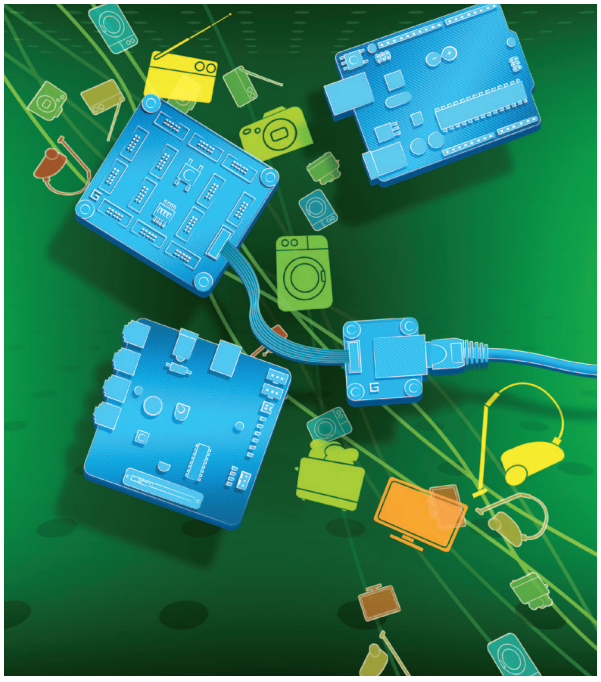


Prototyping Connected Devices for the Internet of Things

Steve Hodges, Stuart Taylor, Nicolas Villar, and James Scott, *Microsoft Research Cambridge, UK*
Dominik Bial, *University of Duisburg-Essen, Germany*
Patrick Tobias Fischer, *University of Strathclyde, Glasgow, UK*

Tools like Microsoft .NET Gadgeteer offer the ability to quickly prototype, test, and deploy connected devices, providing a key element that will accelerate our understanding of the challenges in realizing the Internet of Things vision.

Today, devices such as personal computers and smartphones form a significant fraction of Internet-connected devices globally.[1] However, in the Internet of Things (IoT) vision, network connectivity extends to the very simplest electronic devices—to the point where almost anything can connect to the Internet. Indeed, analysts predict that simpler embedded devices will increasingly complement the established platforms as peers on the Internet in a growing machine-to-machine communication paradigm.[1-3] In addition to networked versions of commonplace devices—washing machines, alarm clocks, doorbells, and so on—new applications are predicted. Pundits imagine embedded devices that continuously communicate with each other and improve our productivity, help us manage daily activities, let us more easily keep in touch with others, provide timely information more conveniently, and enhance our leisure time. The IoT is estimated by some to constitute 100 billion devices as soon as 2020.[2,3]

One anticipated consequence is that the software running within the embedded devices constituting the IoT will increasingly be complemented by cloud-based Web services, made accessible via built-in network interfaces leveraging Web-based protocols such as HTTP and XML. Web services will dramatically extend the effective processing and storage capabilities of these connected devices—relatively cheap embedded processors will routinely leverage sophisticated data processing and access large datasets in the cloud. Ultimately, a new class of applications might emerge in which groups of devices act as the I/O elements of potentially global-scale distributed services and applications.

Although the IoT vision has tremendous potential, many complex technical, social, and economic questions remain unaddressed. With so many possibilities across the broad IoT realm, the role of hardware and software platforms that expedite the reduction of ideas to working prototypes is an intriguing consideration. We outline here some of the currently available hardware tools and services that facilitate the prototyping of networked embedded devices. We illustrate the possibilities these tools afford by focusing specifically on Microsoft .NET Gadgeteer (http://netmf.

 Published by the IEEE Computer Society

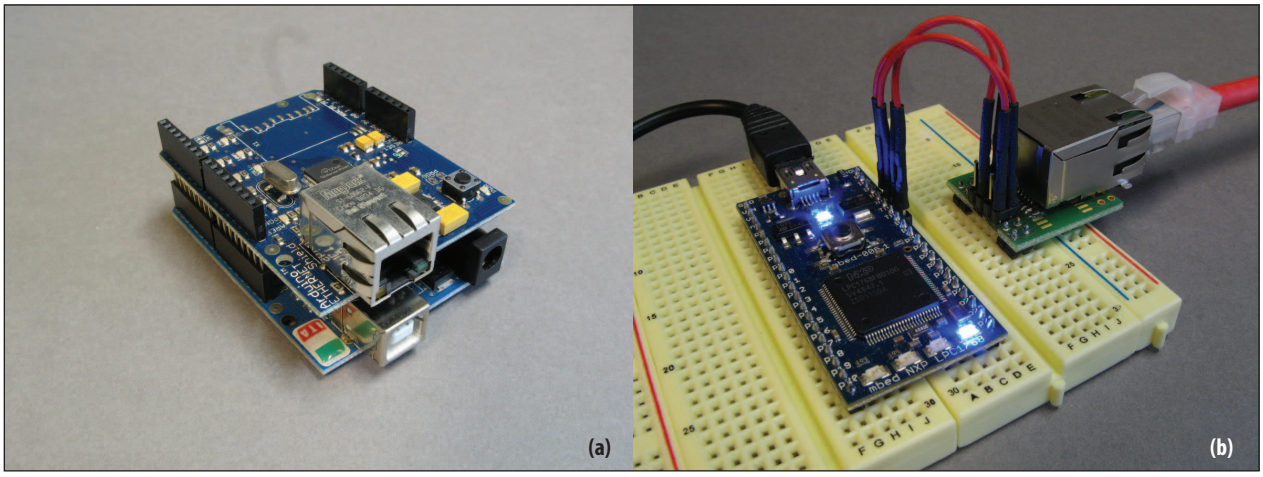


Figure 1. Connected-device prototyping tools: (a) Arduino device prototyping platform with Ethernet shield and (b) the mbed embedded development platform.

com/gadgeteer), a general-purpose device development platform which we have developed.[4] Key elements of tools like Gadgeteer include rapid construction and reconfiguration of electronic device hardware, ease of programming and debugging, and the ability to leverage online Web services for additional storage, communication, and processing. We believe the ability to quickly prototype, test, and deploy devices will be a key element in accelerating our understanding of the challenges and benefits of networked things.

CONNECTED-DEVICE PROTOTYPING TOOLS

Several tools for turning embedded machine-to-machine communication concepts into working systems exist. One of these is the Arduino platform (http://arduino.cc),[5] a family of embedded processors that can be programmed with the C language via an accessible, minimalist integrated development environment (IDE). Debugging with Arduino is typically supported via simple communications over a serial line interface. In terms of electronic hardware, Arduino processors are complemented by an ecosystem of "shields"—add-on circuit boards that extend the platform's basic capabilities (http://shieldlist.org).

From a hardware perspective, shields that provide Ethernet, Wi-Fi, and GPRS connectivity enable Arduino's use for connected-device development. On the software side, developers commonly use the representational state transfer (REST) technique[6] because it is a lightweight, easy-to-debug way to communicate between connected devices such as those built with Arduino, shown in Figure 1a. With REST, services are exposed and accessed using HTTP, which is readily supported by Arduino libraries that implement the relevant networking protocols and enable simple webserver operation. Moreover, Arduino's widespread use has formed a vibrant community of users who create, share, and support additional libraries and examples online, further facilitating the development of new applications.

Figure 1b shows another popular tool, mbed (http://mbed.org), an embedded electronics development platform available as two different microcontroller products. Both of these are small rectangular modules with protruding pins that developers can use to insert the device into a breadboard for prototyping. This form factor also allows the module to be subsequently integrated into a custom printed circuit board (PCB) if necessary.

A key difference from Arduino is mbed's online IDE, accessible via a Web browser without the need to install any software. Extensive documentation and libraries are available through the IDE, which also supports the sharing of user-generated code samples and libraries.

To support connected-device development, one mbed variant includes built-in Ethernet connectivity, and the mbed code repository includes a comprehensive set of networking libraries and examples. Support for debugging code running on the mbed has been limited to date, but it is possible to transition to a more traditional PC-based IDE, and an upcoming version of mbed should provide better support for debugging via the online IDE.

In addition to microcontroller-based platforms such as Arduino and mbed, several small-form-factor devices that run Linux are available. These devices offer the opportunity to leverage an extensive set of preexisting tools and software components such as Node.js (http://nodejs.org), which simplifies the implementation of REST-like asynchronous Web-based application programming interfaces (APIs). These platforms are powerful and flexible, but they can expose more complexity to the user and are typically less cost-effective than Arduino for lightweight device development. However, new products with low price points, such as Raspberry Pi (http://www.raspberrypi.org) and BeagleBone (http://beagleboard.org/bone), are

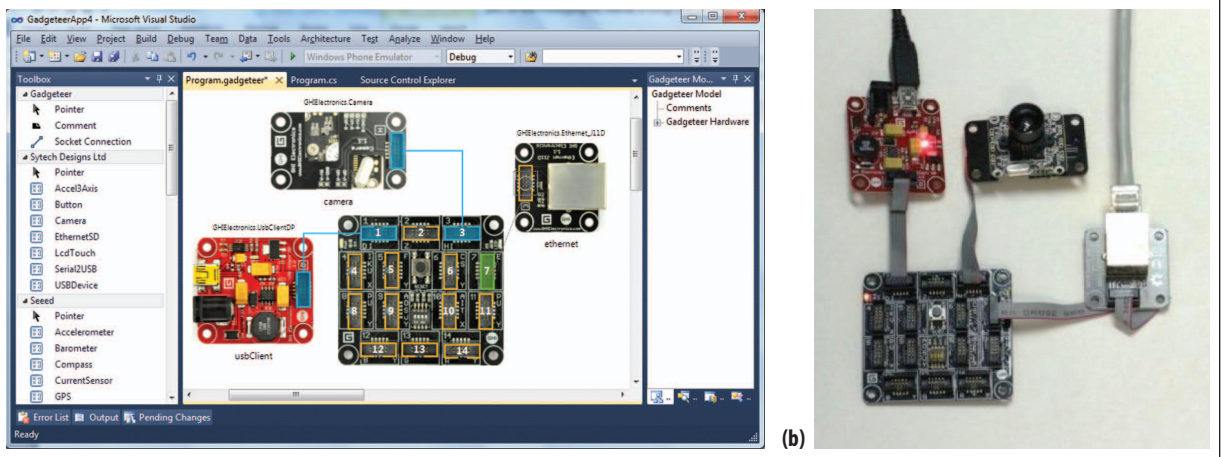


Figure 2. An Internet webcam constructed with .NET Gadgeteer. (a) The hardware configuration, which includes an RJ45 module for a wired Ethernet connection, is entered graphically in Visual Studio. When a module connector is selected, compatible mainboard sockets are highlighted in green to aid the user in wiring up the design. (b) The corresponding physical hardware of the completed webcam.

gaining popularity and consequently have growing online communities.

One final system that illustrates how simple the prototyping of connected devices can be is Microsoft .NET Gadgeteer. A modular platform that facilitates construction of digital-device prototypes,[4] Gadgeteer comprises a central "mainboard" containing a CPU and several sockets that developers can connect to a large number of different modules, including sensors, actuators, displays, and communication and storage elements. The solder-less composability of hardware components allows developers to quickly construct, reconfigure, and extend prototypes.

The Gadgeteer system is tightly integrated with the Microsoft Visual Studio IDE, which provides support throughout the prototyping process. Visual Studio's IntelliSense feature performs dynamic syntax checking and continually provides hints and prompts to ease coding. The IDE also aids debugging via breakpoints, single stepping, variable watches, and execution traces.

MICROSOFT .NET GADGETEER DESIGN CHOICES

When we created Gadgeteer, a primary design goal was to simplify application development as much as possible, even if this occurs at the expense of performance. Developers use C# or Visual Basic to program device functionality—although using managed code is unusual for embedded device development where C-like languages are firmly established, in our experience it tends to reduce the time and expertise needed for prototyping.[7] In a preproduction environment, this more than offsets the increased processor and memory requirements. Similarly, Gadgeteer uses an event-based model wherever possible, which further simplifies the creation of many applications and helps developers familiar with event-based programming on desktop and mobile platforms to transition to embedded device development.

A software library encapsulates each physical Gadgeteer module's functionality through an intuitive high-level API. The high abstraction level often allows modules to be used in sophisticated ways with just a few lines of code, enabling users with relatively little experience to build compelling devices and applications. This approach lowers the barrier to entry for device development, but doesn't limit the flexibility available to more-experienced developers. It is possible to build on top of lower-level APIs to encapsulate a different abstraction or functionality if this is required.

The .NET Micro Framework (http://netmf.com), an open source platform that underpins Gadgeteer's similarly open software stack, contains extensive provision for networking. The Gadgeteer networking API builds on this in a way that supports a compact, easy-to-understand design pattern for responding to REST-ful Web requests with text, images, or byte streams. To simplify IoT application development, when developing Gadgeteer we prioritized REST-ful support over other Web-related functionality, such as serving a hierarchy of content as a traditional webserver does.

BUILDING A WEB-CONNECTED DEVICE WITH GADGETEER

To illustrate how straightforward creating a connected, REST-ful–interfaced device can be, Figure 2 shows a simple "Internet webcam" that we built with Gadgeteer. As Figure 2a shows, the process began with a graphical design tool to specify the hardware components and how to connect them to a mainboard. Having "wired these up"

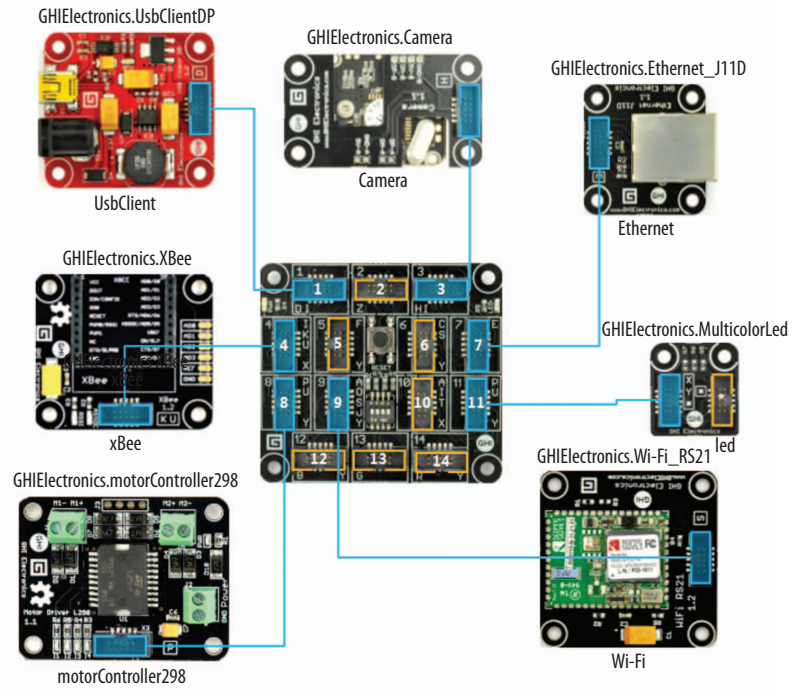
```
WebEvent cameraWebEvent;
Responder currentResponder;

void ProgramStarted()
{
       // associate PictureCaptured event with its handler
       camera.PictureCaptured += new Camera.PictureCapturedEventHandler(camera_PictureCaptured);

       // request DHCP address and associate handler for network setup
       ethernet.UseDHCP();
       ethernet.NetworkUp += new NetworkModule.NetworkEventHandler(ethernet_NetworkUp);
}

void ethernet_NetworkUp(GTM.Module.NetworkModule sender,
         GTM.Module.NetworkModule.NetworkState state)
{
       // start a webserver on port 80
       WebServer.StartLocalServer(ethernet.NetworkSettings.IPAddress, 80);

       // set up a handler for http '/picture' requests
       cameraWebEvent = WebServer.SetupWebEvent("picture");
       cameraWebEvent.WebEventReceived += new
         WebEvent.ReceivedWebEventHandler(cameraWebEvent_WebEventReceived);
}

void cameraWebEvent_WebEventReceived(string path, WebServer.HttpMethod method,
         Responder responder)
{
       // initiate a picture and cache the responder to use when the picture is captured
       currentResponder = responder;
       camera.TakePicture();
}

void camera_PictureCaptured(GTM.GHIElectronics.Camera sender, GT.Picture picture)

{
       // respond to web request with the picture
       currentResponder.Respond(picture);
}
```

Figure 3. Just 11 lines of code are enough to create a webserver that responds to incoming requests with a new image. Note that Visual Studio automatically generates all the function definitions in this example. This code was written using Gadgeteer SDK v4.1.

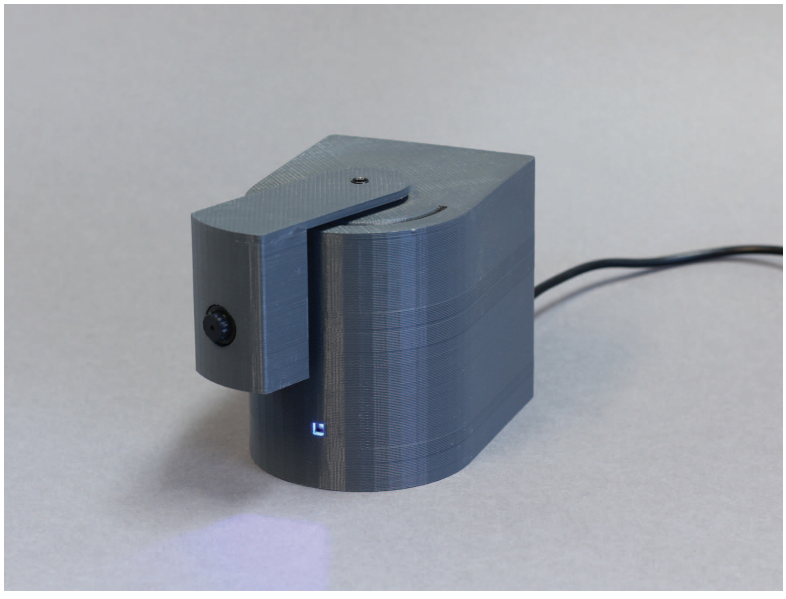
graphically on-screen, constructing the corresponding physical hardware took just a couple of minutes. The completed webcam, shown in Figure 2b, consists of a Gadgeteer mainboard connected to Ethernet, camera, and power supply modules. The code required to encapsulate the hardware configuration was automatically generated and the appropriate libraries were linked in.

With Gadgeteer, the developer only needs a single line of code to set up a webserver once a network connection has been established. Gadgeteer's event-based programming style lends itself to handling REST-ful requests by creating event handlers for each desired HTTP request path; each handler simply responds to the associated incoming request with the appropriate object. The Gadgeteer API directly supports strings (including complete HTML pages), images, and data streams.

In the case of the webcam, an HTTP request from a remote client triggers the capture of a new image. The captured image is returned to the Web client initiating the request. Figure 3 lists the C# code required to implement the necessary functionality using v4.1 of the Gadgeteer SDK—just 11 lines of code excluding the autogenerated function prototypes. Of course, a more robust implementation would cover a variety of potential error conditions, such as lack of network connectivity, but here we show the simplest implementation for clarity.



(a)



(b)

Figure 4. Web-controlled camera. (a) The remote-controllable networked camera with servo-controller pan mechanism. (b) A 3D-printed plastic enclosure houses the electronic modules.

Figure 4 shows a more sophisticated Web-controlled camera. In this case, the camera module is attached to a servomotor-controlled arm, allowing remote panning as well as image capture, again over a REST-ful interface. In addition to a wired Ethernet connection, this device also incorporates wireless Zigbee and Wi-Fi network interfaces. The latter provides an alternative way of connecting to the Internet, should a wired connection be unavailable or inconvenient. The Zigbee interface supports a connection to lighter-weight Gadgeteer devices such as temperature and light-level sensors, effectively giving them a presence on the Internet via additional software running on the camera device that acts as a bridge.

Storing, retrieving, and sharing data

Although an embedded webserver allows a device to expose state or functionality over HTTP, the ability to store, retrieve, and share data is a key element of IoT applications, and an embedded Web client API is equally important for these applications. For this reason, the Gadgeteer libraries were designed to ensure that making a Web request is as straightforward as receiving one. When the details of the HTTP request have been specified, an event handler is created to deal with the anticipated response, then the request itself is sent.

In addition to supporting true peer-to-peer communication, the HTTP protocol offers an intuitive way of providing access to a growing number of hosted Web services that support the process of exchanging data between connected devices. These tools, which include cosm (formerly Pachube; https://cosm.com), ThingSpeak (https://thingspeak.com), and Nimbits (www.nimbits.com), use HTTP and XML to implement REST-ful APIs and are therefore readily accessible to platforms like Gadgeteer.

Figure 5 lists the four lines of C# code needed to upload a barometric pressure reading to the online cosm repository. Figure 6 shows a complete connected device that continuously records and uploads sensor readings, along with a screenshot of the cosm Web interface for visualizing the associated temperature and pressure data.

Cloud-based processing for connected devices

In addition to communication between devices via online repositories such as cosm, a key benefit of con-

```
HttpRequest request = HttpHelper.CreateHttpPutRequest("http://api.pachube.com/v2/feeds/" + feedId +
     ".csv", PUTContent.CreateTextBasedContent(locationId + "temperature," +
     sensorData.Temperature.ToString() + "\n" + locationId + "pressure," +
     sensorData.Pressure.ToString()), "text/csv");
request.AddHeaderField("X-PachubeApiKey", apiKey);
request.ResponseReceived += new HttpRequest.ResponseHandler(req_ResponseReceived);
request.SendRequest();
```

Figure 5. A snippet showing the code required to upload a sensor reading to the cosm Web service.

nected operation is the potential to leverage cloud-based computation. Services such as Amazon EC2 and Microsoft Azure provide a mechanism to deploy online compute services that developers can use to offload computation from connected devices. Project Hawaii from Microsoft Research (http://research.microsoft.com/hawaii/) is a ready-to-use Web services testbed built on Azure. It provides, for free, varied functionality for noncommercial applications. This includes off-the-shelf services to support certain computationally intensive processes, basic communication between remote devices, and online data storage.

To demonstrate how Project Hawaii can extend Gadgeteer's capabilities, Figure 7 shows another camera device prototype. As with a traditional digital stills camera, this device captures an image when the "shutter" button is pressed. At this point, rather than simply displaying the image and storing a copy locally, the camera sends the image to the Hawaii optical character recognition (OCR) service. Any text that is detected is returned and overlaid on the display.

Figure 8 shows the C# code that forms the basis of this example using v4.1 of the Gadgeteer SDK. When the camera has captured and displayed a picture, it creates an HTTP request to be sent to the Project Hawaii service for OCR processing. This request incorporates the image, the appropriate authentication information, and the necessary HTTP header fields. The resulting response from the OCR service triggers an event handler to process the XML-formatted results.

For simplicity, the code in Figure 8 simply selects the first word the OCR service returns and includes no error handling; a more complete implementation would process all the text and associated metadata as well as resolve error conditions. The .NET Micro Framework includes native XML parsing and exception handling capabilities, which simplifies implementing more complete and robust decoding.

To further illustrate how developers can use Gadgeteer to explore applications that leverage the ubiquitous connectivity that underpins the IoT, we built and deployed another camera-based application. Our motivation was to replicate the work of Hideaki Kuzuoka and Saul Greenberg,[8] who explored the use of telepresence proxies, devices that incorporate cameras and displays and that are configured to share images between different physical locations.

We actually built several networked Gadgeteer devices—each incorporated the same components but they all had different form factors. We developed a simple application that periodically took a photo and uploaded

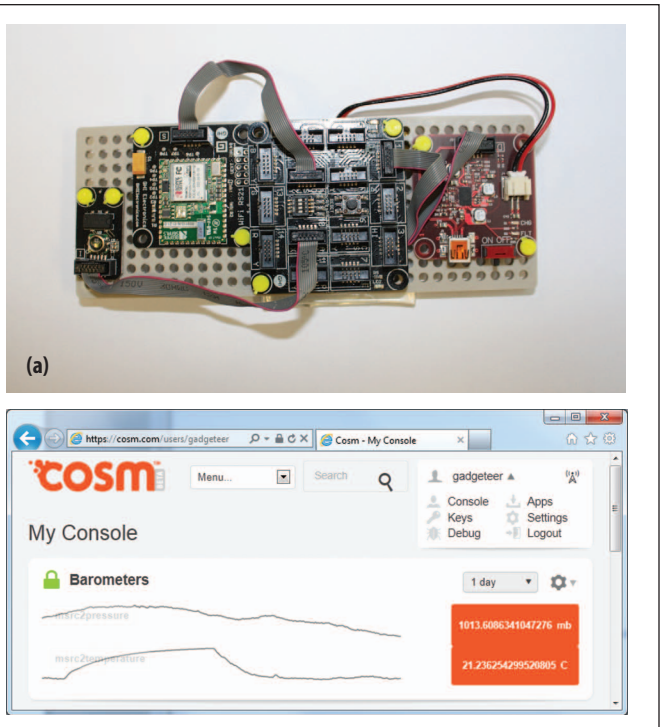


Figure 6. Prototype device created with Gadgeteer. (a) The device periodically stores temperature and pressure readings using the cosm Web service via Wi-Fi. Note that the prototype is assembled using a perforated plastic baseboard; yellow plastic pop-rivets attach the Gadgeteer modules to the baseboard. (b) Data plots collected over a 24-hour period.

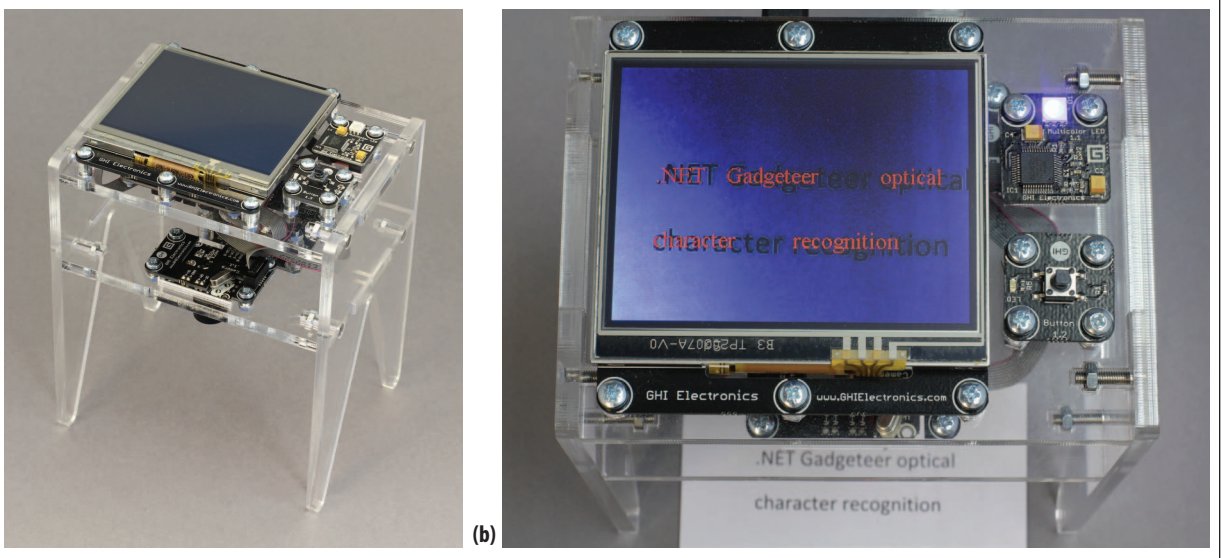


Figure 7. Optical character recognition (OCR) device. (a) The OCR device has a camera facing down toward the surface it is resting on. (b) The display on the OCR device shows an image of some printed text with the OCR'd characters correctly overlaid in red. This is implemented using the Project Hawaii cloud-based OCR service.

it to an Azure-based Web service like the Hawaii Key-Value Store. The Web service was configured to make the most recent photo from each device available to all the other telepresence devices. By displaying the latest photos from other devices in a round-robin sequence, the system maintained a level of mutual awareness between users at different physical locations. Deploying these IoT devices allowed us to experience a lightweight form of telepresence while exploring the different device form factors.

SELECTING AN IOT DEVELOPMENT PLATFORM

We used Gadgeteer to build the prototype connected devices presented here, but many factors must be considered when choosing a development tool. Key points of differentiation include performance, debugging support, cost, power consumption, and form factor. When we designed Gadgeteer, we chose a high-performance processor which supports managed code and real-time debugging. The price premium compared to tools like Arduino and mbed is modest. Power consumption was not a focus during the initial development of the Gadgeteer platform, but we are currently exploring this topic. Flexibility over form factor, however, was a central consideration when we were designing Gadgeteer. It influenced our decision to add cables for interconnecting modules as opposed to a "stacking" approach, in part because it supports using a variety of physical prototyping approaches.

One premise behind the Gadgeteer software stack is that concise code empowers less-experienced users to create useful applications and, at the same time, allows seasoned developers to build embedded-device prototypes more quickly. The anecdotal evidence we have collected at various Gadgeteer workshops bears out this intuition: users report a low hurdle for creating simple projects,[7] yet they can build relatively sophisticated prototypes of connected devices.[9] Indeed, our experience shows that the simplicity of hardware and software development with Gadgeteer inspires students as young as age 13 to engage with the platform,[10] and this may ultimately prove valuable for educating a future generation of IoT developers.

Of course, other platforms will also continue to be relevant. For example, professionals and hobbyists alike can leverage a growing set of samples and libraries for connecting a networked Arduino to online services, while educators have access to platforms like the Open University's SenseBoard (http://sense.open.ac.uk), which has already been used to teach the IoT concept.[11] In the future, it might even be possible to use accessible development environments like Scratch (http://scratch.mit.edu) to create IoT devices and services. Community support is a key element of any modern development platform. Tools like Arduino, mbed, and Gadgeteer have online forums which provide a mechanism for both new and experienced users to pose questions, exchange experiences, and share code.

No matter which tools are used for prototyping, eventually it becomes necessary to build and deploy a greater number of devices, either for larger-scale deployments or ultimately for mass production. In our work so far, we have

```
void ProgramStarted()
{
     ethernet.UseDHCP();
     button.ButtonPressed += new Button.ButtonEventHandler(button_ButtonPressed);
     camera.PictureCaptured += new Camera.PictureCapturedEventHandler(camera_PictureCaptured);
}

void button_ButtonPressed(Button sender, Button.ButtonState state)
{
     camera.TakePicture();
}

void camera_PictureCaptured(Camera sender, GT.Picture picture)
{
     // Show the picture on the display
     display.SimpleGraphics.DisplayImage(picture.MakeBitmap(), 0, 0);

     // create and send an HTTP request which will send the picture to the Hawaii OCR service
     HttpRequest request = HttpHelper.CreateHttpPostRequest("http://157.55.188.73/OCR",
        POSTContent.CreateBinaryBasedContent(picture.PictureData), "image/jpeg");
     request.AddHeaderField("Authorization", "Basic " +
        ConvertBase64.ToBase64String(Encoding.UTF8.GetBytes("<insert your appID here>"));
     request.AddHeaderField("Cache-Control", "no-cache");
     request.ResponseReceived += new HttpRequest.ResponseHandler(request_ResponseReceived);
     request.SendRequest();
}

void request_ResponseReceived(HttpRequest sender, HttpResponse response)
{
     // for this example we just display the first OCR'ed word returned by Hawaii
     // by looking between the "<Text>" and "</Text>" tags
     int start = response.Text.IndexOf("<Text>", 0) + 6;
     int end = response.Text.IndexOf("</Text>", 0);
     display.SimpleGraphics.DisplayText(response.Text.Substring(start, end - start),
         Resources.GetFont(Resources.FontResources.NinaB), GT.Color.Red, 0, 0);
    }
```

Figure 8. This complete code example shows how to program a Gadgeteer-based embedded device to use the Project Hawaii OCR Web service. This code was written using Gadgeteer SDK v4.1.

used Gadgeteer for deployments of up to 50 instances of a given device. This has been relatively straightforward because of the ease of replication of a proven Gadgeteer design and the robustness of the assembled units. At some point, it becomes more cost-effective to move to a custom PCB, which can be made more cheaply and compactly through circuit integration. Like several of the open hardware platforms, with Gadgeteer this process is facilitated by freely available hardware designs from many manufacturers.

More recently, tools specifically designed to facilitate the mass production of connected devices have emerged, such as the ioBridge (http://www.iobridge.com) and the Electric Imp (http://electricimp.com). The latter includes a programmable processor and Wi-Fi radio in a small package, and connects to applications through a hosted Web service. In essence, these devices act as a single physical component that provides a link between a hardware interface and HTTP-based Web APIs.

As the number of network-connected devices continues to grow, it is clear that no single technology will prevail—the IoT's success is inherently tied to heterogeneous devices, protocols, services, and applications. Many open questions still remain within this broad scope, and the tools presented here will not necessarily resolve these issues directly. However, as the IoT vision gradually becomes a reality, using connected-device prototypes to explore the design space will be important. We imagine that developers, researchers, designers, and hobbyists will conceive and build these prototypes and that tools like Gadgeteer will be valuable in this regard.

Although some applications will always be outside the scope of a rapid-prototyping toolkit, we nonetheless anticipate that those working in this exciting field will be able to build on the experiences and examples reported here to explore relevant issues and prototype new applications quickly, and in doing so bring the IoT vision ever closer to reality.

References

1. "Rise of the Embedded Internet," Intel embedded processors white paper, 2009; http://download.intel.com/embedded/15billion/applications/pdf/322202.pdf.
2. Casaleggio Associati, "The Evolution of Internet of Things," Feb. 2011; http://www.casaleggio.it/pubblicazioni/Focus_internet_of_things_v1.81%20-%20eng.pdf.
3. C.A. Valhouli, "The Internet of Things: Networked Objects and Smart Devices," *The Hammersmith Group Research Report*, Feb. 2010; http://thehammersmithgroup.com/images/reports/networked_objects.pdf.
4. N. Villar et al., ".NET Gadgeteer: A Platform for Custom Devices," *Proc. Pervasive 2012, Lecture Notes in Computer Science*, Springer, June 2012, pp. 216-233.
5. M. Banzi, *Getting Started with Arduino*, O'Reilly, 2008.
6. L. Richardson and S. Ruby, *RESTful Web Services*, O'Reilly, 2007.
7. N. Villar, J. Scott, and S. Hodges, "Prototyping with Microsoft .NET Gadgeteer," *Proc. 5th Int'l Conf. Tangible, Embedded and Embodied Interaction* (TEI 11), ACM, 2011, pp. 377-380
8. H. Kuzuoka and S. Greenberg, "Mediating Awareness and Communication through Digital but Physical Surrogates," *Proc. Conf. Human Factors in Computing Systems* (CHI 99), ACM, 1999, pp. 11-12.
9. P. Barden et al., "Telematic Dinner Party: Designing for Togetherness through Play and Performance," *Proc. Designing Interactive Systems Conf.* (DIS 12), ACM, 2012, pp. 38-47.
10. S. Hodges et al., ".NET Gadgeteer: Experiences with a New Platform for K-12 Computer Science Education," *Proc. 44th SIGCSE Tech. Symp. Computer Science Education,* ACM, 2013 (to appear).
11. G. Korteum et al., "Educating the Internet-of-Things Generation," *Computer*, Feb. 2013, pp. 53-61.

Acknowledgments

The authors acknowledge the Project Hawaii team behind the Hawaii Web services described in this article, and the extensive community of people who have contributed to the development of the .NET Gadgeteer prototyping system. We also thank the reviewers for their valuable feedback.

Steve Hodges *is a principal hardware engineer at Microsoft Research Cambridge, UK, where he leads the Sensors and Devices Research Group. He is also a Visiting Professor at the School of Computing Science, Newcastle University, UK. His research interests include novel electronic devices and new technologies and techniques for interaction. Hodges received a PhD in computer vision and robotics from the University of Cambridge. He is a senior member of IEEE and a member of ACM. Contact him at shodges@microsoft.com.*

Stuart Taylor *is a research software development engineer at Microsoft Research Cambridge, UK. He has extensive experience in developing new hardware and software technologies in an industrial research environment. Taylor received an MSc in computing science from the University of London. Contact him at stuart@microsoft.com.*

Nicolas Villar *is a researcher at Microsoft Research Cambridge, UK, where he focuses on the development of new hardware platforms and tools to enable technical innovation. Villar received a PhD in ubiquitous computing from the University of Lancaster, UK. Contact him at nvillar@microsoft.com.*

James Scott *is a researcher in the Sensors and Devices Group at Microsoft Research Cambridge, UK. His research interests span a wide range of topics in ubiquitous and pervasive computing, and include novel sensors and devices, rapid prototyping, wireless and mobile networking, and security and privacy. Scott received a PhD in communications engineering from the University of Cambridge. He is a senior member of ACM and a member of IEEE. Contact him at jws@microsoft.com.*

Dominik Bial *is a research assistant and PhD student at Paluno, the Ruhr Institute for Software Technology at the University of Duisburg-Essen, Germany, where he also received an MSc in software systems engineering. His research interests include future appliances and systems, software engineering, and the future Internet, especially the Internet of Things. Contact him at dominik.bial@stud.uni-due.de.*

Patrick Tobias Fischer *is a PhD student in human-computer interaction at the University of Strathclyde, Glasgow, UK. His research focuses on situated public interfaces in urban environments that have performative aspects. Tobias Fischer received an MSc in novel input technologies from the Cologne University of Applied Sciences, Germany. Contact him at fischer@cis.strath.ac.uk.*