Continuous Integration Platform for Arduino Embedded Software

Wade Penson, Eric Huang, Dana Klamut, Eliana Wardle, Graeme Douglas, Scott Fazackerley, Ramon Lawrence
Department of Computer Science
University of British Columbia Okanagan
Kelowna, BC, Canada E-mail: ramon.lawrence@ubc.ca

***Abstract*—Software development for embedded systems is challenging due to hardware resource limitations and complexities in testing and verification. Although there are numerous approaches and tools, the integration and deployment of these tools for a particular software development project requires effort. The contribution of this work is a platform for continuous integration that adapts common open source testing software for enterprise development for use with Arduinos. A description of a software development workflow that utilizes the platform is provided as well as its specific application to developing a database library for embedded systems.**

***Index Terms*—continuous integration, Arduino, testing, embedded software, verification, platform, framework**

I. Introduction

The quality of software produced is dependent on the software development process and techniques for testing and verification. Continuous integration (CI) automates building and testing and is used extensively for computer and mobile development projects. Embedded developers often struggle with implementing testing into their projects and combining the various tools into an integrated testing process. The result is that testing may be performed on an ad hoc basis.

There are numerous unit testing and CI systems for application development (e.g. Jenkins [1]), but these frameworks are not easily applied for Arduino development. The main challenges are the integration of the system with the devices, communicating test results to the testing platform, and handling the potential error states of the devices.

The contribution of this work is a configurable and expandable CI platform for Arduino development that leverages industrial quality, open source tools designed for application development. The Jenkins CI platform is deployed with a suite of plugins for C/C++ development and a testing framework specifically targeted for Arduinos. The system requires no specialized hardware and has crash recovery that is able to detect and correct many cases that would otherwise require human intervention. The paper describes how to utilize CI in a development project and provides a discussion on the results of applying continuous integration in the development of an existing embedded software library.

II. Background

The volume and diversity of embedded devices is rapidly increasing especially as the Internet of Things [2], [3] expands. Since embedded devices have specific tasks and purposes, development of embedded software is often highly customized to particular use cases and environments. A common challenge is the testing and verification of embedded software [4]. Despite numerous techniques and tools, some projects still use ad hoc testing [5] due to the complexity of configuration and automation. A unique challenge of embedded systems is that their behavior often depends on their deployment environment and testing must often reproduce environmental factors. Further, there may be real-time timing constraints and challenges handling robustness and failures.

Continuous integration is the automation of compilation, build, test, and deployment. Continuous integration [6] is used by the top open source projects, improves the frequency of releases, and increases the quality of the development process. CI encourages developers to commit, integrate, and test more frequently using a shared repository and automated testing. The goal is to speed up time to market and response time to customer requests. Microsoft has created its own continuous integration system [7] used for Bing, Office, and other products, and the use of CI is projected to expand further as more developers understand its benefits and how to deploy it effectively [6]. A popular open source CI system is Jenkins [1] which is also used for continuous delivery [8] that expands on continuous integration to also automate software deployment and release. Jenkins is a CI platform written in Java that is easily extensible using plugins and allows chaining jobs together to produce an overall process pipeline.

Continuous integration and automated testing is especially valuable in detecting regression errors and ensuring the build is always passing. Automated testing allows tests to be performed more frequently and speeds up testing. Adopting continuous integration is not always easy, especially for established teams in the embedded space, due to both social and technical reasons. Experiences at Ericsson [9] demonstrate some successes but also challenges with distributed development groups and getting the technologies to be stable and reliable. Issues cited included poor performance of tests, unstable tests, slow tests, insufficient testing environments and tools, and social issues including lack of time, training, and experience.

There has been limited published work on applying continuous integration to embedded development. The closest related work is Xest [10] that developed a regression testing framework for embedded software that executes tests in parallel on

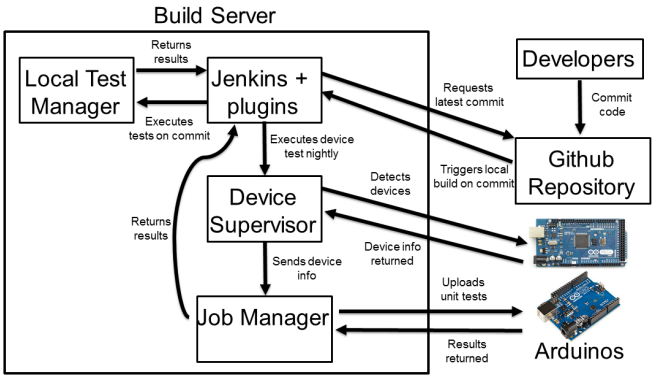

Fig. 1. Testing architecture

embedded hardware. Xest was used for embedded development and for verifying student assignments in an educational environment. A powerful feature is that it organized embedded devices into a remotely accessible pool. Students using Xest in a course see the benefit of test-driven development. The build system can be performed via command-line or for nightly builds. This work expands on the capabilities of the Xest and includes many more tools such as static analysis, memory detection, etc. that is made possible by using an expandable continuous integration platform.

III. Continuous Integration Platform

The continuous integration platform designed for Arduino development is shown in Figure 1. The build server runs Jenkins [1] and a variety of plugins. A collection of Arduino devices are connected to the build server via USB. Jenkins is chosen as it is a powerful open source product that is easily customized using plugins.

On code commit to the Git repository, a build and testing process is initiated by the Jenkins Git plugin [11]. This build process executes the tests on both the build server and the connected devices and results are reported.

To run the tests on the device, a testing framework called Planck Unit was developed. Planck Unit is designed to be lightweight so it can run on embedded devices such as Arduinos with as little as 2 KB of memory. In Listing 1, an example test suite is constructed that executes a single unit test. The unit test executes a function and compares its value to the expected value. Planck unit is a generic unit testing library supporting any type of test. At this time, it does not have special support for tests that require environmental stimulus as input to the Arduino, but this functionality can be built into the tests themselves and is future work for expanding the library.

The job allocator determines the number and types of Arduinos connected to the build server and then uses this information to distribute testing jobs between them. The system determines the device port (which USB port it is plugged into), device type, and processor type. To do this, Python scripts were created that use a list of known Vendor ID (VID) and Product ID (PID) pairs to match against device information. For devices that cannot be automatically detected, the system compiles and uploads a executable to the device to probe the device type as well as detect specific device properties (amount of Flash memory, presence of SD card shield, etc.) The output of the executable is sent over the serial port to the build server to collect and analyze the information. This detection process builds a saved device profile that the allocator uses when allocating tests to devices. The platform supports all common Arduino devices and can be extended to other embedded devices.

Listing 1. Planck Unit Test Example

```
// Create a single unit test
void sample_test(planck_unit_test_t *tc) {
   int result = executeFunc();
   PLANCK_ASSERT_TRUE(tc, result == 10);
}

// Build and execute suite of tests
void execute_suite_of_tests() {
   planck_unit_suite_t *suite
        = planck_unit_new_suite();

   PLANCK_UNIT_ADD_TO_SUITE(suite, sample_test);
   planck_unit_run_suite(suite);
   planck_unit_destroy_suite(suite);
}
```

Each test is built for every different device type connected. For example, if an Arduino Uno and Arduino Mega are connected, then two executables will be produced (one for each). During this process, the code size is checked to verify if the test can be run on the specific device given its hardware capabilities. This allows the job allocator to assign tests to only the devices that can run them. A job manager handles the process of uploading the executable, running it, and collecting the results.

The process of compiling and uploading the executables is managed by CMake. A CMake toolchain for the Arduino utilizes the avr-gcc compiler with libraries, bootloader images, board definitions, programmer definitions, and other various files sourced from the Arduino Software (Arduino IDE). The toolchain requires the board type, processor type, serial port, programmer type, and other various options as parameters in order to facilitate the compilation, linking, and upload processes. Typically, these parameters would have to be manually set for the testing configuration on hand. Our device supervisor automatically resolves this issue by detecting and building a configuration profile for each connected device, which is passed to the job manager to allow it to correctly upload a test executable to the target device. The toolchain then compiles and links the Arduino core, any required Arduino libraries, and the user's test code in the form of Arduino sketches and/or C/C++ files. Lastly, the toolchain then uploads the generated test executable using AVRDude.

Jobs are run in parallel across all devices. Planck Unit collects the output of the jobs and detects common error conditions such as a test that hangs, garbage output, repeated output, and various other problems. Importantly, a failed test or device will not block the entire testing process and failed

TABLE I
PLATFORM FEATURES

Feature	Plugin
C unit testing framework	*Planck Unit*
embedded device testing	*Job allocator/manager*
static code analysis	Cppcheck [13]
code coverage	Cobertura [14]
memory leaks	Dr. Memory [15]
display warnings/errors	Warnings plugin [16]
abort hanging build	Build timeout [17]
auto-commit detection	Git plugin [11]

tests have output reported in the system. A test that hangs is handled using a timeout for each unit test. If a test does not complete before the timeout, the test is failed and related unit tests in the test suite are aborted without affecting other test suites. The job manager monitors the serial port for detecting failures and may perform a soft reset on the device. The test output collected from the serial port is in a robust XML format, which allows the testing platform to recognize if a test has ended early or gone into an infinite loop. The XML output is then converted into XUnit format so that it can be read into Jenkins by the XUnit plugin [12].

The advantage of using a continuous integration platform is that a complete build pipeline can incorporate testing and analysis tools beyond regression tests. The continuous integration platform deployed had numerous plugins (Table I) to provide key features such as static code analysis, code coverage, memory leak testing, and handling of embedded device failures that would break testing. These plugins were used from other sources. Our unique contributions are the testing framework (Planck Unit), the automated job allocator/-manager, and extensions to the Arduino CMake Toolchain to support compilation to the latest devices including the Arduino Mega. The integration of all these facilities into a CI system is also unique and everything is open source which is important to the Arduino builder community.

IV. EMBEDDED DEVELOPMENT WITH CONTINUOUS INTEGRATION

An embedded development project that uses continuous integration follows these steps:

- Check out code from Git repository to local development machine.
- Create a new feature branch for the code changes.
- Perform the code changes.
- Commit the code changes to the source Git repository.
- At commit, the CI system runs all testing and analysis on the build server and provides information on errors, testing failures, memory usage and leaks, and warnings. To make commits fast, testing is not performed on the device unless requested. This is called a “rapid build”. A “full build” is where all testing is performed both on the test server and on the devices.
- If the tests fail, then the developer must go back and improve and refactor the code. If all tests pass both on the devices and build server, then a code review occurs where at least one other developer reviews the changes and must sign off. Sign off is done electronically via the Code Review feature on GitHub.
- Once code review is successfully completed, the developer merges the feature branch into the development branch.

There are multiple ways to perform software development that incorporate continuous integration [6]. One common approach is to not use branching in the repository and always commit to master. The development workflow described was adapted to handle a specific challenge of embedded devices where testing can be very slow on device and causes wear on the device. The initial commit on a feature branch performs testing on the build server first to verify correctness rapidly. This encourages developers to commit and test frequently and get their code reviewed quicker. After the server tests pass, then the device tests are run. Code review occurs after all tests pass. This approach is designed to counter some of the issues against continuous integration outlined in the Ericsson study [9] especially regarding unstable, slow, and unpredictable builds that affected developer productivity.

V. PLATFORM DEPLOYMENT DISCUSSION

The continuous integration platform and development workflow was applied to the IonDB project [18] which is building a database and key-value store for embedded devices. The project involves a team of 6 developers. Development of IonDB began in May 2014 with major development efforts during the summer months (May 2014-August 2014, May 2015-August 2015, May 2016-August 2016) where students are recruited to work on the project for research. The CI platform was deployed in this project in June 2016. Compared to the first two years of development the number of developer commits increased by a factor of 3 (see Figure 2). Further, a number of fatal bugs were detected after the introduction of the regression testing and memory leak testing that were not previously detected. The developer productivity and satisfaction was significantly improved with developers feeling confident to make changes and use the testing framework to verify that regression errors were not introduced.

VI. CONCLUSIONS AND FUTURE WORK

The contribution of this work was the deployment and testing of a continuous integration platform for Arduinos. Continuous integration offers numerous benefits, but prior work has shown that it can often be challenging to deploy in embedded development teams. This work described how the Jenkins platform can be adapted and expanded to work with embedded devices. The challenges of limited device resources and slow testing times can be mitigated by a two-stage build process with testing on device happening after tests pass on the build server. The software developed to interface Jenkins with Arduino devices handles key challenges of device

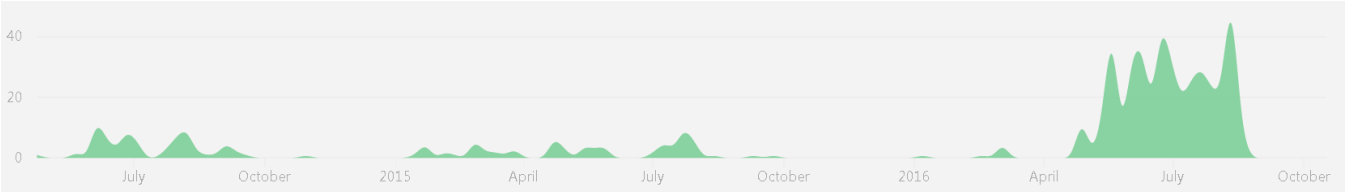

Fig. 2. Number of repository commits per week before and after CI platform

failure and hangs, minimal memory available for testing on device, and integration into the overall Jenkins system for reporting and automation. Future work is to continue to extend the platform including creating a watchdog to detect locked devices requiring a hard reset after failure and adding support for stimulus testing of device inputs.

VII. Acknowledgment

The authors would like to thank NSERC for supporting this research.

References

[1] Jenkins. [Online]. Available: https://jenkins.io/

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[3] Y. Qin, Q. Z. Sheng, N. J. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos, "When things matter: A survey on data-centric internet of things," *Journal of Network and Computer Applications*, vol. 64, pp. 137 – 153, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804516000606

[4] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "On Testing Embedded Software," ser. Advances in Computers, A. Memon, Ed. Elsevier, 2016, vol. 101, pp. 121 – 153. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245815000662

[5] S. Shah, D. Sundmark, B. Lindstrm, and S. F. Andler, "Robustness Testing of Embedded Software Systems: An Industrial Interview Study," *IEEE Access*, vol. 4, pp. 1859–1871, 2016.

[6] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, Costs, and Benefits of Continuous Integration in Open-source Projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016, pp. 426–437. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970358

[7] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's Distributed and Caching Build Service," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889222

[8] V. Armenise, "Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery," in *Proceedings of the Third International Workshop on Release Engineering*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 24–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820690.2820701

[9] E. Laukkanen, M. Paasivaara, and T. Arvonen, "Stakeholder Perceptions of the Adoption of Continuous Integration – A Case Study," in *Agile Conference (AGILE), 2015*, 2015, pp. 11–20.

[10] M. H. Netkow and D. Brylow, "Xest: An Automated Framework for Regression Testing of Embedded Software," in *Proceedings of the 2010 Workshop on Embedded Systems Education*. New York, NY, USA: ACM, 2010, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1930277.1930284

[11] Git plugin. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin

[12] Xunit plugin. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin

[13] Cppcheck plugin. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Cppcheck+Plugin

[14] Cobertura plugin. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin

[15] Dr. memory. [Online]. Available: http://www.drmemory.org/

[16] Warnings plugin. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Warnings+Plugin

[17] Build-timeout plugin. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Build-timeout+Plugin

[18] S. Fazackerley, E. Huang, G. Douglas, R. Kudlac, and R. Lawrence, "Key-value store implementations for Arduino microcontrollers," in *IEEE 28th Canadian Conference on Electrical and Computer Engineering*, 2015, pp. 158–164. [Online]. Available: http://dx.doi.org/10.1109/CCECE.2015.7129178