fabryq: Using Phones as Gateways to Prototype Internet of Things Applications using Web Scripting

Will McGrath[1,3] **Mozziyar Etemadi**[1,2] **Shuvo Roy**[2] **Bjoern Hartmann**[1]

[1]UC Berkeley
EECS
{mozzi,bjoern}@berkeley.edu

[2]UCSF Bioengineering
and Therapeutic Sciences
shuvo.roy@ucsf.edu

[3]Stanford University
Computer Science Department
wmcgrath@stanford.edu

ABSTRACT

Ubiquitous computing devices are often size- and power-constrained, which prevents them from directly connecting to the Internet. An increasingly common pattern is therefore to interpose a smart phone as a network gateway, and to deliver GUIs for such devices. Implementing the pipeline from embedded device through a phone application to the Internet requires a complex and disjoint set of languages and APIs. We present fabryq, a platform that simplifies the prototyping and deployment of such applications. fabryq uses smartphones as bridges that connect devices using the short range wireless technology, Bluetooth Low Energy (BLE), to the Internet. Developers only write code in one language (Javascript) and one location (a server) to communicate with their device. We introduce a *protocol proxy* programming model to control remote devices; and a capability-based hardware abstraction approach that supports scaling from a single prototype device to a deployment of multiple devices. To illustrate the utility of our platform, we show example applications implemented by authors and users, and describe µfabryq, a BLE prototyping API similar to Arduino, built with fabryq.

Author Keywords

Toolkits; ubiquitous computing; swarm devices; prototyping.

ACM Classification Keywords

H.5.2: Prototyping

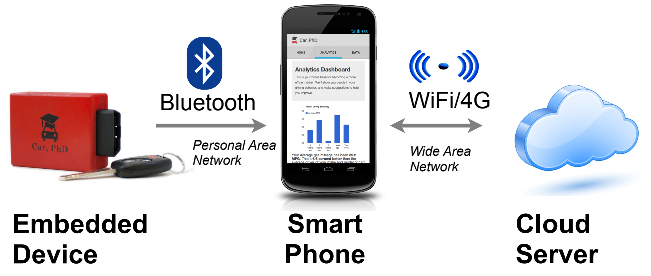


Figure 1. An example MGC application distributes logic and user interaction across embedded device, mobile phone and a cloud server.

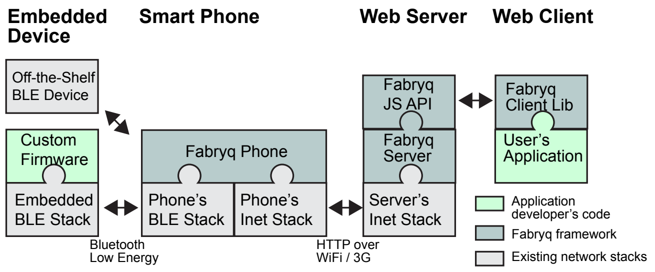


Figure 2. With fabryq, developers use off-the-shelf bluetooth devices or write firmware that exposes devices' inputs and outputs over Bluetooth; and high-level application logic in Javascript. The fabryq framework manages finding the desired hardware device and a mobile phone within range, and handles all message marshaling.

INTRODUCTION

In the predominant vision of ubiquitous computing (or its reincarnation as the internet of things or swarm computing), all kinds of devices, from large to small, become smart and networked [26, 23]. One important class of ubiquitous computing devices are small, wireless, low-power sensors — for example those used in medical, fitness, and distributed sensing applications. Many of these devices cannot be directly connected to the Internet (e.g., via WiFi) because of size and power constraints. In practice, therefore, ubicomp devices are often constructed using a three-level architecture consisting of: 1) a very energy efficient, embedded low power device with a short range radio; 2) a gateway, such as a user's phone, which may show a user interface and acts as a router from short-range networks to the Internet; 3) server code for aggregating data and reasoning across multiple users and devices. We refer to such applications as *MGC* (eMbedded–Gateway–Cloud) apps (see Figure 1). For example, the FitBit fitness tracking device monitors a user's motions. On demand, it relays information to a companion application running on the user's mobile phone (or PC), which in turn communicates with servers that the FitBit company maintains. Building and maintaining such multi-language, multi-platform distributed systems is complex, error-prone, and requires skills in several diverse fields. Thus, experimentation in such devices is largely reserved to teams of experts, and implementation cycles are long and complex, which prohibits rapid prototyping. While research has introduced prototyping toolkits that significantly increase the speed of design explorations [8, 7, 25],

EICS'15, June 23-26, 2015, Duisburg, Germany
ACM 978-1-4503-3646-8/15/06.
http://dx.doi.org/10.1145/2774225.2774835

these toolkits often make power or connectivity tradeoffs that restrict their use to lab settings or stationary, plug-in products.

In response to these issues, we introduce fabryq, a framework that facilitates the creation of new Ubicomp devices by handling the complexities of creating new mobile device, server, and networking code. Specifically, fabryq takes the form of a mobile application and cloud service. The mobile application turns an ordinary smartphone into a bridge that connects the short range wireless technology of Bluetooth Low Energy (BLE) with our cloud service via the Internet. fabryq uses BLE because it is the single short range wireless technology that is ubiquitously available on modern smart phones and can thus be widely employed. fabryq applications are written in Javascript and run in a web browser. fabryq introduces a *protocol proxy* programming model — developers write BLE protocol calls in Javascript as if the target device were locally connected and always available. The fabryq architecture then finds a mobile phone within radio reach of the target BLE device; passes the command(s) through the phone to the target device, and returns data to a the web application. This allows the creators of new devices to focus on writing the devices' firmware and creating new applications with the data from the devices, rather than writing complex and error-prone networking code (see Figure 2).

Our approach deals gracefully with situations where a BLE device may move into and out of the range of a mobile client device. When a device is not available, commands are queued and will be executed in the order that they were issued whenever the device comes back into range. BLE's star topology ensures that a device can only connect to a single gateway at once. fabryq handles the connection management and networking that make this abstraction possible.

fabryq applications can be run in a browser, on the fabryq iOS app, or on a server using our agent interface; but in all cases, it is the same application code using the same API that allows for issuing BLE commands. User interfaces for fabryq applications such as data visualizations are authored using HTML and JavaScript, and they can be viewed either on the web, or inside the fabryq application on a mobile device. We define two application types: *sporadic interaction*, where communication only occurs when a user views a Fabryq application on their phone; and *continuous monitoring*, where data is collected and stored even if a user is not interacting with a user interface. For sporadic interactions, users execute fabryq web applications directly on their phone. For continuous monitoring, fabryq supports the creation of continuously running, remotely executed Web applications ("agents"). To enable showing user interfaces on phones in this scenario, fabryq includes a *UI pushdown* utility command so that data updates on the server can trigger the display of interfaces on the phone that was responsible for collecting the data.

A key aspect our our approach is promoting the gateway from a "dumb router" to a key point of user interaction. When running on a user's smartphone, the most revelvant devices, those nearby the user are always made available to applications. The ability of the gateways to display rich user interfaces also allows for a flexible way of displaying information or options to a user. Additionally, the gateway's configuration UI provides a concrete way of communicating system status and managing connected devices.

Assessing the success of ubicomp prototypes often requires leaving the lab and deploying devices with users "in the wild" [5]. To facilitate expanding from a single prototype to a multi-device deployment, fabryq keeps track of what kinds of devices are necessary for running a given application. Based on a set of hardware service requirements that are automatically generated for each new application, developers can create additional *instances* of an application. Each instance can then be bound by users to different hardware devices that offer the requisite functionality, as encoded in BLE services and characteristics. This makes it possible, for example, to let users use heart rate monitors from different manufacturers.

To demonstrate the utility of fabryq for developing applications with custom BLE devices, we also created μfabryq, a custom BLE device paired with a JavaScript API that make some of the most useful features of embedded processors such as analog to digital converters (ADCs), interrupts, digital input/output pins (GPIO), pulse width modulation (PWM), and a serial peripheral interface (SPI) available to web programmers via fabryq. This firmware mirrors the programming model of popular "maker" platforms such as Arduino, but offers the benefit that applications can be distributed across many wearable devices in many locations. μfabryq was entirely implemented on top of fabryq.

fabryq makes it easier for developers to work with both existing off the shelf BLE devices and custom devices of their own design. In order to both validate fabryq and demonstrate its utility, we describe applications created by students using μfabryq during hackathons and in class, as well as applications created by the authors. These applications make use of a combination of off-the-shelf and custom BLE devices.

RELATED WORK

Ubicomp Prototyping

HCI research has contributed systems for rapid prototyping of Ubicomp devices and systems, e.g., Phidgets [7], d.tools [8], LilyPad Arduino [4], which focus exclusively on simplifying the creation of individual, self-contained devices. Other systems such as .NET Gadgeteer [25] and Shared Phidgets [18] explicitly offer network connectivity to create Internet-connected devices, but assume end-to-end IP connectivity and tethered operation, and thus cannot easily "scale down" to support low energy mobile devices; they generally also do not support distributing interactions over device, smartphone, and servers. Recently, commerical platforms such as the ElectricImp WiFi module [10], Spark Core [21], Tessel [24], and Kinoma Create [15] aim to lower the threshold for developing Internet-connected appliances. Like fabryq, the Imp and Spark Core use a hosted server that handles many lower-level networking tasks. However, both devices require direct WiFi access—power requirements make it infeasible to use them for mobile, wearable deployments. For comparison, when active, the Spark Core has a minimum power consumption of 30mA and a maximum of 300mA

[22]. Conversely, Texas Instruments' CC2541 BLE-enabled microcontroller, consumes 8.5mA [12] when active. Although peak transmission power is only one variable that contributes to total power consumption and power is not always a primary concern while prototyping, BLE's adjustable connection parameters allow flexibility and further optimization as a design is further developed.

Connecting Sensors to Mobile Phones

A number of projects aim to make it easier to connect external devices and sensors to smart phones and use them in applications. iStuffMobile [2] augments existing phones with new sensors – for development speed, sensor mapping logic runs on a nearby desktop computer, limiting deployment options. Amarino [13] allows designers to access events occurring on a mobile phone from an embedded platform. HiJack [16] can power and exchange messages with an embedded microcontroller through a phone's audio jack. Open Data Kit Sensors [6] is an application framework that introduces abstractions to simplify the connection of multiple sensors with different interfaces (e.g., wired and Bluetooth) to a single mobile device. Dandelion [17] generates both smart phone and sensor node binaries from a common source and then uses remote method invocation to call sensor code from the phone.

In contrast to these projects, fabryq uses the phone as a smart gateway to relay commands from a Web server to BLE devices. This allows developers to change their sensor querying code at any time without having physical access to the phone, and applications can span multiple phones. fabryq is also agnostic to which phone is connected to which sensor—only communication between the server and sensors matters.

Cooltown [14] focused on bringing physical devices onto the Web by embedding web servers into products and using mechanisms such as IR for finding a physical objects URL. In contrast, fabryq supports using hardware that *cannot* connect to web services or speak TCP/IP directly because of memory, compute, or battery limitations. This is an important category comprising a quickly growing array of personal devices.

Internet of Things Networking

Connecting resource-constrained embedded hardware to Internet servers is also a concern of "Internet of Things" researchers. One way to provide IP packet support to low-energy embedded devices is through IEEE802.15.4 networks using "6LoWPAN" (IPv6 over Low Power Wireless Personal Area Networks [1]). Alternatively, devices such as the XBee Internet Gateway marshall traffic between a local area network and the Internet. The main difference to much of this work is that it presupposes additional networking infrastructure which is not generally available yet. We instead target ubiquitous smart phones and their data networks. Our focus is on supporting mobile scenarios that can be deployed with end users today; we therefore focus on BLE, which in itself does not currently interoperate with IP networks.

MOTIVATING APPLICATIONS

To inform the design of fabryq, we surveyed the emerging market of wireless smartphone acessories. Fitness trackers

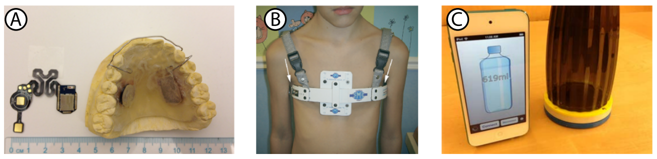


Figure 3. Three motivating wireless medical devices: A) A retainer that tracks wearer compliance using a built-in temperature sensor. B) A chest compression brace that tracks applied pressure over time. C) A water mug that tracks liquid consumption throughout the day.

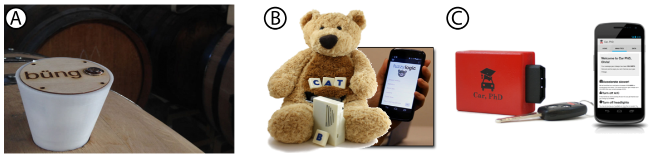


Figure 4. Example devices from our class: A) A wireless barrel gauge for distilleries and wineries; barrel readings are aggregated online. B) A toy to support literacy education; play statistics are collected on the phone and online. C) A car dongle that streams driving telemetry data to the phone and compares driving behavior on a central server.

such as the FitBit or Misfit Shine as well as devices such as the Automatic vehicle data link tend to follow a similar pattern: they use MEMS sensors and minimal display on device and a smart phone to display the main UI to the user. They also use web backends to store, process, or share data.

Our work is also motivated by a collaboration with medical researchers at a local medical center and their needs for wearable patient monitoring devices (see Figure 3). These devices generally require small physical size and weight, but battery life on the order of weeks or months so they can be given to patients without requiring recharging. Researchers want to send patients home with these devices and remotely track gathered sensor data at their institution. Such deployments consist of a few dozen identical devices. Accordingly, a framework should facilitate both continuous monitoring and simple deployment of prototype code to multiple users.

Finally, we draw inspiration from experiences gathered teaching the design of integrated interactive hardware/software devices at our institution. While teams of motivated undergraduate and graduate students can create working prototype devices such as the ones shown in Figure 4, much of the implementation difficulty lies in working with multiple different networking technologies and protocols simultaneously; managing intermittent wireless connections; and doing this in multiple different programming languages on different platforms (embedded, mobile, web) with different conventions, data encoding schemes, etc.

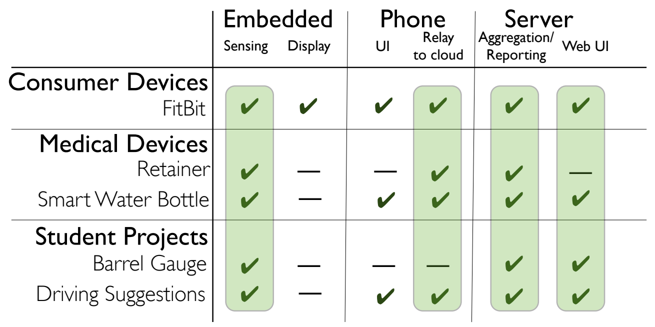
Design Guidelines

Common patterns in our device survey (Figure 5) yielded the following design guidelines:

Smartphone as gateway Leverage the smart phone as a proxy from the local, body-area network to the Internet.

Multiple devices Simplify development and management of multiple prototypes that can be distributed to users.

Display Enable developers to show collected data and other user interfaces both on the web and on a phone.



	Embedded		Phone		Server	
	Sensing	Display	UI	Relay to cloud	Aggregation/ Reporting	Web UI
Consumer Devices						
FitBit	✓	✓	✓	✓	✓	✓
Medical Devices						
Retainer	✓	—	—	✓	✓	—
Smart Water Bottle	✓	—	✓	✓	✓	✓
Student Projects						
Barrel Gauge	✓	—	—	—	✓	✓
Driving Suggestions	✓	—	✓	✓	✓	✓

Figure 5. Features of some motivating examples. fabryq focuses on supporting embedded sensing, relaying data through a phone, and aggregating and displaying info on the Web (green columns).

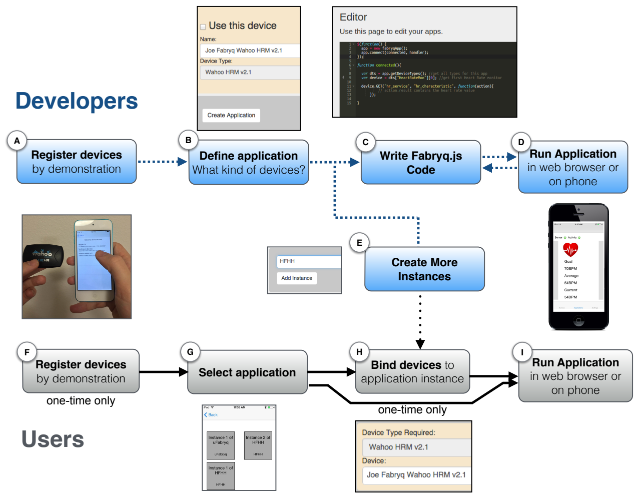

Developers
Use this device
Name:
Joe Fabryq Wahoo HRM v2.1
Device Type:
Wahoo HRM v2.1
Create Application
Editor
Use this page to edit your apps.
A Register devices by demonstration
B Define application What kind of devices?
C Write Fabryq.js Code
D Run Application in web browser or on phone
E Create More Instances
HFHH
Add Instance
F Register devices by demonstration
one-time only
G Select application
H Bind devices to application instance
one-time only
I Run Application in web browser or on phone
Users
Device Type Required:
Wahoo HRM v2.1
Device:
Joe Fabryq Wahoo HRM v2.1

Figure 6. An overview of the process used to define and run a fabryq application.

Abstract networking details Shield developer from network connection management and data transfer details.

Lower threshold, high ceiling Enable users with limited programming experience to write simple scripts in a single language; enable experts to create new, complex devices.

Flexible software Enable developers to easily change application code during deployment without having physical access to phones or sensors.

Conversely we chose to avoid supporting the following cases:

No low-latency, high-throughput apps We focus on working with intermittently read sensor data where milliseconds of latency are not important.

No offline operation without a cloud server We target prototype deployment where the experimenters are in the loop; we do not target the scenario of BLE devices talking to phone applications without code running in the cloud.

CREATING AND DEPLOYING FABRYQ APPLICATIONS

BLE fundamentals

BLE is a wireless protocol for communication between a *central device* (i.e., a mobile phone) and one or more *peripheral devices*. Peripheral devices expose *BLE characteristics* — short, named pieces of information (typically 1-20 bytes) similar to variables, which are organized in a Generic ATTribute profile (GATT) table and identified by unique hex UUIDs. Central devices can perform three operations on characteristics: `GET`, `SET`, and `NOTIFY`. A `GET` signifies that the BLE central device would like to retrieve the contents of a characteristic from the BLE peripheral. A `SET` means that the BLE central would like to modify the contents of a characteristic. A `NOTIFY` signifies that the central would like to be notified if the value of a characteristic changes. Operations happen at a set *connection interval*, a time window when the central and peripheral have decided to communicate. By adjusting the interval, battery life of more than one year can be achieved from a coin cell battery: one of the hallmarks of BLE. A more detailed description of BLE can be found in [9, 3].

Defining the first prototype

A key design goal of fabryq is to make it fast to develop an initial prototype, while making it possible to later scale to multiple users and multiple devices. We introduce a running example to illustrate how to write and deploy such applications. Refer to the video figure for a demonstration of the creation of the example application. For example, a hypothetical *HydrateForHeartHealth* (HFHH) application may track a user's liquid intake over the course of a day and correlate it with heart rate variability. It could require information about liquid level in a smart cup (a custom device also created by the developer) and data from a heart rate monitor (an off-the-shelf device). Developers then write application code (using the fabryq JavaScript API) that references this hardware configuration. To deploy or test an application, the configuration must be linked to particular devices (a specific cup and a specific heartrate monitor).

Demonstration of devices: Our HFHH developer begins the development process by registering her devices in the fabryq mobile application. We have developed a demonstration-based workflow that allows developers to efficiently define abstract device requirements of their application through physical demonstration, without restricting the application to later rely on these particular physical devices. On her phone, she intiates a scan of nearby BLE devices and selects the discovered entries for her cup and heart rate monitor (Figure 6A). fabryq then traverses these devices' GATT tables and either creates new device type entries for them on the server and phone, or recognizes them as instances of previously created devices types. The developer can then create their application by selecting any number of their devices and naming the new application (Figure 6B).

Abstraction: Fabryq separates device types, which are used to set and check requirements for running applications and concrete devices, which are entries corresponding to a single physical peripherial (Figure 7). Registering the devices with fabryq creates corresponding device types (shown in the center of Figure 7). Similarly, defining a new application automatically creates a new application type (top right of Figure 7, Figure 6B) that can be reused by many instances. The application defintion lists the numbers and kinds of devices required by the application as well as the code used for that

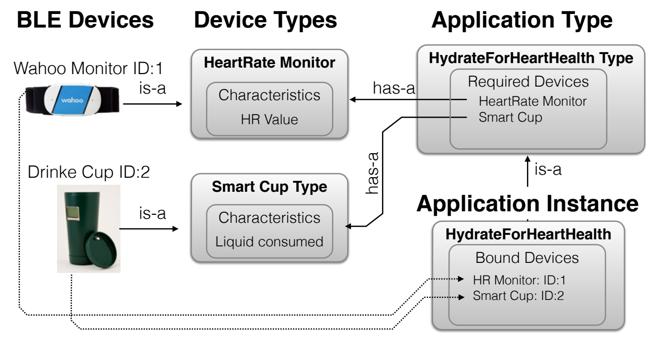


Figure 7. Overview of fabryq application configuration. Device types abstract specific BLE devices and their GATTs. Application types are defined by a list of device types they require. Application instances link specific real-world devices with their digital proxies.

application. In this case, the HFHH application definition is linked to the "HR Monitor" and the "Smart Cup" device types. The application instance represents a single installation of the application and links the application's code with the specific devices used by that application. Now that the application is configured, the developer can open an auto-generated boilerplate application up in the fabryq online code editor and begin programming.

Writing code: In general, fabryq programming consists primarily of issuing GET/SET/NOTIFY commands and writing callbacks to handle the returned data. Applications all run in a web browser — this means they can be opened either on a remote server or PC, or on a mobile device acting as a gateway. In this case, our developer would like to access the heart rate characteristic of her BLE heart rate monitor.In the code, the developer does not need to reference particular, physical device identities, but only their types (Figure 6C). When the application is run, the application's instance ID automatically links the code to the concrete devices used by the instance. This makes the code portable across many instances. In this example, she obtains the current user's heart rate using a `GET` command. Using fabryq's JavaScript API, this is one function call with embedded callback lambdas:

```
$(function() {
  app = new fabryqApp();
  app.connect(connected, handler);
});

function connected(){
  //get all types for this app
  var dts = app.getDeviceTypes();
  //get first Heart Rate monitor
  var device = dts["HeartRateMon"][0];

  device.GET("hr_service", "hr_characteristic",
    function(action){
    // action.result contains the heart rate value
    });
}
```

The `hr_service`, and the `hr_characteristic` constants are human-readable aliases for the hex BLE UUIDs. fabryq automatically translates these for supported UUIDs and accepts ordinary UUIDs as well. `SET` and `NOTIFY` commands can be used in a similar fashion.

Installing and running the application

A user of a fabryq-enabled web application performs three steps: first, they install the fabryq mobile application. Second, they register their devices, as a developer would (Figure 6F). Next, they select the fabryq applications they plan to use (Figure 6G) and choose which of their devices should be used for it Figure 6H). Finally, they launch the application in one of two ways:

Sporadic interaction is initiated by opening an application URL on the phone. Communication with BLE devices and the server only happens when a user is directly interacting with an application's user interface on their phone. In *continuous monitoring*, sensor data is collected even when the user is not engaged with the application. For sporadic interaction, the user launches their desired application from inside the fabryq mobile app on their phone (Figure 6G). For continuous monitoring, they open the equivalent URL on a PC or server, where the client browser can stay open continuously. By using fabryq's UI pushdown technique (described below), developers can still show user interfaces on a user's phone, even if the application itself is running in a different browser.

Supporting multiple devices and mobile gateways

To scale from a single prototype to a multi-user deployment, developers now benefit from the abstraction that fabryq has generated for them. They can define new application instances using a web form (Figure 6E). Each instance can then be run by a different user who can tie the instance to their own hardware devices on the first launch.

Our example considered a situation in which an application has one gateway and two BLE devices. However, in order to expand the number of scenarios in which fabryq can be used, we support the use of multiple gateways and shared gateways as well. In the simplest case, one application requires one peripheral and the user has one mobile gateway device that allows data to flow from application to peripheral (Figure 8A). One application may also allow the user to "roam" with their BLE peripheral, periodically coming into contact with one of multiple gateways devices, as in Figure 8B. In order to work around security restrictions (some OSes scramble the BLE MAC addresses they show to applications), the devices must each be registered once with each smart phone gateway the user wants to use. Finally, users may opt to share their iOS device's BLE radio with other users. In this configuration, users would be able to connect their peripherals to fabryq using other users' iOS devices. As a simple example, a cyclist's heart rate monitor could connect to his heart rate application while he cycles without an iOS device, provided he comes in contact with other users' devices. Depicted in Figure 8C, this "crowd sourced BLE internet access" is inspired by emerging applications such as Tile [1].

Sharing devices' connections of course has privacy implications. Fabryq provides some control: both devices and gateways default to *n*ot shared when they are first registered. Additionally, a device can only connect through a shared gate-

[1]http://www.thetileapp.com

way when both the device and gateway have been designated as shared by their respective owners.

FABRYQ IMPLEMENTATION

One main function of fabryq is analogous to a router: fabryq commands (`GET`, `SET`, `NOTIFY`) can be sent to the server by the JavaScript client API and routed to an appropriate BLE peripheral via a mobile device running the fabryq gateway app. fabryq can also show user interface on a mobile device connected to a BLE peripheral. The mobile gateway also displays system status, and is used for registering new devices.

fabryq mobile

In order to include a useful and flexbile BLE router as a part of the fabryq platform, we had to solve two technical challenges: inconsistent identification of a user's peripheral devices, and poor correlation of command requests with responses.

Firstly, on iOS (our current target platform), a unique identifier is generated for every bluetooth peripheral *for every iOS application*, i.e., three applications connecting to the same peripheral (even on the same device) will be passed a different unique identifier for that peripheral. This behavior provides some degree of privacy and security but presents a challenge in managing peripherals across applications and devices.

Secondly, in all current BLE central implementations, callbacks from `GET`, `SET`, and `NOTIFY` commands are shared and can return in a different order than their initiating calls. While these callbacks do contain the characteristic UUID that is being queried, if one is repeatedly calling `GET`, `SET`, and/or `NOTIFY` on a single characteristic, maintenance of proper callback order becomes a formidable task for the developer that must be reimplemented for each application.

The fabryq mobile application and its JavaScript API have been implemented to overcome these limitations. Specificially, we have devised the following control flow, shown in Figure 9, resulting in minimal extra user interaction with no additional developer code. Commands (i.e. `GET`, `SET`, `NOTIFY`) to be performed on devices are queued on the server by calls to the fabryq javascript API. If commands are found for the user's devices, the fabryq mobile application first determines if it has an OS-specific BLE-identifier for that device. If it does, the action is performed and the result returned to the server. If not, all actions for that device are

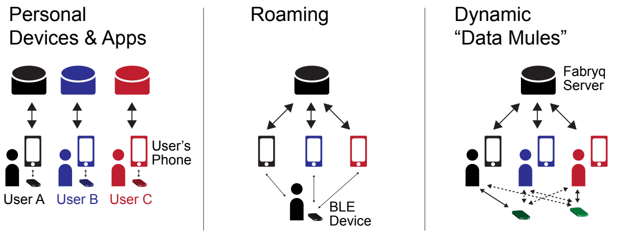


Figure 8. Possible application configurations enabled by Fabryq: A) single user, single device. B) A user's peripheral accesses fabryq through more than one mobile device in a "roaming" pattern C) Users who agree to jointly run an application can also act as "data mules" and pick up data from environmental sensors whenever they walk by such a sensor.

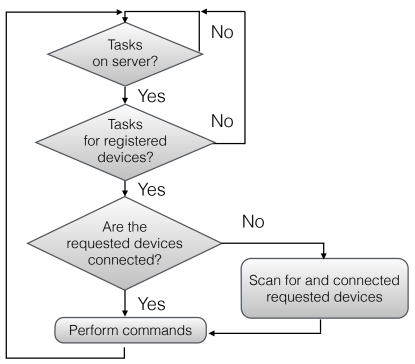

Tasks on server?
No
Yes
Tasks for registered devices?
No
Yes
Are the requested devices connected?
No
Scan for and connected requested devices
Yes
Perform commands

Figure 9. Control flow for fabryq mobile. If commands (i.e. `GET`, `SET`, `NOTIFY`) are found for the logged-in user's devices, fabryq mobile first determines if it has an OS-specific BLE-identifier for that device. If such a "link" between OS-specific identifier and device exists, the action is performed and the result returned to the database. If not, it will not perform that action until a link is established by the user with the fabryq mobile app. Next, the gateway will search for any devices that are requested and have been linked, but are not currently connected.

ignored and skipped. This allows certain devices to be paired with some gateways and not others without causing issues.

By decoupling issuing `GET`, `SET`, and `NOTIFY` commands from performing them, we also address BLE's command ordering limitation. fabryq keeps track of the order in which commands for a given device are issued and executes them in the same order. The result of each command is then associated with the issuing action and passed to the client.

UI Pushdown with `SHOWURL`

In order to make it easier to show user interfaces on a phone when the main fabryq application is executing in a browser elsewhere, the API offers a `SHOWURL` command. Instead of passing the command over BLE, it shows a URL on the fabryq mobile application itself. In this way, a peripheral-specific UI can appear on the mobile device connected to it. Since `SHOWURL` is simply opening a web page,fabryq JavaScript API calls are also available. This utility feature is intended to be used with continuous monitoring applications for alerting the user to events or requesting input.

Fabryq Server and Javscript API

We have implemented a JavaScript API to enqueue and dequeue commands and poll their results. These commands are intended for BLE peripherals but originate on the cloud: our *peripheral proxy* model. Each `GET`, `SET`, or `NOTIFY` has an explicitly defined callback that is passed to the originating JavaScript function call (and per common JavaScript coding practice can be defined in-line).

fabryq contains several features to simplify issuing commands and using their results. Behind the scences, fabryq automatically queues actions to ensure that lists of actions are submitted, and therefore executed in the issued order. fabryq also supports both the lambda-function handler model as well as a tree-structured event handler model inspired by existing GUI systems such as Swing [11]. In the tree model, new data and error events start at their corresponding device object and "bubble up" the hierarchy until they are handled, next to their

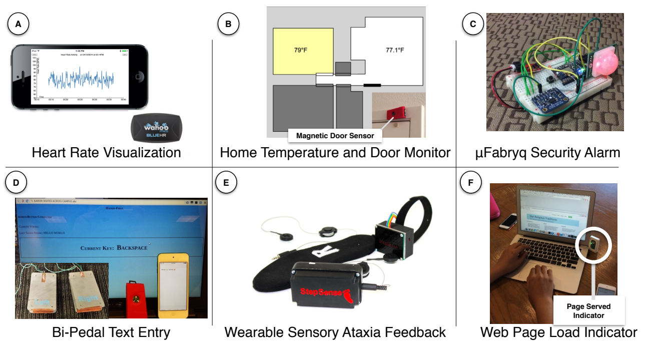


Figure 10. Fabryq applications created by the authors and users leverage both off-the shelf BLE devices (A,B) as well as custom hardware (C-F), including the µfabryq platform (C).

device type, then to the app itself. Handlers can be attached at any point in the tree for maximum programming flexibility. For instance, a given developer might decide that a device disconnection event should be caught by a device-specific hander, new data events should be processed by a handler for all devices of a given type, and an application-level handler could process unexpected or uncommon events.

One limitation of the JavaScript API is the requirement to poll the server to discover the status of an action. Trading off simplicity and portability for latency, future implementations of the interface could utilize WebSockets or other push notification schemes to allow for instant callbacks to JavaScript as soon as fabryq mobile has interacted with the BLE peripheral and updated the log. However, in cases where both the client program and the gateway device have a robust internet connection, the latency from a BLE event's trigger to the update on the client is generally under one second.

The core fabryq implementation was written in about 10,000 lines of code split across iOS (fabryq mobile); PHP and SQL (fabryq server) and Javascript (fabryq API).The heterogeneity of the codebase exemplifies the complexity of development that fabryq seeks to overcome.

Support for long-running applications

While fabryq's base functionality is useful for communicating with BLE devices while a browser window is open, a fabryq application cannot respond to events such as a device disconnection or new data once the browser is closed. To address this issue, a `startAgent()` fabryq command can open a URL on a remote server and keep the page alive. This facility provides a simple route to turn an existing fabryq application into a long-running application by copying the program logic into the "agent" web app and instantiating that agent in the "client" version of the app. The UI of the client can then be updated based on messages passed by the agent.

EXAMPLE APPLICATIONS

The authors and several users have created example applications to demonstrate working with both off-the-shelf and custom peripherals. When creating demonstrative examples, we sought to investigate fabryq's ability to work with an array of different devices and services, at different levels of complexity and development time. Images of several example applications can be found in Figure 10.

We examined people's ability to use fabryq on three different timescales: one hour "lightning hack sessions", a half-day hackathon, and longer-term projects, where users wrote code over the course of several days. In order to aid users' prototyping and development process, we provided several BLE devices:

Texas Instruments SensorTag A small commercial BLE device that contains buttons, as well as inertial, magnetic, and climate sensors.

LightBlue Bean A BLE device with a built-in RGB LED and accelerometer, pre-configured to expose LED control over BLE characteristics.

µfabryq A custom prototyping device flashed with a firmware we created that exposes I/O and an SPI peripheral over BLE to enable connection of other sensors and actuators.

mBed nRF51822 ARM Cortex microcontroller with onboard BLE radio; programmed in C++; for experienced embedded developers.

Author examples

As a test of the complexity ceiling of fabryq and to investigate how fabryq can support the design of custom BLE peripherals, we created a device with functionality resembling Arduino [19], a popular microcontroller, while offering wireless communication and Javascript programmability. µfabryq, shown in Figure 10C, is a Bluetooth system-on-chip based on the BlueGiga BLE113 module. We developed custom firmware that allows Arduino-like commands to be used over BLE. For example, there is a custom BLE characteristic that controls the output of all GPIO pins (high or low) and another that controls the pin direction (input or output). Using only the fabryq JavaScript API and jQuery, we then created the µfabryq JavaScript API, an extension of the main fabryq API, which maps Arduino commands to JavaScript functions, executing them over fabryq and returning their result to the browser. A list of these µfabryq API functions, along with the fabryq JavaScript API functions, is in Table 1. The µfabryq code base is about 1000 lines and demonstrates how other tools can be constructed on top of the fabryq substrate.

We also evaluated how fabryq can be used with multiple peripherals. Our home monitoring system consists of several TI SensorTags and µfabryq boards. The devices are placed around the house and can detect whether doors are open or closed, the temperature and humidity in the house, as well as room occupancy using PIR sensors. Data from the devices is collected by a single fabryq Gateway in the house and is monitored by a fabryq agent. Current data is displayed on a fabryq web app shown in Figure 10B that communicates with the agent via fabryq's message passing interface.

As an example of fabryq's utility for more real-time applications, we also created a class polling application using TI SensorTags. Several tags can be distributed to a group of people and each can vote for one of three options using the Tag's

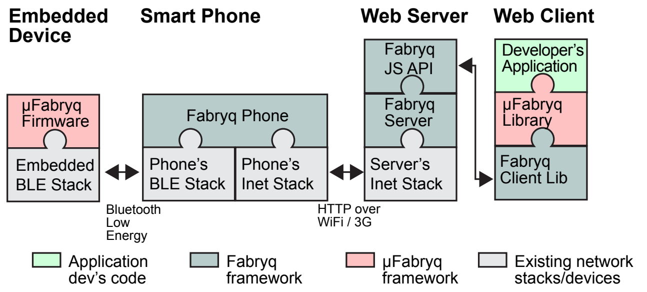


Figure 11. μfabryq offers direct access to embedded peripheral pins in a manner similar to the popular Arduino platform. It was implemented using Fabryq and shows the expressivity of the framework. It comprises a BLE firmware and a JavaScript library written using the Fabryq JS API.

built-in buttons. A fabryq web application displays the results of the vote in real time. The webpage can display which tag voted for which option, if necessary.

We also created a simple fabryq application that can plot the heart rate of a user using any heart rate monitor that exposes the standard BLE heart rate service. In addition to showing the heart rate on the user's web browser, the application uses the `SHOWURL` fabryq command to also display the plot on the fabryq mobile application currently communicating with the heart rate monitor (Figure 10A).

In all of these applications, fabryq simplifies configuring the devices and getting their data. In general, writing the code to get the data from the relevant devices took only a few minutes, while creating, testing, and tweaking the user interfaces (which is beyond the scope of fabryq) took hours.

User examples

In order to evaluate how effective fabryq was for enabing developers to more easily create networked devices, we had users create applications in multiple scenarios.

One Hour: The one hour session participants were all computer science graduate students who were familiar with web programming, but not embedded programming. The sessions produced three example applications with pre-configured devices, each completed in an hour that included setup time, understanding fabryq and development. The most compelling of these, "Website Popularity," (Figure 10F) was built by a user who owned a website with bursty traffic patterns and she wanted a simple indicator of when a burst of activity was occurring. Her app would check a REST endpoint on her server (created during the development time) that would report whether a page had been served since the last check. When the call indicated a page had been loaded, the app would instruct the Bean to blink its LED once. Other users employed the SensorTag as a button to control sound effects; and a Bean LED as a secondary output for a simple game.

Half Day: To evaluate the utility of μfabryq, which encourages the use of custom electronics that take longer to construct, we planned a half-day session to allow for ideation, circuit construction and coding. The participants were all undergraduate or master's students primarily from a Biomedical Engineering background most of whom had limited experience with either web or embedded programming. In our hackathon, two groups produced a networked heart-rate monitor and tic tac toe board. The heart rate monitor cleverly connected an existing 5V light-based heart rate monitor to the 3.3V μfabryq board using PWM, an RC circuit, and the μfabryq board's A/D converter. The tic tac toe board used GPIO pins to digitize the state of the board and displayed it on a web page.

fabryq JavaScript API	
Function	**Description**
GET	Request a BLE characteristic
SET	Set the value of a BLE characteristic
NOTIFY	Receive notification of a changed BLE char.
SHOWURL	Show a URL on the connected fabryq mobile app.
startAgent	Open a fabryq application on a remote server.

μfabryq JavaScript API	
Function	**Description**
DigitalRead	Read a pin's binary value
DigitalWrite	Set a pin's output voltage to low or high
PinMode	Set a pin to input (w/ pullup) or output
AnalogRead	Read a pin's input voltage using an ADC
AnalogWrite	Set a pin's output voltage using PWM
AttachServo	Enable servomotor control on a pin
ServoWrite	Set a pin's servo to a particular location
AttachInterrupt	Attach a JS function to a pin interrupt
SPIbegin	Enable/configure the SPI on μfabryq
SPItransfer	Perform full-duplex SPI communication

Table 1. Function list for the microfabryq JavaScript API (top) and the μfabryq JavaScript API (bottom).

Class projects: Fabryq was also available to students in a course on interactive device design at our institution (its use was not required for the class). One student incorporated a SensorTag into a hands-free text entry device (Figure 10C) during a one-week assignment. A second group of students incorporated fabryq into a wearable medical device for patients with sensory ataxia (loss of feeling in their feet – Figure 10E). The *StepSense* device takes pressure readings from custom shoe insoles and transforms them into vibration feedback delivered to a patient's back. Fabryq was used to transmit step pressure information to a web browser, where a physician could monitor the operation of the device. The class projects were both created by undergraduate EECS students.

Lessons learned

Successes

Many of the participants appreciated fabryq's ability to simplify what would otherwise be a complex networking task. Instead of writing code for an embedded device, a mobile app, and a server, participants were able to get started quickly and write functioning applications using only JavaScript in the browser. One participant remarked, "It's really, really easy to use, especially compared to the complexity of what it accomplishes. I think the current API and the general spirit of the language's structure makes it really intuitive to use." fabryq made it easy for people with little hardware experience to prototype using embedded devices. Reflecting on a one hour session, another user said, "The API was very simple to use! I can't believe you can just tell an LED light to flash and it will do so. Hardware has always seemed very scary to me."

fabryq also shows potential for simplifying the construction of multi-device applications. When designing a native iOS or Android application, the complexity of BLE communication balloons as more devices are added. Thanks to the abstract nature of specifying fabryq applications, adding compatibility for a new device to a multi-device application like the home monitor is as simple as scanning any new devices with the fabryq app and adding them to the application's requirements.

fabryq's structure also gives it a power and flexibility advantage over WiFi-based systems. In terms of power, communicating frequently with the Internet and BLE does consume extra power on the gateway device, however the use of BLE over WiFi helps save power for wearable or embedded devices. In a way, this approach shifts the burden from embedded devices whose batteries should last as long as possible (weeks or months) to the gateway's battery that many users recharge daily. In terms of flexibility, fabryq-connected BLE devices do not have to manage authentication to local WiFi networks or the burden of maintaining an IP stack.

Challenges

The largest challenges that the participants faced involved understanding the somewhat abstract nature of fabryq applications and managing application failure modes. Although specifying applications in a abstract fashion allows fabryq apps to flexibly work with different compatible devices, comprehending the initial concept delayed the progress of some participants. In response, we added the ability to create a single concrete prototype first, and later abstract its hardware requirements to launch multiple instances.

The second main challenge relates to recognizing and dealing with errors for both developers and users. When possible, fabryq tries to shield the developer from the complexity and uncertainty of networking. However, there are times when the system needs to recognize failure states and inform the user in a sensible way. Early versions of fabryq were not explicit enough in informing the developer about error conditions. This prompted us to design our event handling-based method of reporting errors for the present version of fabryq, where errors are handled flexibly by a developer's program and unhandled errors are printed to the JavaScript console.

LIMITATIONS

Design Limitations

Fabryq makes a fundamental tradeoff between performance and ease-of-authoring: all commands require complete round-trips from app to cloud to phone to device and back. We chose this architecture to facilitate prototyping and prioritized simplicity of development. Offloading computation to the phone or caching data there are interesting avenues for future work, but will necessarily impact ease of development.

Fabryq-enabled applications require a priori knowledge of all peripheral types prior to writing and running an application. This allows robust connectivity across multiple mobile devices and scalability of applications to many users and many peripherals. There exist a subset of applications where the peripherals are not known, for example, an application specifically designed to discover *any* nearby peripheral, or applications that interact with beacons that change their identifiers for security reasons. Such applciations are not supported.

Implementation Limitations

The current fabryq mobile application runs only on iOS, because BLE support on Android was not yet stable when we began work. On iOS, there are stringent requirements placed on background applications, such as timing limitations. Fabryq thus works best when run in the foreground. However, we successfully push these limitations in some circumstances. Specifically, whenever the fabryq app receives new data from an established BLE NOTIFY while in the background, iOS briefly wakes the app up to handle the event. During that time, we can post the result of the command and check for and execute new commands, before resigning control. This strategy works best when a constant stream of notifications is arriving, as in the case of a heart rate monitor. However, when no new data is arriving, the only way to ensure proper background behavior on iOS would be to use a jailbroken device. This also led us to implement an OSX gateway application for situations where extended monitoring is valued over mobility.

An additional limitation is poorly–defined command latency. This is not a fundamental limitation; if the application configuration and command log were located on the mobile device, the command latency could be readily defined. Instead, our current implementation relies on polling to A) retrieve command logs for fabryq mobile and B) retrieve results of these commands by the JavaScript API. In future implementations, both of these polling mechanisms, which take place over REST-ful interfaces, could be replaced by TCP/IP socket schemes such as WebSockets. However, making the server the central point for all interactions does allow for apps that operate in geographically distant locations.

Finally, when programming fabryq apps, a developer can either write code in a local text editor or using fabryq's web editor. Testing this code generally involves opening the application in another browser window and having an iOS device running fabryq mobile physically near the peripheral and presumably the developer. This yields five potential points of interaction (two browser windows, an editor window, fabryq mobile, and the peripheral) for the developer which can become a challenge that could slow down the development cycle. In future versions of fabryq, some of these points of interaction can be eliminated, e.g., by providing an integrated workbench that unifies writing, configuring and running applications, and by adding device simulation capabilities (e.g., through trace playback [20]). However, in practice, the application configuration is finalized early in the development cycle and the mobile app can run undisturbed, so the developer primarily focuses on the browser running the application, the editor, and the device(s).

CONCLUSION AND FUTURE WORK

This paper presented fabryq, a platform for rapidly writing and deploying *MGC* applications that use smart phones as proxies to control BLE devices from the Web. The development of fabryq was guided by a survey of commercial devices, medical wearable devices, and student projects. fabryq's main contribution is its *protocol proxy* model of executing BLE protocol calls from the Web in order to simplify the creation of applications of smart devices Future work on fabryq will focus on also making the firmware layer updatable from the Web. Just as devices' GATT tables have been abstracted, future versions of fabryq may similarly absorb device firmware as part of a single application that runs on phone, cloud, and peripheral. Future fabryq applications will have a "complete picture" of tasks to be executed on the peripheral, phone, and cloud. With this picture, individual tasks can be parceled out to the appropriate layers, depending on available network infrastructure, available hardware infrastructure, and the nature of the tasks requested.

ACKNOWLEDGMENTS

This work was supported in part by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Additional support was provided by a Sloan Foundation Fellowship, a Google Research Award, and the CITRIS Connected Communities Initiative. This publication was also supported by the National Center for Advancing Translational Sciences, National Institutes of Health, through UCSF-CTSI Grant Number UL1 TR000004. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the NIH.

REFERENCES

1. 6LoWPAN. http://en.wikipedia.org/wiki/6LoWPAN.
2. Ballagas, R., Memon, F., Reiners, R., and Borchers, J. iStuff mobile: Rapidly prototyping new mobile phone interfaces for ubiquitous computing. In *Proceedings of CHI*, ACM (2007), 1107–1116.
3. Bluetooth Low Energy Specification Adopted Documents. `https://www.bluetooth.org/en-us/specification/adopted-specifications/`.
4. Buechley, L., Eisenberg, M., Catchen, J., and Crockett, A. The lilypad arduino: using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In *Proceedings of CHI*, ACM (2008), 423–432.
5. Carter, S., Mankoff, J., Klemmer, S. R., and Matthews, T. Exiting the cleanroom: On ecological validity and ubiquitous computing. *Human–Computer Interaction 23*, 1 (2008), 47–99.
6. Chaudhri, R., Brunette, W., Goel, M., Sodt, R., VanOrden, J., Falcone, M., and Borriello, G. Open data kit sensors: Mobile data collection with wired and wireless sensors. In *Proceedings of the ACM DEV*, ACM (2012), 9:19:10.
7. Greenberg, S., and Fitchett, C. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of UIST*, ACM (2001), 209–218.
8. Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., and Gee, J. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of UIST*, ACM (2006), 299–308.
9. Heydon, R. *Bluetooth low energy: the developer's handbook*. Prentice Hall, 2013.
10. Electric imp. http://electricimp.com/.
11. Java swing. http://docs.oracle.com/javase/7/docs/.
12. Kamath, S., and Lindh, J. Measuring bluetooth low energy power consumption. *Texas instruments application note AN092, Dallas* (2010).
13. Kaufmann, B., and Buechley, L. Amarino: A toolkit for the rapid prototyping of mobile ubiquitous computing. In *Proceedings of MobileHCI*, ACM (2010), 291–298.
14. Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J., Serra, B., and Spasojevic, M. People, places, things: Web presence for the real world. *Mob. Netw. Appl. 7*, 5 (Oct. 2002), 365–376.
15. Kinoma create. http://kinoma.com/create/.
16. Kuo, Y.-S., Verma, S., Schmid, T., and Dutta, P. Hijacking power and bandwidth from the mobile phone's audio interface. In *Proceedings of ACM DEV*, ACM (2010), 24:124:10.
17. Lin, F. X., Rahmati, A., and Zhong, L. Dandelion: A framework for transparently programming phone-centered wireless body sensor applications for health. In *Wireless Health 2010*, ACM (2010), 74–83.
18. Marquardt, N., and Greenberg, S. Distributed physical interfaces with shared phidgets. In *Proceedings of TEI*, ACM (2007), 13–20.
19. Mellis, D., Banzi, M., Cuartielles, D., and Igoe, T. Arduino: An open electronic prototyping platform. In *Proceedings of CHI*, vol. 2007 (2007).
20. Newman, M. W., Ackerman, M. S., Kim, J., Prakash, A., Hong, Z., Mandel, J., and Dong, T. Bringing the field into the lab: Supporting capture and replay of contextual data for the design of context-aware applications. In *Proceedings of UIST*, ACM (2010), 105–108.
21. Spark core. https://www.spark.io.
22. Spark core hardware characteristics. http://docs.spark.io/hardware/.
23. Sterling, B., Wild, L., and Lunenfeld, P. *Shaping things*. MIT press Cambridge, MA, 2005.
24. Tessel. https://tessel.io.
25. Villar, N., Scott, J., Hodges, S., Hammil, K., and Miller, C. .NET gadgeteer: A platform for custom devices. In *Proceedings of Pervasive*, Springer-Verlag (Berlin, Heidelberg, 2012), 216–233.
26. Weiser, M. The computer for the 21st century. *Scientific american 265*, 3 (1991), 94–104.