

JITPCB

Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan W. Haldane, and Richard Lin

EECS Department, UC Berkeley

{ jrb|biancolin|abuchan|dhaldane|rlin }@berkeley.edu

Abstract—Commercialization of desktop milling machines has made rapid Printed Circuit Board (PCB) fabrication accessible. Unfortunately, PCB design for embedded and robotic systems is still a tedious and time consuming activity. In this paper, we present a technique, Just In Time Printed Circuit Board (JITPCB) for designing PCB systems at speeds commensurate with the capability of desktop PCB milling machines. We propose designing boards by writing software circuit generators that wire together and lay out circuit components in a hierarchical and reusable fashion. We have developed a declarative design mechanism allowing users to specify desired input and output peripherals and well as application code. Given this input, our system produces a complete working circuit board design, along with necessary initialization and networking code. Our system is an open framework that allows users to create a set of highly reusable parametric hardware/software modules. We demonstrate our approach by showing some common robotics applications designed with JITPCB to show its utility and generality.

I. INTRODUCTION

A. Motivation

Robotics, Internet of Things (IoT) technology, and interactive art have all benefited from the ubiquity of open hardware and software platforms. This technology revolution is exemplified by the Arduino prototyping platform, with over one million units estimated in distribution as of 2014¹. Sharing enabling technology is relevant for communities that benefit from rapid iteration on proof-of-concept designs, such as educators, researchers, and hobbyists. The success of these projects is based on unlocking creative potential for non-experts via well designed and published standards that allow expression of designs in high-level concepts, active communities that support distribution and reuse of designs, and availability of low-cost prototyping materials. Affordable rapid prototyping tools are becoming increasingly available, such as 3D printers, laser cutters, and milling machines, furthering creative accessibility by reducing fabrication time and cost.

The majority of embedded systems can be described as a collection of input and output peripherals connected to one or more computational units with information shared over a communication network. When designing these systems, correctly matching functional goals (control the position of a motor based on encoder input)

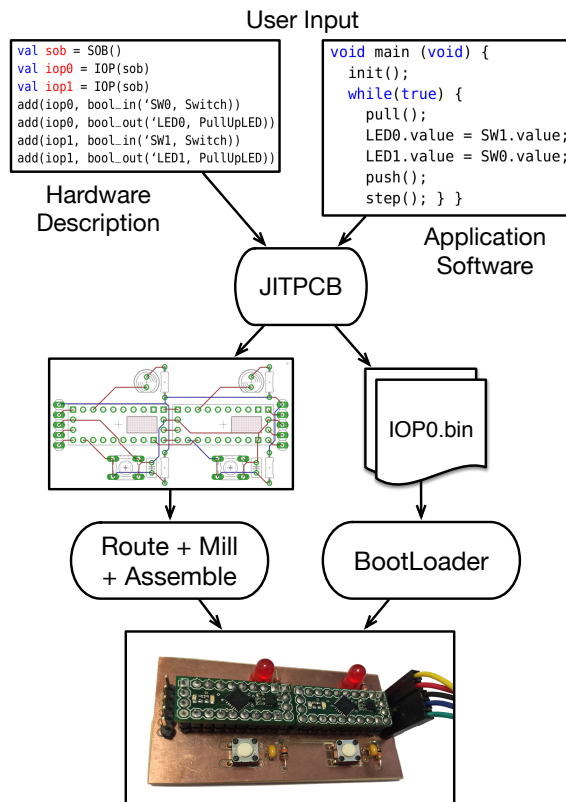


Fig. 1. System description, application code, and resulting PCB for Hello World application.

to electrical implementation details (pin and peripheral assignments, power distribution, driver configuration) remains a time-consuming and error-prone process, even with expert knowledge and commercial software tools. Furthermore, expressing these systems in a format that makes design pattern reuse and distribution remains problematic. The field of Very Large Scale Integration (VLSI) design for integrated circuits has addressed many of these issues using design capture tools with textual source code descriptions of modular, parametric, and hierarchical design units. Inspired by this success, we seek to transfer many of these concepts to embedded system and schematic capture design.

We propose an approach, Just In Time Printed Circuit Board (JITPCB), to address these issues. JITPCB expresses these systems as high-level software, which in

All authors contributed equally to this work.

¹<http://medea.mah.se/2013/04/arduino-faq/>

turn generates one or more custom circuit board designs and supporting infrastructure code that enables rapid manufacturing, assembly, and application software development for embedded systems. Fig. 1 shows a simple “Hello World” application as an example of our approach. By leveraging high-level programming language concepts such as inheritance and functional programming, design productivity and modularity is improved over conventional schematic capture tools. This approach is extensible with new types of peripherals and computational units, allowing users to integrate custom or commercially available electrical components. We show that our approach for Hardware/Software (HW/SW) co-designed systems can produce a variety of functional embedded systems with very little user-supplied code, producing workable results for circuit board area, use of processor units, and control loop frequency.

B. Prior Work

There are several common approaches used to prototype embedded hardware. The most accessible choice for non-expert users is to combine off-the-shelf peripheral boards, creating an aggregated modular board system which has the desired functionality. Examples of these modular circuit board systems include Little Bits[3], Arduino [8] and BeagleBone [5] Shields, and Gadgeteer[12].

Modular circuit prototyping platforms make mixing and matching peripherals with processors easier, but these approaches suffer from bulky and expensive connectors and boards, poor mechanical integration, and are still manually assembled leaving room for (often costly) connection mistakes. Finally, these systems do not scale well to more complex designs involving multiple microcontrollers.

Several HW/SW co-design approaches have been explored to automate the process of generating systems from high-level specifications. Mehta et al. address generation of robotic designs with modular software descriptions, but is still dependent on manual connection of devices [7]. Slomka et al. [11] and Sarma et al. [10] focus on describing modular hardware description language designs for prototyping embedded systems, but provide little evidence for being able to quickly interface with a variety of peripheral devices.

A custom-designed PCB overcomes some these issues but for the most part, PCB CAD tools are tedious, manually intensive, and rely on proprietary graphical interfaces. Users manually place components and route wires. Any design change often results in ripping up a portion of the design and reworking it by hand. Finally, it is difficult to reuse physical design of higher-level modules.

Prior work has targeted automation of the PCB design process. PHDL [9] is a Python-based circuit board construction kit. The user can parameterize

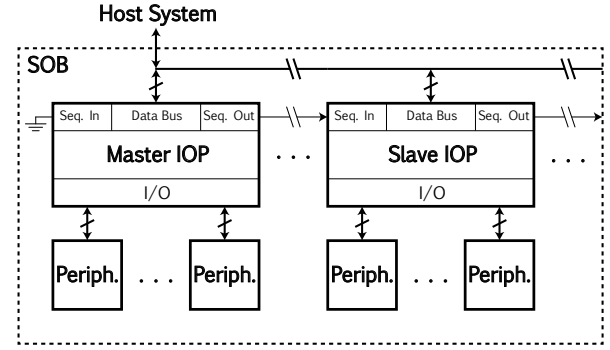


Fig. 2. JITPCB system organization.

board construction, but cannot specify the layout of the circuit components from within the software description. Chisel [2] is a Scala-based RTL design library. Like JITPCB, users write programs to construct circuits, but unlike JITPCB, it has neither layout nor PCB specific facilities, instead targeting FPGAs and ASICs. Lava [4] is a Haskell-based circuit description language for gate level descriptions, mostly used for FPGA programming. Like JITPCB, Lava has a set of layout functions for hierarchically specifying the location of hardware components, but to our knowledge was never used to describe circuit board construction

No system to date has provided the level of abstraction afforded by JITPCB and the ability to deliver both hardware and software together to minimize the ensuing integration costs. Furthermore, unlike previous approaches, our JITPCB system generates the hardware and software for a network of microcontrollers.

II. METHODS

A. System Overview

A JITPCB embedded system design is a System on Board (SOB), a collection of *peripherals* connected to a network of Input/Output Processors (IOPs), as illustrated in Fig. 2. Each peripheral provides a desired functionality to the system, such as sensing ambient light or driving a motor. The IOPs and peripherals may be distributed across one or more physical PCBs as required by the overall application.

An IOP is a microcontroller that exposes electrical pin functionalities (e.g. analog input, PWM output), and has basic power regulation and isolation features. Peripherals are connected to IOP pins based on their functionality requirements, and common fixed voltage supplies are wired to peripherals and IOPs.

The first IOP allocated is designated as the Master IOP, and the remaining are Slave IOPs. The IOPs are connected to a data bus (I2C in our implementation) that is used to initially program the devices from a host system, and exchange data between the master and Slave IOPs during operation. A handshake signal is connected between sequential IOPs (or ground for

the master), allowing them to be assigned unique bus IDs during startup.

Fig. 3 details the process for automatically generating and programming a JITPCB system given a system hardware description and application software. A user specifies a collection of peripherals and processors (with optional layout constraints) using the Stanza [6] programming language, and application code in C++. JITPCB generates PCB geometry files with the help of the EAGLE CAD automatic trace router, as well as system infrastructure code. This support code is compiled with user code and microcontroller-specific libraries to produce binaries for each IOP. The PCB files can be turned into functional boards with a rapid-prototyping PCB mill, or a conventional PCB etch process. Once the boards are populated with components, a host system programs the IOP binary files on the assembled SOB with bootloader software to achieve a working system.

B. Hardware Description

A key aspect of JITPCB is allowing the user to describe hardware in a declarative way (e.g. “Place and wire 64 LEDs arranged in an eight by eight grid.”) rather than an imperative one (e.g. “Place LED0 at [6.5, 8.4] and wire to pin PB7, place LED1 at ...”). We chose to implement this hardware description aspect of JITPCB in Stanza because it is an open-source functional programming language that cleanly supports expressive syntactic extensions, though any language that supports object orientation and functional programming could be used in practice. By embedding hardware description in a fully-featured programming language, we can describe components and systems in a modular way that enables enforcing declarative “correct-by-construction” design principles, with efficient and convenient code reuse, as well as allowing users to extend the language with custom design patterns or constraints.

JITPCB currently produces EAGLE² format PCB files. These files express netlist connectivity between electrical components, as well as the physical layout and copper routing geometry of PCBs in a well-defined XML format. EAGLE CAD is a popular and (for small designs) free software suite that can also automatically route copper traces between components once connectivity is specified. We chose to leverage the EAGLE CAD suite in JITPCB for this autorouting feature, and widely-available component description libraries.

Fig. 4 shows how components and peripherals are described in JITPCB Stanza. *Components* are the lowest level of hardware objects. They specify the correspondence between physical package pins and logical connection signals. Component definitions reference an EAGLE library and package name (e.g. the “LED” library and “LED5MM” package), and identify the signals that can be connected for that component using the “pad”

Component Descriptions:

```
defcomponent LED ("LED", "LED5MM") :
  pad A
  pad K

defcomponent Resistor ("RCL", "0204/7") :
  pad 1
  pad 2
```

Module Description:

```
defmodule PullUpLED () <: Module:
  ;; interface
  inherit gnd
  inherit v3
  sig io
  ;; subcomponents
  mod l : LED("red")
  mod r : Resistor("1k")
  ;; netlists
  sig up = [r.1, l.A]
  sig v3 = [r.2]
  sig gnd = []
  sig io = [l.K]
  ;; layout
  lay HBox([Rot(r,90), Rot(l,90)], 1.0)
```

Fig. 4. Modular hardware description in Stanza.

keyword. Components can then be grouped and wired as *modules*; these specify logical connections between signals as netlists, and optionally declare the physical layout of components. Basic parametric layout functions are provided such as horizontal containers (HBox) and rotate by angle (Rot), which can easily be composed for more complex layout functions (see Sec. III-A). The PullUpLED module thus describes a red LED with 1 k Ω pull-up resistor arranged horizontally.

Peripherals and IOPs are special types of modules that respectively require or provide electrical pin functionality. Fig. 5 shows how a “DigitalOut”³ pin is required for the “io” signal for the module passed to the “bool_out” function. Similarly, the Stanza description of an IOP lists all of the functions available on each IOP pin. When the add() function is called with an IOP and peripheral reference (Fig. 1), the next IOP pin providing the DigitalOut function is greedily assigned to be wired to that peripheral’s designated pin.

Users are free to explicitly assign peripherals to IOPs as in the Hello World example, but the main convenience of the correct-by-construction paradigm is achieved by the default behavior algorithms that are run as a final stage of all JITPCB designs. These procedures can assign any unspecified peripherals to existing IOPs, or even instantiate new IOPs as required to satisfy all peripheral pin functions. They also ensure that power supplies are wired to all global voltage signals (designated by the “inherit” keyword in module descriptions), and that the appropriate data bus and sequence signals are wired to IOPs. Finally, any

²<https://cadsoft.io/>

³The single tick-mark preceding DigitalOut designates an enumerated “Symbol” type in Stanza.

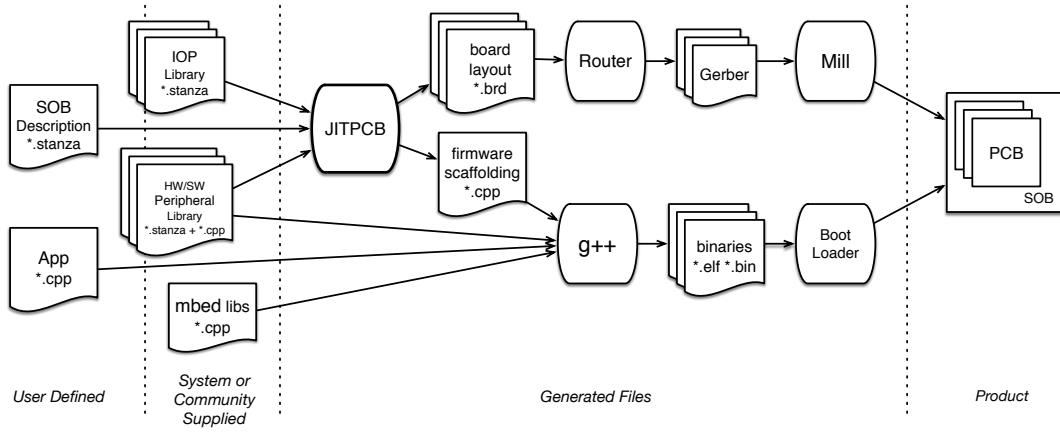


Fig. 3. JITPCB workflow.

components with unspecified layouts are placed in free space on circuit boards, with user-tunable heuristics for simplicity or signal length. In all cases, components are placed with sufficient buffer space to guarantee that the EAGLE autorouter will complete signal trace routing.

C. Software Model

Once a collection of peripherals and IOPs is described in Stanza, JITPCB generates a software environment for that particular SOB that allows high-level application code to be run on the system that shields the user from low-level details. The goal of this environment is to allow the user to reference inputs, outputs, and behaviors of peripherals without needing to know to which IOP they are physically connected. JITPCB achieves this abstraction by generating most of the configuration and communication code that runs on the Master and Slave IOPs based on the SOB hardware description, and using the concept of peripheral *ports* to share information between application code and peripherals.

A peripheral port is effectively a member variable of a peripheral C++ class that is synchronized between the Master and Slave IOPs. These ports behave similarly to memory-mapped registers in microcontroller devices in that reading or writing them can simply transfer data, or trigger more complex behaviors in the peripheral driver code. For example, the application code in Fig. 1 shows reading the binary state of switches on one IOP, and assigning that to the binary output value of the LED on the opposite IOP. In user code, this only requires referencing the “value” variable of the LED and switch instances in the application code, as the synchronization and peripheral functionality is handled by the underlying JITPCB support software. The JITPCB-generated `master.h` header that must be included in a JITPCB application file instantiates a peripheral object for each SOB peripheral constructed with the appropriate pin names corresponding to those assigned to the peripheral. It also provides the `init()`,

Peripheral Interface (Stanza):

```
defn bool_out (name:Symbol, hw:Symbol -> PeripheralModule) :
  Peripheral(name, "BoolOut", hw,
    [Pin('DigitalOut', pad#io)],
    [Port("value", IntType(0))])
```

Peripheral Interface (C++):

```
// Automatically generated Peripheral and Master class
class BoolOutPeriph : public Peripheral {
public:
  int value;
};

class BoolOutPeriphMaster : public BoolOutPeriph {
  // Initialize and synchronize local class variables
  // to Slave instantiations over network
};

// User completed Slave functionality class
class BoolOutPeriphSlave : public BoolOutPeriph {
public:
  BoolOutDevice(PinName pin) : dout_pin(pin) { }

  void step(void): {
    dout_pin.write(value);
  }

protected:
  DigitalOutPin dout_pin;
};
```

Fig. 5. Peripheral HW/SW interface.

`pull()`, `push()`, and `step()` functions, which respectively initialize all IOPs, retrieve and write Slave data, and execute Slave `step()` functions.

The `step()` function in the C++ peripheral Slave class interface (Fig. 5) specifies how a peripheral uses port data. The Slave class constructor is also responsible for configuring the microcontroller appropriately based on the pins it has been assigned by the JITPCB framework. JITPCB will automatically generate the class prototype that declares all port variables indicated in the Stanza peripheral interface, and arguments for accepting pin name specifiers, as shown in Fig 5. Then the Slave class must interface with processor-specific libraries to achieve the desired physical behavior. In

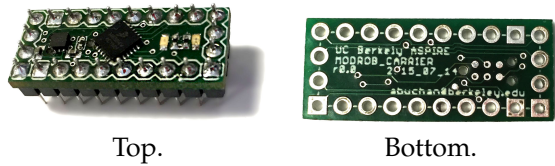


Fig. 6. Minimal IOP carrier module.

this example of the `BoolOutPeriphSlave`, the constructor simply initializes a digital output driver on the provided pin, and writes the integer stored in `value` to the `DigitalOutPin` instance in the `step()` function. For our software framework, which uses the `mbed` libraries [1] for ARM microcontrollers, this will connect the pin to ground for a 0-valued integer, and the processor supply voltage for any other value. By default JITPCB includes drivers for basic peripheral functions such as digital input and output, analog input, and PWM output. More experienced users can write drivers for other peripheral devices, and then share them with the community of JITPCB users to grow the number of supported peripherals.

III. RESULTS

To demonstrate the utility of JITPCB, we present three example applications that use a variety of digital and analog peripherals at varying levels of integration across one or more PCBs. These include a hand-held version of the popular video game *Pong*, a two-axis robotic arm with force feedback, and a mobile line-following robot. To verify that the designs produced working systems, the PCBs were fabricated with an Othermill desktop CNC machine⁴, assembled by hand, and programmed with the JITPCB bootloader. The construction process and behavior of each system is shown in the supplemental video.

In order to achieve more compact designs, we designed a minimal IOP carrier module with a Freescale MKL05Z32 microcontroller. This module exposes power and I2C bus contacts at both of the short edges of the board, and 16 I/O contacts on the long edges, making it easy to route the bus between IOPs and I/O to nearby peripherals. The Video Game and Robot Arm applications use this minimal IOP module, shown in Fig. 6.

It is difficult to present an objective measure of the usability of our tool. Some insight can be gained by observing the required lines of code (LoC) and the time required to run the tool for each example; specifically we provide:

- 1) **SOB LoC:** The Stanza LoC used to describe the SOB hardware, excluding library module definitions.
- 2) **App LoC:** The C++ LoC used in the application, excluding the `mbed` framework and library module drivers.

⁴<https://othermachine.co>

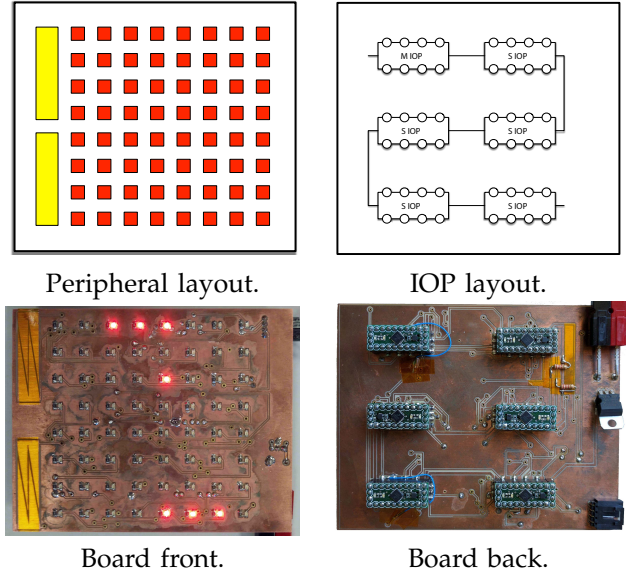


Fig. 7. Video game circuit board.

```
val led_rows = Vector<Layout>()
for j in 0 to 8 do :
  val led_row = Vector<Layout>()
  for i in 0 to 8 do :
    val name = symbol_join(["led-" i "x" j])
    val led = bool_out(name, PullUpLED)
    append(led_row, Space(V2f(12.0, 12.0), led))
  append(led_rows, HBox(led_row))
val sliders = [slider(0), slider(1)]
val lay = HBox([VBox(join(sliders, Fill()),
  HSpace(2.0), VBox(led_rows))]
val sob = SOB(lay)
```

Fig. 8. Video Game SOB description.

- 3) **Compile Time:** The time for JITPCB to generate the unrouted PCB files and software headers.
- 4) **Route Time:** The time for EAGLE software to produce routed PCB design file.
- 5) **Board Area:** The total board area of the generated PCBs.

The sum of the SOB LoC represents the total user input required to describe the hardware and application software of the system, the sum of the compile and route times represents to the total time to generate the design files from the user input. Run times were measured using a i7-920 processor with 12GB of RAM. Results for each of these applications are summarized in Table I.

TABLE I
QUALITY METRICS FOR EACH OF THE APPLICATIONS.

Application	SOB LoC	App LoC	Compile Time (s)	Route Time (s)	Board Area (cm ²)
Video Game	62	188	23.1	89.1	129
Robot Arm	44	99	24.3	52.9	185
Mobile Robot	34	166	24.3	32.4	92.9

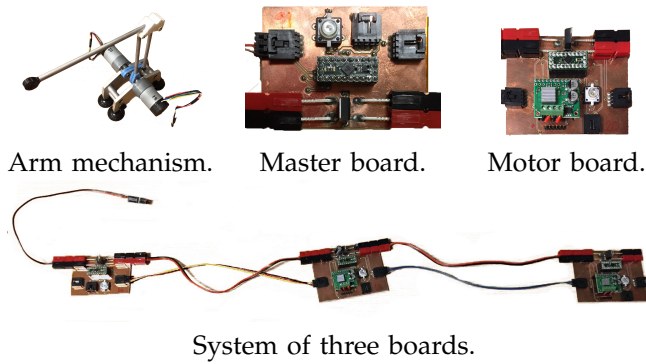


Fig. 9. Force feedback robot arm and circuit boards.

A. Video Game

The Video Game application allows users to move virtual paddles on an eight by eight grid LED display using two capacitive touch sliders. Fig. 8 shows the full listing of the hardware description code describing the user interface layout, and Fig. 7 shows the resulting layout of peripherals, IOPs, and resulting finished board. This application exemplifies how a user can, with very little code, design a parametric collection of peripherals, and allow the unspecified details like IOP layout and peripheral wiring to be handled by JITPCB. The bootloader for programming the IOP network also made the application software development process significantly more convenient than programming each IOP individually.

B. Robot Arm

This example shows the co-design of electronics and software for force feedback control of a two-axis robot arm, shown in Fig. 9. For this example we adapted the board generation code to create separate motor driver and master boards so they can be placed near the motors in the actual robot. This code automatically generates power and communication bus connectors, allowing convenient specification of a physically distributed modular system. As the two motor driver boards are identical, they can be interchanged on the robot, and the sequence assignment algorithm will correctly identify their location and apply control appropriately.

Each motor controller board contains a motor driver with an integrated current sensor and drives a gear-motor with position feedback from a quadrature encoder. The motor control boards run a local current-sensing torque control feedback loop, while the master runs a task-space impedance (spring and damping element only) controller. By separating these control loops so that one runs in the peripheral Slave class of the motor controllers, and the other on the Master, we achieve parallel distributed execution of code for reasonable real-time performance of the overall arm control.

C. Mobile Robot

The mobile robot uses a line-sensing camera to command a differential-drive robot chassis to follow

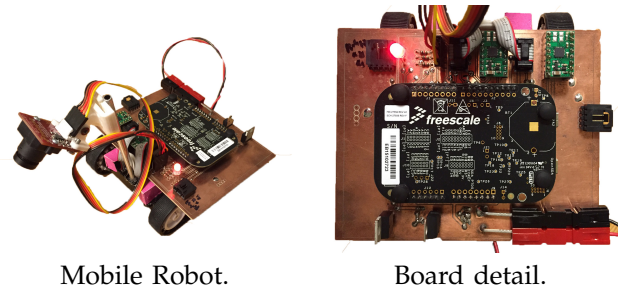


Fig. 10. Mobile line-following robot and circuit board.

a line on the ground. To show how the IOP module can be generalized, this application uses the more powerful and commercially available Freescale FRDM-KL25Z development board instead of the minimal IOP carrier board. Extending the framework to use this commercial board as an IOP only required describing the pin functionalities of the development board, and adding the pin header footprint to the IOP library. In general, a user will also need to adapt the on-board bootloader code and bus communication drivers when adding a new type of processor as an IOP. This example also demonstrates a more complex peripheral electrical interface for the line-scan camera sensor, which requires synchronization of analog input and digital clocking to read each pixel.

IV. CONCLUSION

We have introduced JITPCB and SOB, techniques for software defined hardware defined software which provide a powerful and practical PCB prototyping methodology. Representations of IOPs and HW/SW peripherals objects allow the system to synthesize hardware along with made-to-fit software, dramatically lowering integration challenges. Programmers write applications in an implementation independent fashion and let the design software glue together the various implementation artifacts. JITPCB represents an open framework for designing PCBs which we hope will allow us to realize the dream of a library of flexible components as digital artifacts.

While rapid iteration over designs benefits those who prototype embedded systems professionally, we also believe this is applicable to education, where students may not have the luxury of spending half a semester learning a board design tool. Providing a higher level of design would allow students to focus more on core material, while affordable fabrication facilitates integration with classes on a budget.

REFERENCES

- [1] ARM, "The ARM mbed IoT device platform," <https://www.mbed.com/>, 2014.
- [2] J. R. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzyniec, and K. Asanovic, "Chisel: Constructing Hardware In a Scala Embedded Language," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE Press, 2012, pp. 1212–1221.
- [3] A. Bdeir, "Little Bits," <http://littlebits.cc>, 2009.

- [4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *ICFP '98 Proceedings of the third ACM SIPLAN international conference on Functional programming*, 1998.
- [5] J. Kridner and G. Coley, "The Beagle Bone," <http://www.beagleboard.org>, 2011.
- [6] P. Li and J. Bachrach, "The Stanza Language," <http://www.stanzalang.org>, 2016.
- [7] A. M. Mehta, J. DelPreto, B. Shaya, and D. Rus, "Cogeneration of mechanical, electrical, and software designs for printable robots from structural specifications," in *IEEE Int. Conf. on Intell. Rob. and Sys.* IEEE, 2014, pp. 2892–2897.
- [8] D. Mellis, M. Banzi, D. Cuartielles, and T. Igoe, "Arduino: An open electronics prototyping platform," in *Proceedings of the Conference on Human Factors in Computing (alt.chi) (CHI'07)*. ACM Press, 2007.
- [9] B. Riching, R. Black, and B. Nelson, "PHDL," <http://phdl.sourceforge.net/1.0/index.php>, 2011-2012.
- [10] S. Sarma and N. Dutt, "FPGA emulation and prototyping of a cyberphysical-system-on-chip (CPSoC)," in *Rapid System Prototyping (RSP), 2014 25th IEEE International Symposium on*, Oct 2014, pp. 121–127.
- [11] F. Slomka, M. Dorfel, R. Munzenberger, and R. Hofmann, "Hardware/software codesign and rapid prototyping of embedded systems," *IEEE Design Test of Computers*, vol. 17, no. 2, pp. 28–38, Apr 2000.
- [12] N. Villar, J. Scott, and S. Hodges, "Prototyping with Microsoft .NET Gadgeteer," in *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction*, 2011.