# Modular-Things: Plug-and-Play with Virtualized Hardware

JAKE ROBERT READ* and LEO MCELROY*, MIT Center for Bits and Atoms, USA and Hack Club, USA
QUENTIN BOLSEE, MIT Center for Bits and Atoms, USA and Vrije Universiteit Brussel, Belgium
B. SMITH, Hack Club, USA
NEIL GERSHENFELD, MIT Center for Bits and Atoms, USA

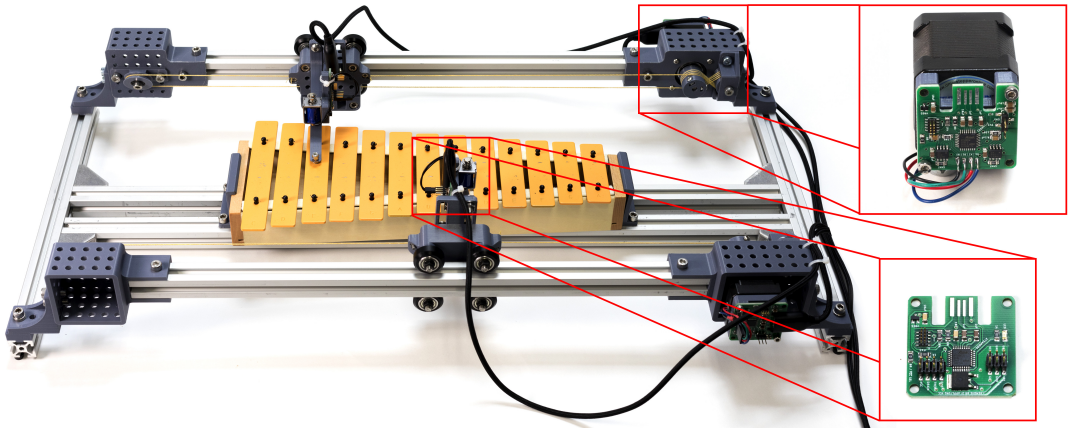

Fig. 1. A xylophone music machine built with the Modular-Things framework, comprising four modules: two stepper motors for motion, and two DC load drivers for triggering the mallets with electric coils.

We present a collection of tools for building plug-and-play modular physical computing systems that we call Modular-Things. Our tools consist of a set of single purpose embedded devices, a link layer agnostic message passing system for communication between devices, and a web-based programming environment. The devices are dynamically discovered and virtualized into software objects that can be programmed in the web IDE. We tested Modular-Things in a classroom setting where groups of novice machine builders constructed custom machines that integrated embedded systems modules with high-level responsive interfaces built with web and computer vision technologies. Users also extended our system by constructing new customized devices.

CCS Concepts: • **Networks → Programming interfaces**; • **Computer systems organization → Embedded systems**; • **Human-centered computing → User interface toolkits**.

Additional Key Words and Phrases: modular physical systems, virtualization, composability, prototyping frameworks

**ACM Reference Format:**
Jake Robert Read, Leo McElroy, Quentin Bolsee, B. Smith, and Neil Gershenfeld. 2023. Modular-Things: Plug-and-Play with Virtualized Hardware. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23), April 23–28, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3544549.3585642

## 1 INTRODUCTION

The design and assembly of physical computing systems is challenging. Creators have to navigate a broad set of skills that cut through disciplines in design, mechanical, electrical, and software engineering. Consequently HCI researchers and developers have created a collection of physical computing and electronic prototyping systems that aim to reduce complexity, increase speed of integration, and lower barriers to entry [12] [15]. Platforms like Arduino aimed to do this by making embedded programming easier with an abstraction layer across systems that allowed users to write portable embedded code.

This has been highly successful in making embedded programming more accessible, and has created ecosystems of breadboards, breakout boards, and libraries. But there remains room for improvement in the development process of physical computing systems. We see three main issues with the single microcontroller compiled firmware approach:

(1) Compiling and flashing firmware limits the interactive potential of the development process, because it inserts a delay between writing and testing code.
(2) Single-microcontroller projects lead to the creation of monolithic systems where individual functions are tightly coupled across design, mechanical, electrical, and software layers.
(3) Monolithic systems can present bottlenecks when a single microcontroller's available physical resources are used up (i.e. GPIO pins are all occupied or I2C address collisions occur).

One emerging approach to remediating the first issue is to embed high-level language interpreters into devices to create interactive development environments that run on microcontrollers [19]. This technique is seen in MicroPython on the Raspberry Pi Pico [7]. We implement an alternative approach to creating responsive development tools. Instead of trying to put high-level programming into devices, we lift devices into high-level programming environments by way of *virtualization*.

Virtualization (akin to the paradigm of Object Oriented Hardware, [16]) means each hardware module is represented to an application programmer as a unique software object that can be manipulated in a high-level programming environment (in our case JavaScript). Because this approach allows functionality to be broken out at a device level, monolithic systems ordinarily composed of one MCU with many peripherals can be recomposed as heterogeneous systems of many devices each with dedicated functionality.

In this paper we present our approach to improved prototyping of cyber-physical systems with virtualized modules, Modular-Things. Modular-Things consists of a web-based programming environment that we explain in Section 3.4, a small message passing library that supports a variety of link layers [10] and topologies (Section 3.6), and a collection of single purpose boards (or "things") which can be easily extended by users (Sections 3.1 and 3.3). In Section 4 we present a limited user trial, and in Section 5 we describe some limitations of our approach and plans for future improvements.

## 2 RELATED WORK

Our work builds upon developments in accessible physical computing systems [12] and extensible hardware construction kits [17] [8]. Over the last decade the most popular accessible physical computing tools have included Arduino [3], micro:bit [1], mini-computers from Raspberry Pi [20], and more recently the Raspberry Pi Pico [7]. These tools are all based on singular development boards, where users compile firmware for a single MCU and attach peripherals through exposed GPIO pins.

Some kits are designed with modularization in mind as is the case with SEEED Studio's Grove [5], Adafruit's STEMMA, Sparkfun's Qwiic [6] and LittleBits [4]. These modularized circuits treat the entire kit as a library of functionalities

embodied in each circuit which can be composed to build complete systems. This composition occurs by connecting circuits together in a consistent manner. Sadler *et al.* [18] emphasize how this interfacing step must be a one step process.

In order to simplify the interfacing process, construction kit developers use intra-board communication systems. Notably with Grove, STEMMA, and Qwiic I2C is used for this purpose. I2C is an on-circuit bus protocol for talking to a large number of devices, but most commonly sensors [9]. One disadvantage of I2C is that devices require unique addresses, which limits the discoverability of the system because users must know addresses beforehand.

Recently Microsoft Research developed a modular system similar to our own, Jacdac [11] [2]. Jacdac also offers an extensible physical computing system. We share similar design goals of creating easy hardware composition, plug-and-play software abstractions, and low cost systems. Superficially our work varies by opting to use USB as the communication link layer rather than a custom communication protocol and connector. In our system however USB is only one choice of embodiment among available link layers which the underlying networking system supports. We further differentiate ourselves from Jacdac by demonstrating the ease of developing new modules. In workshops new users created new modules by rapidly fabricating boards, hacking existing boards, and using our provided breadboard Modular-Thing. Figure 2 shows three such ad-hoc boards that were included into the Modular-Things system by their developers.

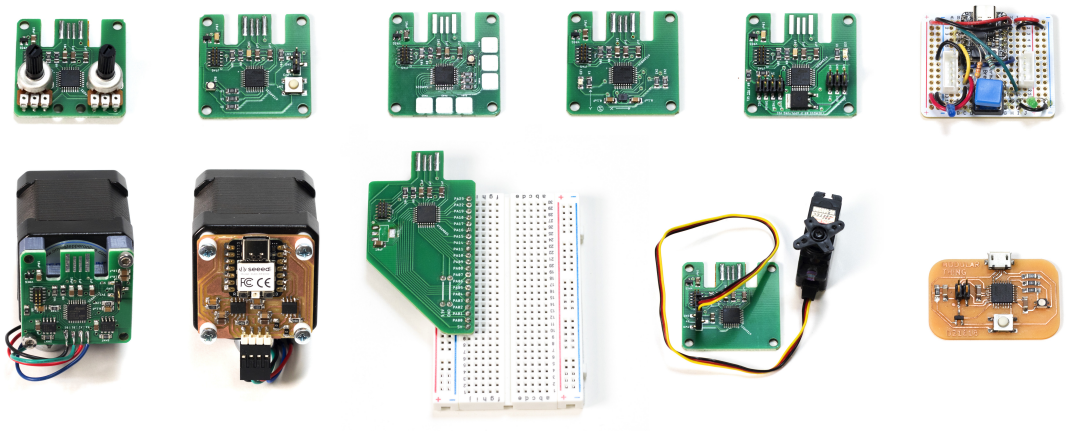

Fig. 2. Our circuit kit, including boards designed by beginners and generic development boards. Each embeds a USB-capable microcontroller for easy interfacing with a host computer. Top row left to right: potentiometer, RGB LED and button, capacitive touch sensor, accelerometer/gyroscope, DC load driver, LED/button/I2C/Serial Port homemade. Bottom row left to right: SAMD21 H-bridge stepper motor driver, Xiao RP2040 H-bridge stepper motor driver, breadboard, servo, homemade CNC milled LED/button.

## 3 DESIGN GOALS AND IMPLEMENTATION

The Modular-Things framework comprises three main elements. (1) An Arduino library that allows embedded programmers to rapidly turn any Arduino project into a new Modular-Thing. This library provides naming, routing and discoverability layers that allows modules to be found on simple or complex network topologies, and which enables inter-device message passing. (2) A web-based interactive development environment that presents available devices to user-programmers, and allows them to quickly write new programs with those modules. (3) A set of purpose-built Modular-Things (circuits and firmwares) that can be easily extended.

Collectively these elements allow developers of physical computing systems to rapidly assemble modules of hardware into new systems using a high-level programming language. In this section, we go into more detail on each aspect.

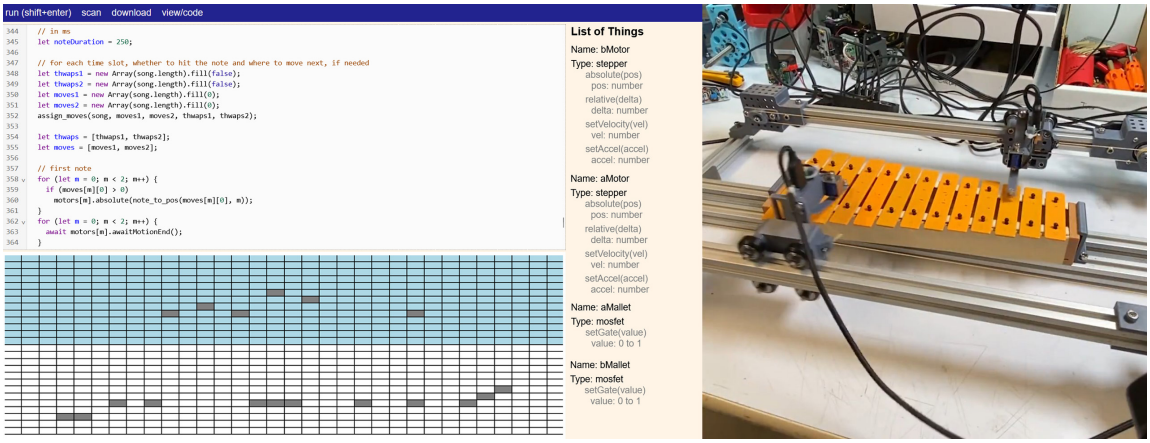


Fig. 3. User interface for editing the project's JavaScript code (upper left), with a dynamic listing of connected modules (middle) along with their APIs. An optional HTML view can be defined by the user for making custom interfaces (bottom left).

### 3.1 High-Level Application Programming for Embedded Devices via Virtualization

There are a number of reasons why development in high-level languages is seen to be preferable to development in the low-level languages used in microcontrollers. (1) High-level languages are easier to learn. (2) The delay between writing and testing code is brief and interactive. (3) There are rich debugging tools in high-level languages. (4) There is a broad set of available libraries and packages in high level languages (i.e. the commons of PIP and NPM).

Only two of these points (1, 2), actually pertain to the language itself. The rich debugging tools and availability of libraries both depend on the context in which those languages are deployed. For example much of the value in JavaScript is in the browser and its litany of tools. Much of the value of Python is in the availability of PIP packages.

In order to leverage the full value of high-level programming, our system opts to adopt a strategy of **virtualization**. In this strategy function-specific firmwares are built ahead of time into composable modules, and those modules are remotely operated in a high-level language (JavaScript) which is running in a high-level context (the browser). *Rather than "embedding" a high-level language into hardware, we are "lifting" hardware modules into a high-level language.* This lift allows us to shift application-specific code out of hardware systems and into a friendlier programming environment. It also affords a straightforward way of composing multiple modules, by maintaining that each unique module is addressable as a unique software object.

### 3.2 User Workflow

With Modular-Things, users can assemble new physical computing systems in a plug-and-play manner, such as the example machine illustrated in Figure 1. When users plug devices into a network, they are automatically discovered, and their unique name, along with their API, is presented to the programmer as a virtualized software object. The device can immediately be used by calling the functions presented in its API. Since multiple things can be plugged in to the same network, composition of systems is just as simple as building a program that uses multiple software objects.

To the programmer, the only substantive difference is that these software objects are remotely operating real hardware modules.

### 3.3 Rapidly Virtualizing Embedded Codes

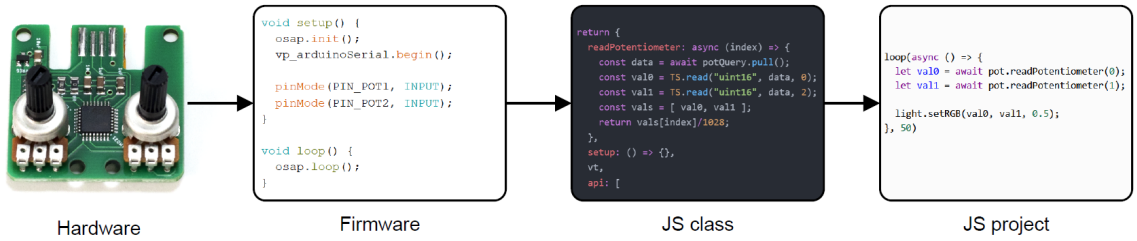


Fig. 4. Integrating a new embedded device as a Modular-Thing means defining an embedded API, and describing that API in a matching JavaScript file. Modules extend the Arduino framework, and the programming burden can be as small as 50 lines in total for simple modules.

Using existing modules makes it possible to build new physical computing systems without engaging in any embedded programming or device design. However, creating new modules is also simple, as shown in Figure 4. First, device authors write firmwares that operate their module, and design an API. They they install OSAP (a library which is explained in Section 3.6), which they can use to name their device and instantiate a link layer that allows the device to connect to the Modular-Things network. Next, authors write message handlers in their firmware that operate their API, and write a matching file in JavaScript that serves as an intermediary between the high-level API and their device firmware. At runtime, the JavaScript object is dynamically paired with a data channel to the device, and the device is then virtualized.

### 3.4 Aligning Code and Reality

A primary goal of our system design was to automatically reconcile misalignments between the programmers' code and the physical reality of the system. Generally, embedded devices must know ahead of time what type of messages to expect and high-level programs must know ahead of time what the devices are and where they will be located. We address this issue by dynamically presenting a list of available devices, which each maintain individual device identities with in-device non-volatile names. This list is seen in the side panel of the web IDE in Figure 3. This allows users to easily switch between writing code, adding/removing devices, and testing devices.

### 3.5 Built-in UI Development

Physical computing systems often require the integration of user interfaces. This was a strong factor in our choice of the browser as a high-level computing environment. In order to rapidly combine controllers with UIs, our web IDE allows programmers to switch from a programming view to a rendered view, as diagrammed in Figure 3. UI-defining JavaScript simply lives alongside machine controller code.

### 3.6 Discoverability, Scalability and Extensibility

Many prototyping tools for physical computing systems are characterized by the link layer across which they operate; for example STEMMA and Qwiic systems are fundamentally based on I2C interconnect and Jacdac is characterized by

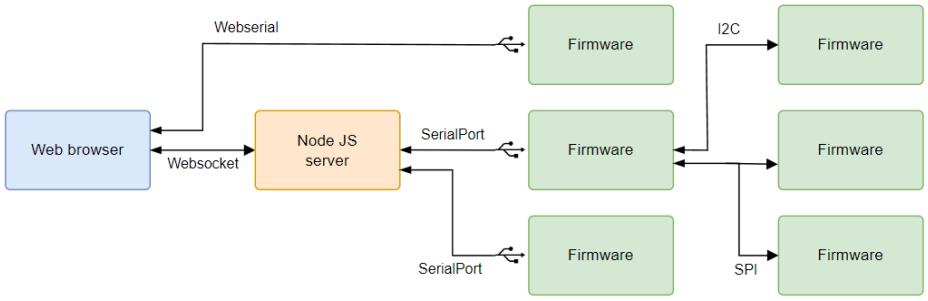


Fig. 5. Modular-Things can be attached to one another using a wide range of link layers, and network topologies using a message-passing, source-routed scheme.

the Jacdac bus. While we primarily developed USB-based devices, we wanted to enable Modular-Things to be discovered across any potential link layers, even those that have not yet been designed. Moreover, we wanted to be able to combine link layers with one another to form ad-hoc networks. To do so, we implemented a small interconnect system called "OSAP" (Open Systems Assembly Protocol) that provides naming, routing, and discoverability services to our application. A complete description of this layer is outside of the scope of this paper, but we provide a summary of its operation here.

*3.6.1 Packetized, Source-Routed Message Passing.* OSAP's primary function is to route messages throughout graphs of various topology. To do so, it uses source routes that are embedded within the packet header itself. This is important because each OSAP device is also an OSAP router, and source routing means that routers can remain simple and stateless even in the context of messy graphs.

*3.6.2 Link-Layer Agnostic Graphs.* OSAP's own runtime knows nothing about the link layers it uses to form a graph: it merely differentiates between one-to-one and one-to-many links (ports and busses, respectively). To attach OSAP devices to one another, link layers are interfaced to the OSAP runtime through a simple API that hands off outgoing messages and uses a callback for incoming messages. This means that almost any packetized transmission device can be integrated with OSAP: to date we have used WebSockets, USB Serial, UART and UART-based busses.

Because OSAP devices are also routers, building an OSAP bridge layer (between a WebSocket and USB Serial Ports - for example), can be done by attaching multiple link layers to the same OSAP runtime. We show a diagram of the types of device graphs that are possible using this scheme in Figure 5.

*3.6.3 Graph Discovery.* OSAP contains affordances for automated graph traversal, in the form of a special packet that queries a device's neighbourhood for active links. Devices can also respond to queries about their name and their contents. Using these packets, our web IDE can build a map of the device graph, and use that map to identify new devices and pair them with their JavaScript APIs.

### 3.7 Motion Controllers

We found it was possible to implement synchronized motion without motors being directly connected to one another. We did so using a pure software object called a "synchronizer" that operates multiple motor modules simultaneously. This was enabled by a performant transport layer, that delivers packets to up to seven devices within 5*ms* of one another

in worst case measurements, and most often within 500*us*. Because packet arrival is nearly synchronous, and the motor controllers are sophisticated enough to run entire segments of motion, it is feasible to remotely operate a group of motors as if it were one cohesive machine.

This strategy is akin to `virtual machines` [14], though our synchronization strategy is less sophisticated, and our modularization strategy is more extensible. Because these systems are modular by nature, users can assemble new machine systems that have wide and varying numbers of peripheral devices.

## 4 APPLICATION, USE AND EVALUATION

We did not conduct a full user study, though we did deploy Modular-Things in a classroom setting and during a five day workshop with instructors in digital fabrication. Approximately 60 students were broken into groups of 15 and tasked with building complete machines (including mechanism, actuation, automation and application) over the course of one week. Each group designed and constructed a unique machine with custom controls and high-level graphical interfaces. Some of these machines can be seen in Figure 6, and included a sand art drawing machine, a pancake plotter, a CoreXY[13] pen plotter, and an *auto-aiming toilet paper throwing machine* that integrated computer vision libraries to recognize faces and auto-aim the machine towards them. This level of system integration was made possible through Modular-Things because motor control was available in the same computing environment as the facial recognition software. Some students found Modular-Things intuitive to use and elected to reuse it for their final projects in the class. One of these students (a first time machine builder) developed a machine for photopolymerization of photo-responsive liquid crystal elastomers by direct laser writing. We observed users were able to easily replicate the work of others by copying code, plugging in the required modules, and renaming devices to match the naming scheme of their example snippets. Users did this to rapidly recreate motion systems which would have been difficult to recreate at a firmware level.

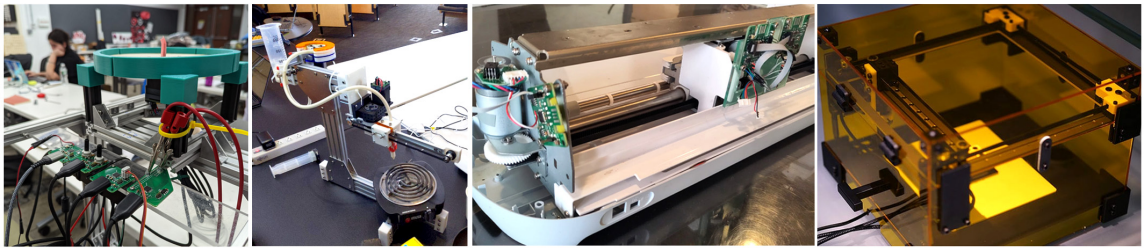

Fig. 6. Machines made or modified by students as part of the digital fabrication class. Modular-Things were used to develop these machines' control systems, including improvised devices with custom firmware and APIs.

Successful development of these machines demonstrated it was possible for novice users of Modular-Things to create non-trivial integrated physical computing systems, with demanding coordination among devices. Additionally it was possible for users to leverage existing knowledge of high-level technologies (i.e. HTML/CSS/JS stack and computer vision libraries) to create responsive interfaces for these machines which would generally be difficult to integrate with low-level devices.

We found that during this machine building week and our five day workshop many testers wanted to develop their own custom devices which could be integrated with Modular-Things. This was demonstrated by users constructing custom electronics from our "breadboard-thing", hacking DC motors with encoders to be used with "stepper-things",

and fabricating their own "things" by writing custom firmware with the networking library, one of which is pictured in Figure 2.

## 5 LIMITATIONS AND FUTURE WORK

The version of Modular-Things presented here has a few limitations. Notably, most single purpose devices we developed relied on USB for communication with the high-level programming environment, which can be expensive and unwieldy as systems scale. Besides including lower-level UART-based link layers that we have already developed in our basic set of boards, we plan to include lightweight link layers for I2C, SPI, and wireless links like BLE and LoRA.

It could be seen as burdensome that each module requires its own microcontroller and data link. Currently these costs are on the order of $5 USD per module; we suggest that they are outweighed by the value provided by virtualization in prototyping contexts.

Another limitation is that systems are tethered to high-level computing devices in which they are programmed, normally user laptops. We also plan to develop design patterns for stand-alone systems that replace the high-level desktop computer with an embedded device which can exist within the system, or simple methods for the deployment of Modular-Things onto smaller, stand-alone computers that can be permanently integrated with projects.

Operating modular devices over a network also presents timing challenges. Although our system can provide order-of-operation guarantees (using asynchronous programming patterns in JavaScript), function calls to remote modules can take up to 10ms to complete. This means that user-created real-time *feedback* applications are not possible, although *feed-forward* applications (like most machine controllers) can be implemented (as demonstrated). Our approach relies on module authors to implement tightly timed system aspects (like motor controllers) in their firmwares, that can be combined at a high level by user-programmers. It seems likely that the incorporation of lower-level link layers and deployment on i.e. single-board-computers may improve this performance bottleneck, but much future work lays in this direction.

## 6 CONCLUSION

In this work we presented Modular-Things. It consists of a collection of single purpose boards and firmwares, but more importantly, a toolkit for building extensible virtualized physical computing systems. The toolkit consists of a networking and discoverability layer to connect modules together, a library for the authorship of new modules, and a Web IDE for their integration. In the future we plan to develop a wider variety of link layers, and methods to allow Modular-Things to operate independently of the high-level computing environments in which they are developed and configured.

With our system users were able to rapidly create machines with rich features and interfaces without writing embedded firmwares. This demonstrated the viability of using virtualized hardware in discoverable interactive development environments to prototype cyber-physical systems. Importantly, we found that providing the tools to develop new Modular-Things allowed users to integrate their own virtualized devices, which greatly extended the capabilities of the overall system.

## REFERENCES

[1] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli De Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale. 2020. The BBC micro: bit: from the UK to the world. *Commun. ACM* 63, 3 (2020), 62–69.

[2] Thomas Ball, Peli de Halleux, James Devine, Steve Hodges, and Michal Moskal. 2023. *Jacdac: Service-based Prototyping of Embedded Systems.* Technical Report MSR-TR-2023-4. Microsoft. https://www.microsoft.com/en-us/research/publication/jacdac-service-based-prototyping-of-embedded-

systems/

[3] Massimo Banzi and Michael Shiloh. 2022. *Getting started with Arduino.* Maker Media, Inc.

[4] Ayah Bdeir. 2009. Electronics as material: littleBits. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*. 397–400.

[5] Charles Bell. 2021. Introducing Grove. In *Beginning IoT Projects*. Springer, 481–509.

[6] Charles Bell. 2021. Introducing Qwiic and STEMMA QT. In *Beginning IoT Projects*. Springer, 217–258.

[7] Charles Bell. 2022. Introducing the Raspberry Pi Pico. In *Beginning MicroPython with the Raspberry Pi Pico*. Springer, 1–42.

[8] Paulo Blikstein. 2013. Gears of our childhood: constructionist toolkits, robotics, and physical computing, past and future. In *Proceedings of the 12th international conference on interaction design and children*. 173–182.

[9] Peter Corcoran. 2013. Two wires and 30 years: A tribute and introductory tutorial to the I2C two-wire bus. *IEEE Consumer Electronics Magazine* 2, 3 (2013), 30–36.

[10] John D Day and Hubert Zimmermann. 1983. The OSI reference model. *Proc. IEEE* 71, 12 (1983), 1334–1340.

[11] James Devine, Michal Moskal, Peli de Halleux, Thomas Ball, Steve Hodges, Gabriele D'Amone, David Gakure, Joe Finney, Lorraine Underwood, Kobi Hartley, et al. 2022. Plug-and-play physical computing with Jacdac. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 3 (2022), 1–30.

[12] Mannu Lambrichts, Raf Ramakers, Steve Hodges, Sven Coppers, and James Devine. 2021. A Survey and Taxonomy of Electronics Toolkits for Interactive and Ubiquitous Device Prototyping. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 5, 2 (2021), 1–24.

[13] IE Moyer. 2012. Core xy.

[14] Ilan Ellison Moyer. 2013. *A gestalt framework for virtual machine control of automated tools.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[15] Nadya Peek, James Coleman, Ilan Moyer, and Neil Gershenfeld. 2017. Cardboard machine kit: Modules for the rapid prototyping of rapid prototyping machines. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 3657–3668.

[16] Nadya Nadya Meile Peek. 2016. *Making machines that make: object-oriented hardware meets object-oriented software.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[17] Mitchel Resnick and Brian Silverman. 2005. Some reflections on designing construction kits for kids. In *Proceedings of the 2005 conference on Interaction design and children*. 117–122.

[18] Joel Sadler, Kevin Durfee, Lauren Shluzas, and Paulo Blikstein. 2015. Bloctopus: A novice modular sensor system for playful prototyping. In *Proceedings of the ninth international conference on tangible, embedded, and embodied interaction*. 347–354.

[19] Nicholas H Tollervey. 2017. *Programming with MicroPython: embedded programming with microcontrollers and Python.* " O'Reilly Media, Inc.".

[20] Eben Upton and Gareth Halfacree. 2014. *Raspberry Pi user guide.* John Wiley & Sons.

Received 19 January 2023