes, cars, railway systems and medical devices all incorporate "built-in" software whose purpose is to govern the interaction between the machine or device and its physical context.

In most cases, embedded software is developed especially for the hardware in which it is incorporated. Since many forms of embedded software - such as automotive software, aerospace software and railway software - are both highly complex and mission-critical, the process of embedded software testing is a crucial component of development.

Because of these factors, many embedded software developers choose to utilize automated software testing tools to verify that their projects are free of defects. Automated software testing enables a high number of rigorous tests to be conducted systematically, beginning early in the development process.

Embedded software testing is viewed as particularly important in settings where small software defects can have a large impact on the reliability of systems upon which people's lives and livelihoods depend.

REFERENCES

- I. "Software Test Attacks to Break Mobile and Embedded Devices", by Jon Duncan Hagar. 2014. Pp. 236-241.
- II. "Testing Embedded Software" by Bart Broekman and Edwin Notenboom. 2009. Pp. 330-312.
- III. "Embedded Software and Systems: First International Conference, ICESS 2004", by Zhaohui Wu. 2004.pp 524-533.
- IV. "Software Testing: Effective Methods, Tools & Technique", By Rajani. 2003. Pp. 23-34.

- V. “Improving Software Testing: Technical and Organizational Developments”, By Tim A. Majchrzak. 2012. Pp. 134-145
- VI. “Improvements for an Embedded Software Testing Method”, by Kungliga Tekniska högskolan, 2010. Skolan för industriell teknik och management. Pp. 19-23