Executable Test Definition for a State Machine Driven Embedded Test Controller Module

Benjamin Steinwender*†, Michael Glavanovics†, Wilfried Elmenreich*
*Smart Grids Group/Lakeside Labs, Alpen-Adria-Universität Klagenfurt, Austria
†KAI Kompetenzzentrum Automobil- und Industrie-Elektronik GmbH, Villach, Austria
E-mail: benjamin.steinwender@k-ai.at

***Abstract*—Microcontroller modules are introduced to a life test system to apply stimuli and stress patterns to the unit under test. The firmware must evaluate the status or operational function of the tested device for a multitude of different applications. A finite-state machine design for the microcontroller is presented that is configurable via a communication bus. The controller acts according to the given test plan description. Various sources can trigger *events* to cause a transition in the state machine. Custom Lua script code is executed in each state in order to control and communicate with the test hardware.**

Keywords: microcontroller, Lua, embedded scripting, finite-state machine, life testing

I. Introduction

Life testing of devices and applications for reliability assessment purposes requires a large number of units to be tested in a harsh environment, i.e. temperature ranges up to $125\,°C$ or repetitive short circuit testing [1]. The devices and their application are usually put into a climate chamber, while the stress test hardware is left outside because it will not sustain the high temperatures. Traditional systems come with a control system outside the climate chamber, which requires a large number of analog signal wires and limits the flexibility of the test setup. In [2] we proposed a paradigm shift towards a system of distributed control and test nodes which are located close to the unit under test (UUT) to enable:

- closed loop control,
- in-situ data acquisition and analysis,
- real-time failure diagnosis, and
- high speed interfaces to the UUT.

A new generation of life test systems for power semiconductor devices and applications is currently developed at the KAI center of competence. It provides an array of controller and application modules, linked by a communication bus to a central host computer [2]. The microcontroller modules are used to apply stress patterns and monitor the devices on the application modules. Several of these microcontroller hardware targets are used to obtain statistical data about multiple tests.

To provide a further advantage, i.e., higher flexibility towards different test setups, an appropriate configuration approach for the test software is required. The test procedures can be defined on the host computer at run-time and transferred to the microcontroller to be executed there.

During product development and subsequent reliability life testing, the test requirements change as soon as problems are identified or superseded by other problems. Thus, not only the execution of tests takes a significant amount of time, but the test setup also takes considerable effort. Typically, different instruments like power supplies, electronic loads, as well as pattern and waveform generators are used simultaneously. During tests the engineer needs to change the wiring of the test setup as well as to modify the test software. By reducing this setup time, more tests can be performed, providing more data about the long-term behavior of products and minimizing their time to market.

To reduce complexity, we propose a modular software architecture which can be adjusted to the respective hardware and required test. The modular approach is beneficial, because it provides several advantages over the previous generations of monolithic test systems [3], [1]:

- Separate test development from hardware and software design
- Simplify development, debugging and replacement of individual hardware modules
- Enable a smaller development team to tackle one problem at a time while using proven modules for their main code.

The purpose of this paper is to describe and evaluate a software framework that is able to handle a variety of different modular hardware setup scenarios. The proposed framework builds upon the concept of finite state machines and Lua scripts for executing test routines. The use of the script language Lua is expected to reduce the cognitive complexity of a test routine. The finite state machine (FSM) allows for a modular composition and extension of existing test sequences.

On the hardware side, there are currently only few controllers that are powerful enough to drive such tests and able to sustain the required temperatures. The described work is based on the XMC4500 microcontroller [4] featuring an ARM Cortex-M4F core running at $120\,\mathrm{MHz}$ with $1\,\mathrm{MiB}$ program flash and $160\,\mathrm{KiB}$ RAM. The controller was selected due to the required automotive temperature operating range of $-40\,°C$ to $125\,°C$ and outstanding analog performance.

In the next sections, we describe related concepts from literature and previous work followed by a description of our test definition model. The overall system is sketched in Section IV. Finally, first experiences with the approach are discussed in Section V and we give an outlook to future work.

II. Related & Previous Work

In order to find and fix bugs faster and to reduce the effort of setting up and redoing a test suite after a code modification, automated unit testing has been in use in the context of software development for the past two decades [5]. The idea has been extended to firmware [6], digital and analog hardware. Hardware tests - often built upon Built-in self-test (BIST), i.e., test generation, test application, and response verification - are usually accomplished by built-in hardware [7]. However, in case of a power stress test, the test system must be physically separated from the UUT in order to avoid failures of the test system caused by the extreme conditions during the test, such as the dissipated heat power, high current or voltage.

In such a modular power stress test system, a variety of different applications should be addressed without the need to change the basic firmware of a given microcontroller target. Therefore, it is not sufficient to only change the parameters of the test program (e.g. timing values, output voltage levels), but it is usually required to also change the behavior and sequence of certain actions. The requesters of these application tests are typically company internal hardware designers and product engineers. They prefer directly configuring a test sequence rather than low-level programming of microcontrollers. Thus, a dedicated firmware concept with a simple on-line configuration approach is required.

Over the last years, networked boot-loaders have emerged that allow transferring executable code during power up of the controller via serial data buses such as UART, CAN or Ethernet [8], [9]. Using these mechanisms, updating the controller firmware can be simplified substantially, since it is not required to plug the flash tool into each controller individually. However, this does not solve the problem that for each different task a new firmware image needs to be created.

Improved concepts implement a base firmware that supports the execution of script commands. Several projects have already investigated approaches to provide embedded virtual machines for Lua (*eLua* [10]) or Python (*p14p* "Python-on-a-chip" [11], the *Owl Embedded Python System* [12] and MicroPython [13]) which allow the execution of user provided code on a microcontroller.

Barr [12] describes a sophisticated software framework based on the Python virtual machine that removes the hitherto required compile-flash cycle of microcontroller programming. This approach features an interactive Python interpreter on a microcontroller. The original Python virtual machine is modified and only a subset of Python commands is made available. The Python program however needs to be compiled into bytecode on a desktop machine and is transferred to the microcontroller via a serial data bus for execution. Additionally, the bytecode can also be stored on the controller.

In contrast, the MicroPython project features a reimplementation of the Python language on a microcontroller. It contains all language features as well as an on-line interpreter. Furthermore, a read-eval-print loop can be accessed via its serial connection and programs can be stored in the embedded device's flash memory.

In the same line of thought, the eLua-project runs a full Lua interpreter on the microcontroller. Thus, the script can be compiled on-line and all language features are available. eLua also provides an interactive command access for simple debugging. Additionally, Lua supports the loading of pre-compiled bytecode in addition to raw source files.

Both script projects provide a powerful C-API for the incorporation of custom C-functions, enabling simple access to the controller's hardware periphery. In addition, they provide means for a standalone program execution. However, they fall short in terms of distributing the program code to multiple controllers at the same time. Distribution is essential if numerous controllers are situated in the same test system.

III. Test definition model

Before deploying test descriptions onto the controller target, a generic test model has to be found. By investigating typical power electronics test scenarios, it became evident that the test program on the controller has to take care of:

- Applying configuration, digital and analog stress test patterns.
- Reading system responses and measurement values.
- Receiving and executing messages from the governing test handler on the host (i.e. "host messages", especially *START* and *STOP* messages) in order to proceed to the test sequence where dependencies on external hardware are defined.
- Handling trigger signals generated by the application or microcontroller periphery (i.e. interrupt events)
- Evaluating internal states (i.e. measured analog signals, return values from digital interfaces)
- Running background routines for sending and receiving messages and transferring measured data to the host.

A. State machine

These bullets can be summarized and generalized as "actions" and "events". An "action" is the sequence of instructions which is to be carried out by the controller in order to perform a specific task and evaluate the result. An "event" is triggered by internal modules in the controller or externally through the test application and requires the controller to perform an "action". Given these two basic terms, the non real-time behavior of a test procedure can be modeled with an FSM [14].

The reaction to an "event" can be described by a transition in the state diagram. While the controller is in a certain FSM state, it will continuously execute the associated "action". This implicit behavior can be modeled by the unconditional *@else*-event, forming a transition from each state to itself in a loop. In order to execute a state only once and immediately move to the next state, the loop needs to be cut. This can be achieved by manually pointing the *@else*-event to a different state.

The information whether an event has occurred is stored in a fixed-size, pre-allocated table indexed by the event number. Thus, the interrupt service routines (ISRs) and the main loop

may modify the table, since no dynamic memory allocation - as found in a queue-based system - is required. Furthermore, locking of this global table is not required, as event occurrences are set (by hardware ISR or software) and cleared (by the FSM handler described below) by independent instances.

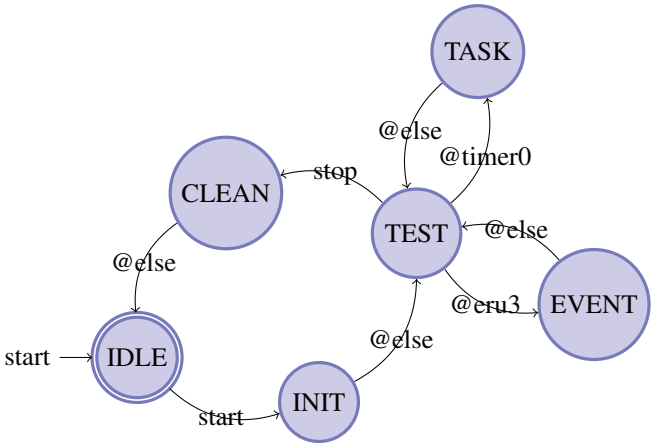

Figure 1. Simple controller state machine

A simple example of a state machine structure for such a test sequence is given in Figure 1. Once it has finished booting, the controller waits for the reception of a new state machine description via the serial communication interface. It parses the contents and builds up a directed graph. Finally, the controller starts executing the state machine in the IDLE state. While in the IDLE state, no user definable actions can be carried out and only background routines are being processed. These background routines are required to transfer the configuration to the controller and convert it into the state machine. The IDLE-state is also the final state and the controller can be stopped or the FSM structure may be changed.

Upon reception of the *start* event through the host, the FSM will propagate into the INIT state, perform the one-time setup procedures given by the test description and automatically change into the TEST state indicated via the *@else* event. In the example TEST state, the automaton listens for the hardware-events *@timer0* (periodic time-triggered work load) and *@eru3* (external events - e.g. button pressed) and the software-event *stop* (for stopping the test).

Such a state diagram can easily be converted into C-code, compiled and transferred to the controller for execution. However, alteration of the procedure (e.g. adding another event and its corresponding action to the TEST state or synchronization with the host or other controllers) requires a skilled programmer who is capable of implementing these changes as well as perform recompilation and finally flashing the firmware. This solution is thus not favorable for practical use in the test laboratory.

An improved version allows reconfiguring the states and transitions in order to execute arbitrary sequences. Therefore, the controller may receive a state transition table as given in Table I. The table is constructed by enumerating all transitions between states, which allows full reconstruction of the state

#	current state	event	next state
1	IDLE	*start*	INIT
2	INIT	*@else*	TEST
3	TEST	*stop*	CLEANUP
4	CLEANUP	*@else*	IDLE
5	TEST	*@timer0*	TASK
6	TASK	*@else*	TEST
7	TEST	*@eru3*	EVENT
8	EVENT	*@else*	TEST

Table I
STATE TRANSITION TABLE

machine diagram. To simplify the diagram and to reduce the table size, the implicit loop transitions via the *@else* event are not stored.

The transition table is constructed on the host computer where the test plan is created and converted into a serial form in order to be transferred via the communication interface. Each state and each event are converted into numerical IDs and then collected as a triple, which represents a transition. The controller receives the list of triples and reconstructs the table in its internal RAM.

According to this procedure, states can be traversed in a simple way as presented in Algorithm 1. If the event matches none of the events listed in the transition table, the current state is kept (Line 1). While iterating through the table, only transitions that originate from the current state are considered. This behavior is improved by storing the outgoing transitions from each state in a tree-like data structure which is indexed by the state. If the current state contains a transition using the *@else* event, it is saved and the search algorithm continues querying the table in order to match a possible triggered event (Line 4). If the current state contains a transition event that has actually been triggered, the result state is immediately assigned and the search is complete.

B. Hardware interaction

The main reason to use microcontrollers is justified by the possibility to interact with the integrated on-chip hardware modules like analog converters, timers and digital interfaces. Since the interaction should be included in the previously mentioned state machine, a set of commands is desired which can be executed during each state.

Of course, one could build a simple parser for reading commands and parameters. However, since low-level programming of microcontrollers is difficult, this task is error-prone and a very large amount of commands must be implemented. Furthermore, such a design is rather inflexible and poses many risks to introduce bugs. The favored solution is to use an interpreted language which also supports features such as loops, conditions and even functions (see Section II).

Lua has been chosen for the implementation, because it is small, utilizes a minimum of RAM while still being powerful. The real benefit comes with the highly sophisticated C-API that allows an implementation of custom modules and hardware access routines. The interface between the Lua virtual machine and C-functions is defined with a modifiable virtual stack. Thus,

```
Data: transitionTable, currentState
Result: nextState
1  nextState ← currentState;
2  foreach transition in transitionTable do
3      if transition.origin == currentState then
4          if transition.event == ELSE then
               /* save the @else state, it
                  will be used if no other
                  event occurred                */
5              nextState ← transition.nextState;
6              continue;
7          end
8          if eventOccurred(transition.event) then
               /* event occurred, update
                  nextState and exit            */
9              nextState ← transition.nextState;
10             clearEvent(transition.event);
11             break;
12         end
13     end
14 end
```

Algorithm 1: State machine handler

the C-code may receive parameters from Lua function calls and can provide results back to the Lua virtual machine.

Listing 1. Accessing a GPIO module instance

```
p12 = gpio.init("p12") -- get GPIO object
p12:setOutput()
p12:write(1)
```

To simplify the user experience, the modules are provided in an object oriented way as given by Listing 1. The user script first needs to obtain a general purpose input output (GPIO) object by calling the single public function `init` from the `gpio`-table. The invoked C-function then takes care of setting up the periphery access and returns a Lua object `p12`. A Lua metatable is used to protect the Lua object instance from incorrect usage [15]. Consecutively, the user script may configure the pin as output and write a value to it. According to this simple example, we have implemented modules for most of the periphery units available on-chip, i.e. analog input & output (IO), digital IO, delta-sigma demodulator, event request unit (external interrupts, ERU), PWM, SPI and timer functions.

As executing Lua scripts may cause the interpreter to fail for erroneous scripts, an additional event *@error* is introduced. When running the current FSM's script fails, this special event is triggered and allows the test program to transit into a safe state.

C. Real-time & background routines

In addition, real-time functions need to be carried out by the controller such as closed loop PI control, PWM generation and analog waveform acquisition. Modern microcontrollers already provide powerful hardware modules that can do most of the required work independently in parallel to the software code. Therefore, the implemented Lua modules described above are used to configure these hardware units through the microcontroller registers.

Some hardware periphery modules will return data, e.g. the ADC or serial interfaces. In order to avoid race conditions and deadlocks when writing data to the memory, the following two options are provided:

Non-blocking: Within the FSM INIT-state, the user code must allocate a memory region for the specific module by stating how many samples need to be stored. The result data is then written by the direct memory access (DMA) controllers. The data can be accessed later, upon completion of the DMA transfer. Additionally, the host can also request data from an allocated memory region via the communication interface.

Blocking: The user code may read the requested hardware resource, but the command will not return a result until the value is available. Due to this blocking nature, further script commands as well as the FSM handler are delayed.

For the PI-controller, input and output signals as well as the control loop parameters are configured via Lua script commands, so that the invoked ISR function could be written in a purely functional style. This gives a large performance benefit compared to functions executed in Lua.

IV. System Integration

The microcontrollers are placed close to the units under test. Therefore, they can easily apply patterns and monitor the application. However, a typical power stress test also involves external instruments, such as power supplies and active loads as displayed in Figure 2. These devices are usually shared between several test instances. Furthermore, the microcontroller may not be powerful enough to perform advanced analysis procedures or store results to a centralized data location. Thus, a hierarchical approach is required to control the overall test routines.

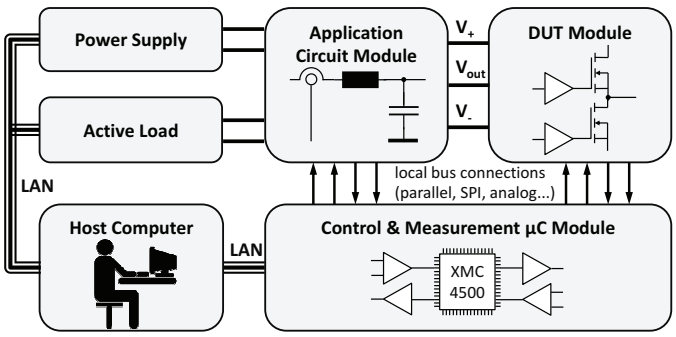

Figure 2. Typical DCDC converter application test

Figure 3 shows the software architecture of the host program. It has been designed according to a general actor model and deals with the communication between the microcontroller targets and the test control. Each physical target has a virtual representative (the Node Actor) in the host software to manage the communication and test execution. Additionally, it can display the status of the controller and the UUT depending on the test definition. For each test definition loaded into the host

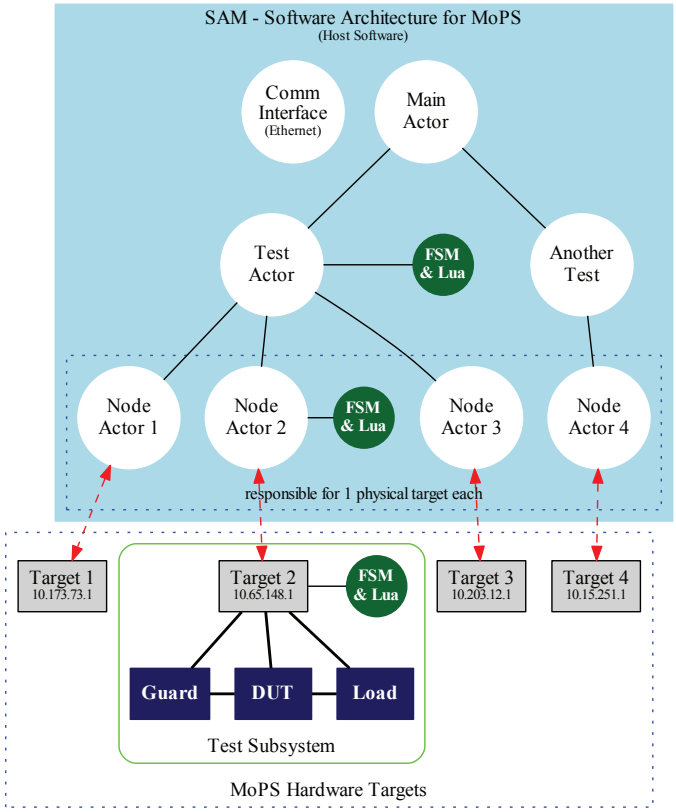

Figure 3. Software architecture

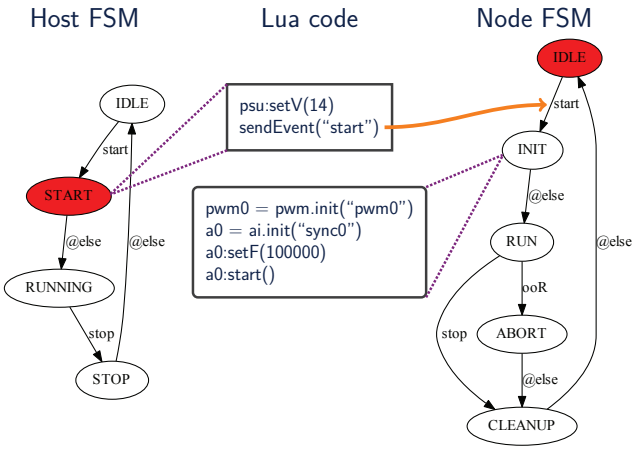

Figure 4. Host to node FSM synchronization

software, a test manager (the Test Actor) is created. It checks if the the requested hardware targets are available and assigns these to the test. Subsequently, the communication interface is used to transfer the configuration to the microcontrollers, where it is used to update the state machine table.

As can be seen in Figure 3, the FSM concept is present on all three hierarchy levels: The hardware targets use the state machine handler and the Lua interpreter to access the on-chip hardware periphery. Each Test and Node Actor can create a dedicated FSM Actor for custom data analysis and storage, for communication with the next lower or higher hierarchy level (sending events and transition notifications) or for the purpose of instrument control (Test Actor only).

By sending events via the communication interface, state machines from different hierarchy levels can be synchronized. Once the test definition is loaded, the host program can send events to the hardware targets in order to start the test execution or to notify that an external instrument has been configured properly. Therefore, two independent state machines are created, one for the host and one for the microcontroller. Figure 4 demonstrates a minimum example of such test definition:

1) In the **Host FSM**, the `start` event is triggered (e.g. by pressing the "Start test" button on the GUI).
2) The FSM proceeds to the START state and executes the associated Lua script code: First, the voltage of the power supply is set. Afterwards, a separate `start` event is sent to the microcontroller target via the communication interface.
3) The **Host FSM** transits to the RUNNING state and waits for test completion (indicated by the `stop` event).
4) The microcontroller receives the `start` event and can now move to the INIT state where it executes the assigned Lua function.

All function calls visible in this picture are custom module implementations on top of the Lua standard libraries. In addition, the Host-FSM supports reading measurement values from the microcontroller. According to the evaluation of the response, the Lua-based user script may invoke a transition in the FSM.

This layered structure allows the test operator to describe the test in a simple, understandable way. A graphical tool, called the "Testplan Builder", is provided to ease the creation of such tests. Using this tool, the state machines can be created graphically, Lua code can be entered into the states and the test sequence can be stepped through.

The tool also provides consistency checking, thus preventing most of the possible erroneous configurations from being created. The execution of a mis-configured test is further prevented by the host software. It verifies the requested Lua application programming interface (API) functions against the available features of the selected microcontroller-target. Therefore, a test program will only be transferred to and executed on the microcontroller, if all used Lua API function calls are considered valid.

V. Discussion & Conclusion

This paper presents a simple but powerful paradigm for embedded test control, allowing an efficient implementation in the microcontroller while enabling online definition of the test sequence. The addition of the Lua interpreter to the microcontroller provides a highly flexible system while reducing the initial setup effort for test engineers. Further, an extensible software architecture is presented. Using Lua scripts and custom FSMs, it is possible to create hierarchical and

concurrent software programs, thus enabling highly flexible test sequences. Incorporating instruments on the host level is very simple, as is the control of digital and analog signals on the microcontroller. With the help of a graphical test-plan creation tool, the product test engineers can describe a test program flow and add Lua scripts for accessing hardware resources.

A power conversion application life test with four distinct converters according to Figure 2 has been set up using a microcontroller-target for each converter. Each microcontroller successfully runs a closed loop PI control to regulate the converter current. Voltage and temperature measurement values are monitored live on the host system. The power supply and active load are shared among the units. Thus, all microcontrollers are connected to the same host and supervised by one single test state machine.

The lab engineer is able to autonomously create the proper ramp-up and turn-off sequences after a short introduction into the system and explanation of the Lua API, appreciating that he does not need to change the microcontroller-firmware in order to modify the test program.

Further work will focus on the scalability of the host system in order to implement high numbers of microcontrollers and test applications running in parallel. The system has not been pushed to its full limits yet, however a significant number of 16 controllers can be handled smoothly so far.

ACKNOWLEDGMENT

The authors would like to thank their colleagues at the KAI center of competence, Infineon Technologies Austria and the Klagenfurt University for fruitful discussions.

This work was jointly funded by the Austrian Research Promotion Agency (FFG, Project No. 846579) and the Carinthian Economic Promotion Fund (KWF, contract KWF-1521/26876/38867).

REFERENCES

[1] M. Glavanovics, R. Sleik, and C. Schreiber, “A compact high current system for short circuit testing of smart power switches according to AEC standard Q100-012,” in *Proc. IEEE Power Electronics Specialists Conf. PESC 2008*, 2008, pp. 1828–1832.

[2] B. Steinwender, S. Einspieler, M. Glavanovics, and W. Elmenreich, “Distributed power semiconductor stress test & measurement architecture,” in *INDIN’13*, 2013.

[3] M. Glavanovics, H. Köck, V. Košel, and T. Smorodin, “Flexible active cycle stress testing of smart power switches,” *Microelectronics Reliability*, vol. 47, no. 9-11, pp. 1790–1794, Sep. 2007. http://linkinghub.elsevier.com/retrieve/pii/S0026271407003216

[4] 32-Bit Industrial Microcontroller based on ARM® Cortex™-M. Infineon Technologies. Accessed 2015-02. http://www.infineon.com/xmc

[5] K. Beck, “Simple Smalltalk Testing,” *The Smalltalk Report*, vol. 4, no. 2, 1994.

[6] P. Fagerburg and A. McInnes, “Develop Robust Firmware Faster With Automated Unit Testing,” Synchroness Inc., White Paper, Mar. 2008.

[7] J. Zakizadeh, S. Das, M. Assaf, E. Petriu, M. Sahinoglu, and W.-B. Jone, “Testing Analog and Mixed-Signal Circuits with Built-In Hardware - A New Approach,” in *Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE*, vol. 1, May 2005, pp. 166–171.

[8] E. Schlunder. (2010) High-Speed Serial Bootloader for PIC16 and PIC18 Devices. AN1310. Accessed 2015-02. Microchip. http://ww1.microchip.com/downloads/en/appnotes/01310a.pdf

[9] J. Garcia-Zubia, I. Angulo, U. Hernandez, M. Castro, E. Sancristobal, P. Orduña, J. Irurzun, and J. de Garibay, “Easily Integrable platform for the deployment of a Remote Laboratory for microcontrollers,” in *Education Engineering (EDUCON), 2010 IEEE*. IEEE, 2010, pp. 327–334.

[10] eLua. Accessed 2015-02. http://www.eluaproject.net

[11] p14p - python-on-a-chip. Accessed 2015-02. https://code.google.com/p/python-on-a-chip

[12] T. W. Barr, “Microcontroller Programming for the Modern World,” Ph.D. dissertation, Rice University, 2014.

[13] D. George. (2015, Feb.) MicroPython. http://micropython.org/

[14] D. Harel, “Statecharts: a visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987. http://linkinghub.elsevier.com/retrieve/pii/0167642387900359

[15] R. Ierusalimschy, *Programming in Lua*, 3rd ed., R. Ierusalimschy, Ed. Ierusalimschy, Roberto, 2013.