Hardware-Based Real-Time Simulation on the Raspberry Pi

Jörg Walter, Maher Fakih, and Kim Grüttner

OFFIS – Institute for Information Technology, Oldenburg, Germany
joerg.walter@offis.de

Abstract. Hardware prototypes are commonly used during embedded control unit design. Existing commercial tools offer an integrated workflow from mathematical models down to hardware simulation. Researchers also build low-cost simulation platforms out of commodity equipment. We present a platform that is an order of magnitude cheaper than existing systems but still easy to integrate into present workflows: Within an existing model-driven design methodology, we perform real-time hardware simulation using the Raspberry Pi single-board computer to simulate an electromechanical system with little development effort.

1 Introduction

The design flow for embedded control units (ECUs) usually includes hardware simulation at some point. In it, a prototype takes part in a simulation of an electromechanical control system. Two major variants exist:

In the first case, developers want to validate their control algorithm against actual environment behavior, especially when analog power electronics are involved. Simulation of such components is difficult [1], so real hardware is needed. The control unit doesn't need to be in its final form as long as the control algorithm can be simulated with sufficient accuracy.

In the second case, developers want to confirm predicted functional and extra-functional behavior of the finished ECU, e.g. power usage, heat emission, or fault tolerance. While the ECU is close to its final form, the environment may be simulated; unlike the former case, full modeling of physical environment properties is neither required nor desired. This way, the testbed is available long before the real device is built, designers can inject fault conditions that cannot safely be reproduced with real system components, and a misbehaving ECU cannot harm these (possibly expensive) components.

In this paper we focus on the latter case. Institutions not involved with actual construction of electromechanical components rarely encounter the first case. As a consequence, specialized simulation equipment like the dSPACE platform [2] may not be available in-house and may be difficult to fund. Instead, developers use hardware-in-the-loop (HIL) simulation with a mathematical model of the environment, modeled in Matlab/Simulink [3], for example.

Hardware-in-the-loop setups based on modern commodity PCs are not real-time capable as these no longer come with low latency communication interfaces

like RS-232. The ubiquitous USB interface operates with a frame length of 1 ms (USB 1.0) or 125 µs (USB 2.0), while a typical motor model built in Simulink might need a time resolution of 100 µs. Furthermore, general-purpose operating systems cannot guarantee real-time behavior for software. As a result, HIL simulations frequently run at a fraction of real-time speed, which limits their usefulness.

So real-time testbeds must rely on additional hardware. Possibilities range from PC add-on cards with low-latency high-bandwidth communication interfaces to dedicated DSP- or FPGA-based simulation hardware, posing significant costs.

In this paper we present a case study based on a model-based workflow to design a new control unit for electric cars as part of the MotorBrain project [4]. We perform real-time simulation using the low-cost Raspberry Pi single-board computer [5] as dedicated hardware testbed and compare results to software simulations.

The key contributions of this paper are:

- Reduction of the cost of real-time hardware simulation testbeds to the point where the decision to build one does not affect project budgets and may even be made ad-hoc. As we will show, our methodology is portable; it can be used with many other devices that may already be available, making it essentially free.
- Rapid prototyping for this low-cost simulation testbed by fully integrating it into a model-based design flow.
- A fast and lightweight framework for direct access to peripherals of the Raspberry Pi including drivers for UART and SPI communication.

In the next section, we give an overview of related work. In Sect. 3, we show our model-based design flow. Section 4 gives details about the simulation hardware. Finally, we evaluate our use case in Sect. 5. Section 6 summarizes our findings.

2 Related Work

Real-Time (and faster) Simulation. In [6], the authors show high-performance environment simulation using FPGAs. They can simulate environments in much greater detail than we can, but at significant equipment and development cost. They improve on that in [7] by using parameterized models, but FPGAs remain platforms difficult to target.

Model-Based Design and Simulation. Since model-based design is state-of-the-art in research and industry, many papers cover hardware simulation in such a context. Two detailed papers regarding this topic are [1] and [8]; the former explains model-based hardware design in great detail, while the latter focuses on rapid prototyping made possible through hardware simulation. Both papers deal with simulation of the control unit inside a real analog environment. While we evaluate the opposite constellation, both papers employ the dSPACE platform,

which applies to our use case as well. However, we try to get hardware-assisted simulations using inexpensive equipment.

Low-Cost Simulation. The authors of [9] build such a platform from off-the-shelf x86-based PC hardware, while we use the Raspberry Pi single-board computer. Their approach is very similar to ours: we use almost the same system software, but they impose a custom solver/simulation environment. We just assume the availability of executable code for environment models to be simulated.

Another difference is that we do not use dedicated data acquisition/signal generator hardware. We transmit digital sample values instead of analog signals so that we don't need any active components beyond the Raspberry Pi.

Raspberry Pi. Others also take advantage of the low-cost availability of the Raspberry Pi. In [10], the authors describe the benefits of using the Raspberry Pi for inexpensive hardware simulations, albeit in a completely different context, and using stock Linux system software. Similar to our results, they experience a distinct advantage of being able to afford a realistic simulation platform.

Bare-Metal Frameworks. Established APIs for microcontroller programming like the Arduino programming interface [11] help making programs readable (i.e., maintainable), but their abstraction tends to increase latency. Existing direct-access helpers (like [12]) are fast and extensive but provide little type safety and readability. Our framework uses strongly-typed register definitions, symbolic constants and inlinable helper functions in order to combine both advantages.

3 Model-Based Design Flow in the Automotive Domain

Model-based design manages complexity by means of abstraction and separation of concerns. Abstraction allows the designer to iteratively refine an abstract model of the system under design towards a final implementation. This process is accompanied by creating models of different viewpoints. For example, a functional model of the system describes system functions and their decomposition independently from their realization in hardware or software. Different viewpoint models would then be combined and refined towards the actual implementation.

Figure 1 shows the different types of models and their connections. Different models serve different tasks on different abstraction levels:

High Level System Design is where requirements and needs models are created using Excel sheets or requirements tools like DOORS [13], while functional aspects are captured and linked with requirements using EAST-ADL [14]. We do not cover this phase in more detail, since our focus is on lower abstraction levels.

Functional Modeling/Simulation targets functional modeling of algorithms and components. This enables functional simulation as well as simulation-based verification of components and systems. Some of the Matlab/Simulink models serve to create the final software code using code generation techniques.

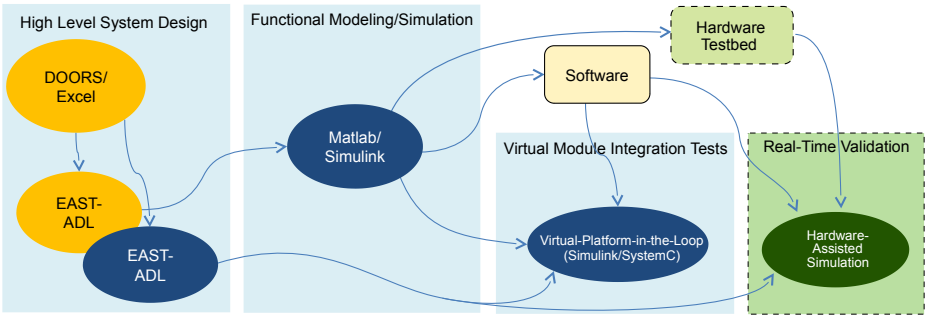

Fig. 1. A model-based design flow. Dashed Boxes highlight our contribution.

Virtual Module Integration allows us to combine functional models from different abstraction levels into system models in order to perform system tests early in the design process. It is related to HIL simulation, only that the hardware is simulated in a virtual platform model, resulting in virtual-platform-in-the-loop (VPIL) simulation [15]. This way, whole systems can be tested and validated even if final implementations are not yet available.

Real-Time Validation uses a hardware-simulated system environment to validate production-quality ECU and control software prototypes in a real-world setting while vehicle components are still under construction. When linking this simulation to an industrial driving simulator [16], we can capture effects ranging from thermal behavior through user interaction and feedback. Without the platform presented in this paper, we would have skipped real-time validation due to its cost and left it to the designers of other vehicle components.

4 A Real-Time Simulation Platform for Everyone

Figure 2 shows the components of a system simulation and their communication channels. In the illustrated case, an ECU is running in an environment fully simulated by a single hardware simulator. Expanded simulations may contain multiple ECUs, and environment components may be simulated by different physical devices. Simulation components communicate through low-latency links of varying bandwidth, depending on requirements and device capabilities.

An external interface allows simulation control (start, stop, user input, fault injection) and retrieval of timing measurements, operating parameters, etc. Figure 3 shows such a control and visualization interface running on a regular PC.

4.1 Anatomy of a Pi

The Raspberry Pi [5] (RasPi) is a single board computer based on a 700 MHz ARM11 SoC having 256 MiB RAM. Its release price is 25 USD for model A.

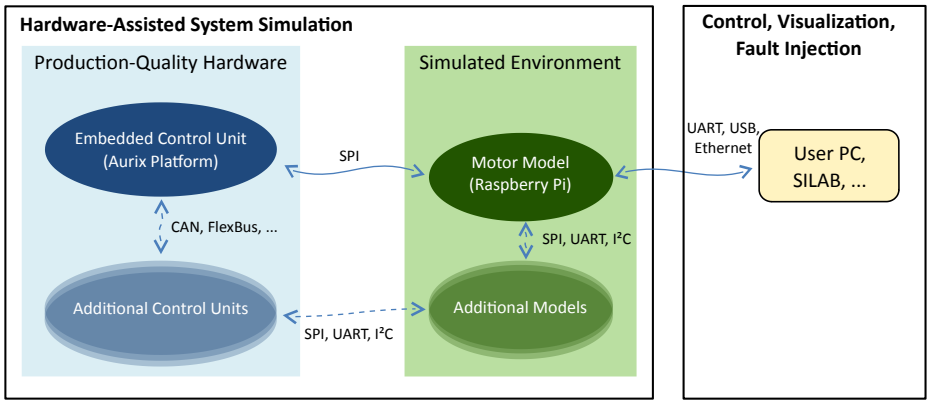

Fig. 2. Setup of a system simulation using our platform.

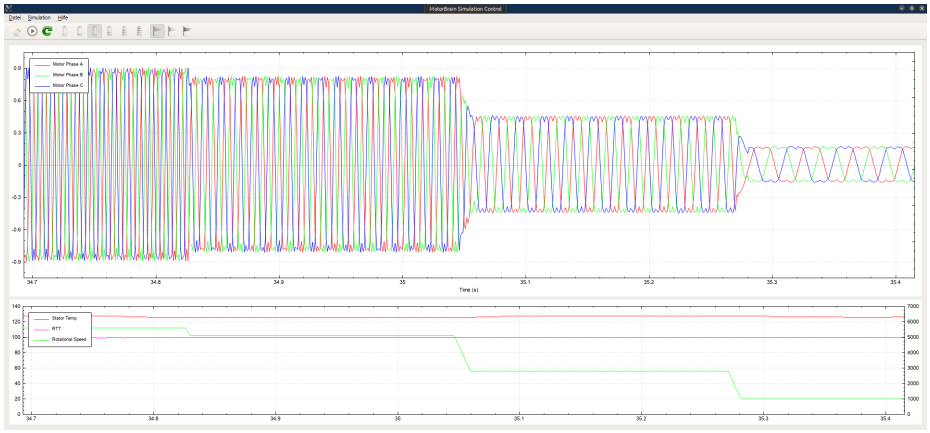

Fig. 3. Example of a control and visualization user-interface.

Model B has 512 MiB RAM and adds an Ethernet interface at a price of 35 USD. The RasPi runs the GNU/Linux operating system, and there is a lot of educational material available around using and programming the RasPi [17].

A general-purpose I/O (GPIO) pin header allows access to the low-latency communication interfaces we use:

- one asynchronous serial interface (UART),
- one serial peripheral interface (SPI),
- two two-wire serial interfaces (commonly called I²C), and
- on a secondary connector, a full-duplex serial audio codec interface (I²S).

Other on-board connectors include USB, Ethernet (only model B), and analog audio. Advanced interfaces (e.g. CAN or FlexBus) are meant to be added via daughter boards connected to the pin headers.

4.2 Challenges and Limitations of a Low-Cost Platform

For running simulations on the RasPi, system performance is an issue. The ARM core includes a single-precision floating-point unit, which helps for environment models built on mathematical algorithms. If performance is lacking by a small margin, the RasPi can safely be overclocked by about $10\% - 20\%$; some boards can been overclocked by more than 50% (up to 1.1 GHz has been reported).

In order to minimize cost, we do not use add-on boards for additional I/O. Instead, we try to get by with just the bare RasPi. This poses an additional challenge: The RasPi has no usable digital/analog and analog/digital converters. Its two internal PWM generators are wired to the headphone jack via an audio-quality RC filter. This may fit some use cases, but we ignored that possibility in order to avoid potential signal integrity issues.

Instead, we bypass analog signal synthetisation and sampling. We assume that there is a common API call for data acquisition on the ECU, and we replace its implementation with one that talks to the RasPi via SPI (see Fig. 4). As we control the simulation environment on the RasPi, we can freely choose how to transfer simulation data.

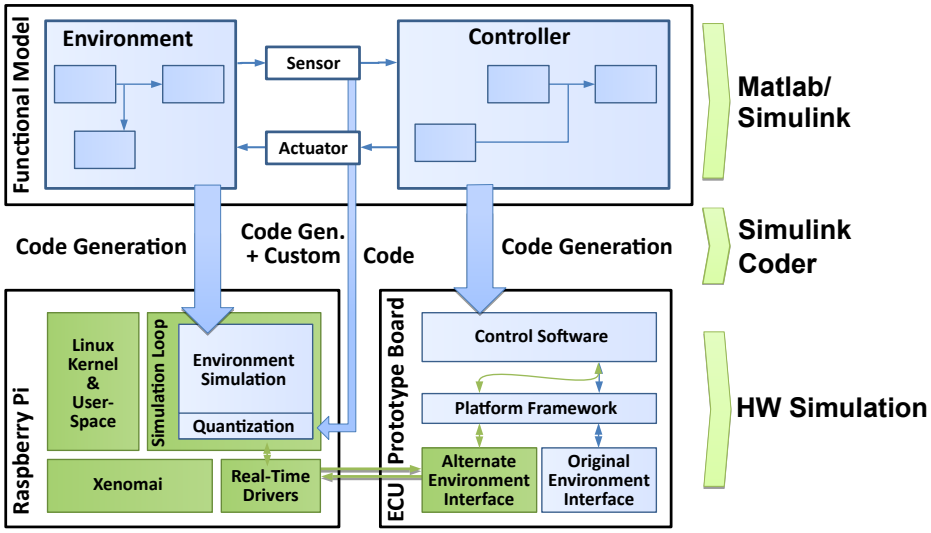

Fig. 4. Application architecture for our simulation approach. Shaded (green) boxes indicate custom simulation components, light (blue) boxes are unmodified parts.

With this scheme we get low latency and high bandwidth. The RasPi is able to perform SPI transfers at up to 32 Mbit/s[1]. Using direct hardware access, FIFO fill level dominates latency. This depends on the capabilities of the target ECU,

[1] The peripheral can be configured for up to 125 MHz, but signal quality is insufficient.

although virtually every system-on-a-chip has some sort of SPI interface, even if it may require soldering to access it.

4.3 Real-Time Operating System and Bare-Metal Drivers

We use a system software architecture like that shown in [9], with small changes. The Raspberry Pi runs one of the available Linux distributions, preferably of the "hard-float" ABI variant. However, we build a custom Linux kernel containing the Xenomai real-time patch [18].

Using Xenomai, developers can write applications using a subset of the regular POSIX C API, test and deploy them in a debug-friendly Linux environment, and afterwards trivially modify them to become hard real-time processes in the Xenomai environment using Xenomai's POSIX emulation. They can decide at runtime when to switch to a real-time context; for example, applications can load configuration data from the file system or even from a network server, process it, and only then switch to real-time mode.

This approach combines ease of development and deployment provided by Linux with real-time behavior of Xenomai and full hardware control through bare-metal style programming. There is no need for special programmer hardware or cross-compilation toolkits. Programmers use the regular POSIX API, except for access to the communication interfaces. Using our Raspberry Pi hardware access API [19], a single source file works in both contexts.

In case simulation runs significantly faster than required, real-time threads can let the Linux kernel take over to access USB, Ethernet, or the file system. Xenomai will ensure real-time operation by preempting the kernel if required.

For the low-latency communication interfaces listed in Sect. 4.1, we wrote a light-weight driver framework that allows direct hardware access to the RasPi peripherals independent of execution mode. It includes declarations for all known hardware registers and helper functions for GPIO, UART, SPI (including limited SPI slave support), and System Timer peripherals [19].

Since the peripherals we use are limited, we don't need the complexity of interrupt-driven operation and DMA, further simplifying drivers. Our model of operation works fine with simple polling of all relevant peripherals. As a result, most functions of our API boil down to one or two lines of C code. This even improves timing predictability as there are no competing data transfers on the internal bus.

4.4 How to Run Simulation Code Predictably

Based upon this setup, we can use any C program to run our simulation[2]. This can be an environment model using an in-house solver, code generated by Matlab/Simulink Coder [20], or a third-party IP model. Since we want to run in a real-time context, there are some restrictions; application developers not experienced with real-time code may have to adapt [21]. However, these issues are no

[2] Even binary objects could be used, as long as they were compiled for ARM.

problem for mathematical simulation models or for our simulation framework as shown in Fig. 4.

Since digital systems cannot exactly reproduce continuous analog systems, simulation will be divided into discrete time steps [15]. Listing 1 shows an example for a processing loop: All we need to do is to poll for updated input signal values (line 8), run the simulation for a single time step (line 12), and then publish updated output signal values (line 16). Finally, we wait until the next time step begins (line 25). We process control messages in a less timing-critical part of the loop (line 21). This doesn't even warrant a distinct software framework: our full implementation consists of less than 100 lines of code.

Listing 1 Core processing loop. MOTOR_PROCESS is the entry point into the environment model, SPI and UART functions are part of our generic driver framework. The remaining functions are custom helpers skipped for brevity.

```
1  float motor_data[4];
2  int i;

4  for (;;) {
5      /* fetch current ECU outputs */
6      for (i = 0; i < 4; i++) {
7          spi_write(0); /* SPI needs a write for each read */
8          motor_data[i] = convert_to_float(spi_read());
9      }

11     /* run a simulation step */
12     motor_process(motor_data);

14     /* send updated ECU inputs */
15     for (i = 0; i < 4; i++) {
16         spi_write(convert_to_uint8(motor_data[i]));
17     }

19     /* check for simulation control commands */
20     if (uart_poll(1)) {
21         process_control_message();
22     }

24     /* sleep until the next simulation time step begins */
25     wait_for_next_step();
26 }
```

Communication is straightforward: We transmit raw sample values in native format, just like ADC/DAC/PWM hardware registers would contain them. These values are passed from/to the existing ECU code as-is, so we only need to

redirect the appropriate register accesses. As a result, our communication strategy does not lower accuracy. On the environment side, we convert them back to floating-point values as required by the environment model (lines 8 and 16).

5 It Works—Real-Time, Even!

In this section, we evaluate our methodology. The MotorBrain team [4] designed a control unit for a three-phase brushless DC motor as part of an electric car design. We tested production-quality hardware in our low-cost hardware testbed to confirm software simulation analysis.

We used a Matlab/Simulink model of the motor control algorithm to be implemented on the ECU and a model of the DC motor, i.e. environment behavior, as reference models for subsequent implementation steps (see Fig. 1).

First, we analyzed and validated ECU timing behavior using a virtual-platform-in-the-loop approach [15]. By this approach, we validated the functionality of our implementation and its mapping to the target platform. Furthermore, we performed real-time analysis to determine timing requirements. As shown in Table 1, the average duration of a time step is 9.3 µs, which is far away from violating the timing requirement of 100 µs.

After the hardware prototype of the ECU was available, we built the hardware simulation.

Table 1. Timing as determined by virtual module integration tests.

Action	Average Time
Sensor Polling	3.0 µs
ECU Calculation	4.7 µs
Actuator Updates	1.6 µs
Total	9.3 µs

5.1 Embedded Control Unit Architecture

The ECU is based on a heterogeneous multi-core and multi-processor architecture containing safety features required for the automotive domain. The main processor is an Infineon AURIX system-on-chip with three TriCore architecture cores [22]. Among others, the ECU board contains interfaces for controlling motor power electronics via PWM, ADCs to measure motor current, and a rotational position sensor. It provides no easily accessible general-purpose chip-to-chip interfaces (like SPI).

The ECU software uses a classic bare-metal approach. A manufacturer-supplied software framework provides basic system management and drivers, but no explicit real-time operating system is employed (see Fig. 4).

5.2 System Setup

We wanted to use the real ECU with a simulated motor model. The control unit generates four output values, one PWM-derived voltage per motor phase, and one temperature sensor measurement. It needs six inputs to operate: effective current per phase, rotational speed, requested torque, and electric flux of the motor. Reduced to 12-bit quantized values, this results in communication payloads of 6 and 9 bytes, respectively.

The main problem we encountered was that the ECU doesn't have external access to an SPI slave interface. Our Simulink models use a time step of 100 µs, and we need two data transfers per step, so the on-board USB interface won't work. The board's CAN bus interface is too slow: at 1 Mbit/s, only 100 bit can be transmitted in a single simulation step. With an overhead of about 48 bit for a basic CAN frame, two packets barely fit into that period—without payload. Moreover, time spent for transmission can't be used for calculation.

The ECU board only had one solder-less alternative left, SPI (master) via the SD-card slot. Unfortunately, the Raspberry Pi doesn't have an SPI slave interface. Using the I^2S interface and a synchronization procedure, we made the Raspberry Pi behave like an SPI slave without bit-banging, i.e. at speeds of 20 Mbit/s and more. At first glance, this appears to be fragile, so we used a simple framed protocol with CRC error detection. It confirmed the validity of our approach: not even a single CRC error occurred during our experiments.

5.3 Results

We measured the duration of individual intervals of a single simulation step using the integrated timer peripheral of each board and transmitted the measured values in-band with ECU outputs and simulation control messages. Timing variation was in the order of microseconds for all measurements.

The final system meets the requirement of 100 µs per simulation time step reliably, as shown in Table 2. However, the remaining 11 µs were not enough for Xenomai to switch back to Linux and allow the network stack to operate. The Raspberry Pi ran our simulation exclusively, so we used the real-time capable UART interface for control and visualization.

ECU timing matches the results of VPIL simulation. Due to the complex style of SPI communication (see Sect. 5.2), the overhead introduced by our simulation approach is significant. Given better SPI support, data packets would need about $3 - 4$ µs at 20 Mbit/s, so within the same order of magnitude as native I/O (as shown in Table 1). Nevertheless, the final simulation system works and outputs the same values as our Matlab/Simulink model.

When first testing our simulation, at some point simulation time of the motor model increased until a single simulation cycle took longer than 100 µs. At the same time, data values were wildly incorrect, to the point that the motor stopped operating. Since simulation ran autonomously, we ran it for much longer times than VPIL simulations previously did. We found out that the model had a problem that led to erroneous long-term feedback. That accumulated until the

Table 2. System simulation timing, rounded to the nearest microsecond.

Action	Average Time
Sensor Polling (transmission of ECU inputs)	26 µs
ECU Calculation	5 µs
Actuator Updates (transmission of ECU outputs)	18 µs
Motor Model Calculation	40 µs
Total	89 µs

control algorithm failed. The simulation was actually correct—it behaved exactly like the model, but previously, no one ran the model for extended periods.

After fixing the model, we deployed updated (generated) code in mere minutes. Finally, our ECU ran for many hours with no problems encountered, always staying within the predicted time limits.

6 Conclusion

In this paper we have used a Matlab/Simulink model of a three-phase brushless DC motor control system to generate the core ECU algorithm code, perform timing validation on it, and validate our predictions in a mostly auto-generated real-time hardware testbed.

We have shown how to build a real-time capable hardware testbed out of the inexpensive Raspberry Pi. We used Linux running under control of the Xenomai real-time layer as deployment platform. Xenomai's POSIX API emulation allowed us to re-use established software development tools and experience, minimizing development time. We needed about one week for the first iteration of the hardware testbed, including writing our direct-access framework [19].

We only used baseline features of the RasPi that should be included on virtually every system-on-a-chip. As a result, our approach works with any computer that may happen to be available and that has suitable communication interfaces, and it will work with future incarnations of the Raspberry Pi. We could increase performance by using the RasPi-specific DSP and GPU, but it may actually be more cost- and time-effective to parallelize simulation across multiple RasPi boards (as done in [10]) than to spend additional developer time.

Finally, we have shown the value of real-time simulation for algorithms and methodology research. We demonstrated a case where it exposed long-term stability issues. Due to the integrated design flow, we were able to fix an issue in the model and have the fix propagate into the virtual platform and the real-time simulation in minutes. The final demonstration platform runs as expected for hours and will be part of a more elaborate vehicle simulator.

Acknowledgement. This paper has been partially supported by the MotorBrain ENIAC project under the grant (13N11480) of the German Federal Ministry of Research and Education (BMBF).

References

1. Monti, A., Santi, E., Dougal, R.A., Riva, M.: Rapid prototyping of digital controls for power electronics. IEEE Trans. Pow. Elec. **18**(3) (2003) 915–923
2. dSPACE GmbH: dSPACE Systems. http://www.dspace.com/en/inc/home/products/systems.cfm (2013)
3. The MathWorks, Inc.: Matlab/Simulink. http://www.mathworks.de/products/simulink/ (2013)
4. MotorBrain Consortium: MotorBrain. http://www.motorbrain.eu/ (2013)
5. The Raspberry Pi Foundation: Raspberry Pi | An ARM GNU/Linux box for $25. http://www.raspberrypi.org/ (2012)
6. Huang, C., Miller, B., Vahid, F., Givargis, T.: Synthesis of networks of custom processing elements for real-time physical system emulation. ACM Trans. Des. Autom. Electron. Syst. **18**(2) (April 2013) 21:1–21:21
7. Miller, B., Vahid, F., Givargis, T.: MEDS: Mockup Electronic Data Sheets for automated testing of cyber-physical systems using digital mockups. In: DATE. (2012)
8. Trujillo, O.A., Hoyos, F.E., Garcia, N.T.: Design, Simulation and Experiment of a PID Using Rapid Control Prototyping Techniques. In: SICEL. (2011)
9. Lu, B., Wu, X., Figueroa, H., Monti, A.: A low-cost real-time hardware-in-the-loop testing approach of power electronics controls. IEEE Trans. Ind. Elec. **54**(2) (2007) 919–931
10. Tso, F., White, D.R., Jouet, S., Singer, J., Pezaros, D.: The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures. In: ICDCS-CCRM. (2013)
11. The Arduino Project: Arduino - Reference. http://arduino.cc/en/Reference/HomePage (2013)
12. Gottschlag, M.: VideoCore Tools. https://github.com/mgottschlag/vctools (2013)
13. IBM: Rational DOORS. http://www.ibm.com/software/products/us/en/ratidoor (2012)
14. MAENAD Consortium: EAST-ADL Language Specification. (2012)
15. Fakih, M., Grüttner, K.: Virtual Platform in the Loop Simulation for Accurate Timing Analysis of Embedded Software on Multicore Platforms. In: ASIM Konferenz STS/GMMS, Wolfenbüttel. (2012)
16. WIVW GmbH: Driving simulation software SILAB. http://www.wivw.de/ProdukteDienstleistungen/SILAB/index.php.en (2013)
17. The Embedded Linux Wiki Community: RPi Hub - eLinux.org. http://elinux.org/RPi_Hub (2013)
18. Gerum, P.: Xenomai - Implementing a RTOS emulation framework on GNU/Linux. Technical report (April 2004)
19. Walter, J.: Raspberry Pi Integrated Peripheral Access without Operating System Drivers. https://github.com/offis/raspi-directhw.git
20. The MathWorks, Inc.: Automatic Code Generation - Simulink Coder. http://www.mathworks.de/products/simulink-coder/ (2013)
21. The Xenomai Project: Porting POSIX applications to Xenomai. http://www.xenomai.org/index.php/Porting_POSIX_applications_to_Xenomai (2013)
22. Infineon Technologies: AURIX – Safety joins Performance. http://www.infineon.com/cms/en/product/microcontrollers/32-bit-tricore-tm-microcontrollers/aurix-tm-family/channel.html?channel=db3a30433727a44301372b2eefbb48d9 (2013)