Plug-and-play physical computing with Jacdac

JAMES DEVINE*, Microsoft, UK
MICHAL MOSKAL*, Microsoft, USA
PELI DE HALLEUX*, Microsoft, USA
THOMAS BALL*, Microsoft, USA
STEVE HODGES*, Microsoft, UK
GABRIELE D'AMONE, Microsoft, USA
DAVID GAKURE, Microsoft, USA
JOE FINNEY, Lancaster University, UK
LORRAINE UNDERWOOD, Lancaster University, UK
KOBI HARTLEY, Lancaster University, UK
PAUL KOS, Microsoft, China
MATT OPPENHEIM, Lancaster University, UK

Abstract Physical computing is becoming mainstream. More people than ever—from artists, makers and entrepreneurs to educators and students—are connecting microcontrollers with sensors and actuators to create new interactive devices. However, physical computing still presents many challenges and demands many skills, spanning electronics, low-level protocols, and software—road blocks that reduce participation. While USB has made connecting peripherals to a personal computing device (PC) trivial, USB components are expensive and require a PC to operate. This makes USB impractical for many physical computing scenarios where cost, size and low power operation are often important.

We introduce Jacdac, an open-source hardware and software platform that brings the user experience of USB to physical computing at a fraction of the cost and resource requirements. It combines an intuitive connector, standardized hardware and software interfaces, a simple bus-based protocol that runs on virtually any microcontroller, and integration with Microsoft MakeCode for the micro:bit. In this way, Jacdac provides a low-cost, extensible web-based plug-and-play experience for physical computing. We evaluated Jacdac in an international hackathon involving over 80 participants who built functional artifacts with Jacdac. Interviews reveal that users find Jacdac intuitive and appreciate the plug-and-play experience where sensors and actuators are automatically discovered by the web browser in real-time.

CCS Concepts: • **Human-centered computing → Interactive systems and tools**; • **Hardware → Communication hardware, interfaces and storage**; • **Computer systems organization → Embedded systems**.

Additional Key Words and Phrases: physical computing, plug-and-play, protocol, microcontrollers, prototyping

ACM Reference Format:
James Devine, Michal Moskal, Peli de Halleux, Thomas Ball, Steve Hodges, Gabriele D'Amone, David Gakure, Joe Finney, Lorraine Underwood, Kobi Hartley, Paul Kos, and Matt Oppenheim. 2022. Plug-and-play physical computing with Jacdac. In . ACM, New York, NY, USA, 30 pages. https://doi.org/10.1145/1122445.1122456

*These authors contributed equally to this work.

IMWUT, Proc. ACM Interact. Mob. Wearable Ubiquitous Technol., 2022

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

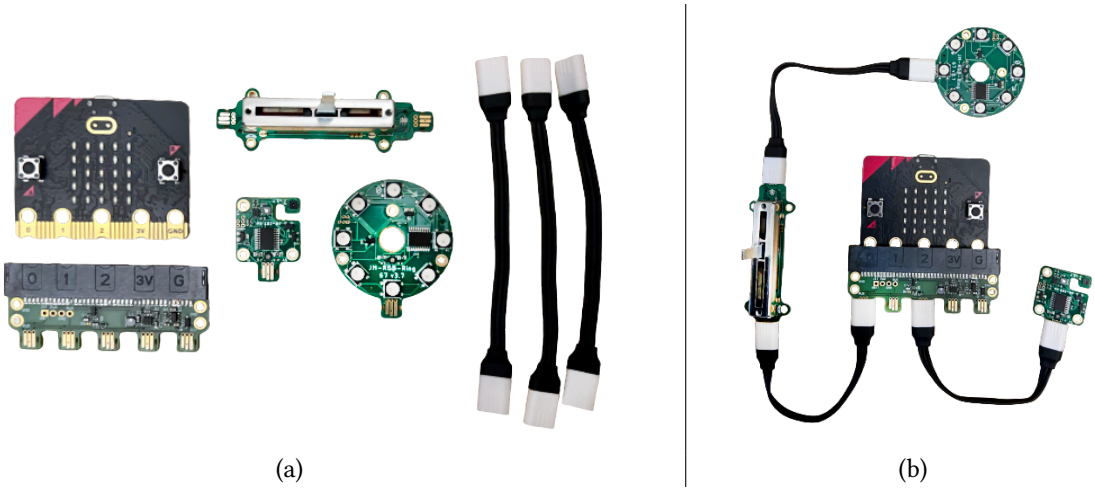


Fig. 1. Jacdac is a new open-source hardware and software platform, inspired by USB but implemented more simply and cheaply to support plug-and-play physical computing: (a) A BBC micro:bit (top left), a micro:bit Jacdac adaptor (bottom-left), three Jacdac modules—slider, CO_2 sensor and LED ring (middle), and three Jacdac cables (right); (b) The micro:bit, Jacdac adaptor, and three modules, connected together to create a self-contained and low power CO_2 monitor that can be battery-operated; note that the slider and LED ring are daisy-chained.

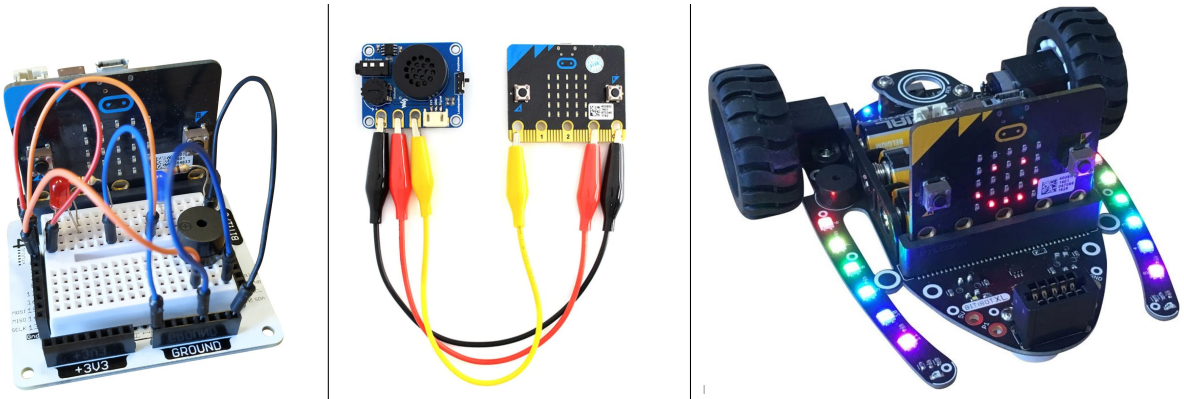
1 INTRODUCTION

Physical computing refers to the creation and use of interactive digital systems that sense and respond to the world around them [18]. Typically, sensors, actuators and communications modules are combined with a microcontroller and power source to map sensed inputs into the realms of lighting, sound, electro-mechanical actuation, and/or to pass information onto other digital systems. Physical computing builds on a wide range of disciplines including electrical engineering, electronics, mechatronics, robotics, computer science and software development. It commonly involves the exploration of novel devices and the software that powers them through an iterative, creative and experimental process.

Since the turn of the century, the process of building physical computing systems has evolved rapidly. This has been driven by a "pull" from users, many of whom are steeped in the creative maker movement [11], while others are increasingly using physical computing in the classroom [18]. At the same time we have witnessed a technology "push": the symbiotic development of physical computing technology in both academia and industry has resulted in numerous tools and platforms that simplify and accelerate the development of a physical computing solution [22]. In particular, a variety of tangible, embedded microcontroller-class devices targeted at students and hobbyists are available in the market, with three of the most well-established being Arduino [5, 36], Raspberry Pi [7] and the BBC micro:bit [2], with total sales currently estimated at 10 million, 40 million and 6 million units respectively [2, 38, 41].

1.1 Physical Computing is Intrinsically an Extensible Experience

Despite having different features, physical computing platforms all offer a hybrid and extensible experience that cuts across hardware and software. From a hardware perspective, a key element of physical computing involves leveraging a variety of sensors and actuators to build an interactive system. Even physical computing devices like the micro:bit that have basic sensors and actuators built-in often require the addition of external hardware. Three



Level 1 Level 2 Level 3

Fig. 2. The three paradigms for prototyping with physical computing identified by Lambrichts et al. [22] in relation to the BBC micro:bit. Level 1: micro:bit connected to discrete electronic components mounted in a solderless breadboard. Level 2: A peripheral connected to the micro:bit using individual wires. Level 3: micro:bit plugged directly into the ready-to-use Bitbot wheeled robot accessory.

paradigms for adding hardware—that also exhibit increasing levels of abstraction—have become established in physical computing. Lambrichts et al. refer to these as Levels 1 to 3 [22]:

- **Level 1:** Connecting discrete external electronic components, often mounted in a solderless breadboard, via individual wires.
- **Level 2:** Connecting modules and breakout boards that integrate the support circuitry for a given peripheral, again via individual wires.
- **Level 3:** Attaching ready-to-use accessories, often in the form of shields, HATs or capes, that are purpose-built for a particular platform.

Moving from Levels 1 to 3 results in additional layers of abstraction in the underlying electronic circuitry. This reduces complexity, making the development process quicker, easier and less error-prone [22]. Figure 2 shows how the micro:bit can be used in these three different ways.

Although the physicality of device hardware is naturally a key element of physical computing, software plays a vital role too: a program that encapsulates the desired device behavior must be created. Users of established physical computing hardware platforms usually have a choice of development environments, ranging from low-level programming in a compiled language like C, through high-level interpreted languages like Python, to 'no code' visual programming solutions like Scratch [32] and MakeCode [10], which are available via a web browser. These approaches broadly represent increasing levels of abstraction, and accrue to the goals of making coding more instant, intuitive and convenient, while providing extensibility and a sustaining experience.

1.2 The Limitations of Physical Computing in Practice

In spite of the flexible nature of physical computing—or perhaps because of it—there are many drawbacks. Working with discrete components (Level 1) certainly provides a great deal of extensibility, but is time-consuming and error-prone [22]. Using purpose-built physical computing accessories (Level 3) overcomes this by providing an intuitive, convenient and reliable way to add sensor and actuator hardware, and oftentimes an associated library or package provides APIs on the software side. But this usually **limits composability**, i.e. in many cases it's only possible to use one accessory at any time, which restricts what can be built. The use of electronic

modules and breakout boards (Level 2) provides a useful and popular middle ground, but still falls short of many desirable characteristics for physical computing.

No matter which of these three paradigms is adopted, there are additional drawbacks. Peripherals are not usually automatically detected; instead, the user has to **manually configure** the programming environment or the program itself to define which are in use, creating an opportunity for errors to creep in. Next is the problem of **inconsistent APIs**: vendors often design and implement APIs differently even for functionally similar peripherals, which adds confusion and complexity. And finally, even when a physical computing platform supports the notion of simulation, as with Makecode for example, **simulation is limited**. It's not possible to simulate the operation of each and every peripheral.

There are some specific counter-examples to the drawbacks highlighted above. For example, some Arduino shields support composability through stacking; Raspberry Pi includes provision for HATs that identify themselves; and MakeCode can simulate a handful of peripherals. But these are largely infrequent exceptions.

1.3 Adding Abstraction for True Plug-and-play Physical Computing

We present Jacdac, an open-source hardware and software platform for plug-and-play physical computing that introduces standardized services for peripherals like sensors and actuators without significantly increasing the size, cost or flexibility of components. This addresses many of the limitations described in the previous section.

As illustrated by the CO_2 alarm shown in Figure 1—an example we will return to later in this paper—Jacdac's novel plug-and-play hardware modules can be connected in any topology using purpose-built Jacdac cables that provide both power and connectivity. Each Jacdac device describes itself via a set of standardized services over the Jacdac bus using a packet-based protocol. A layered yet efficient protocol allows Jacdac to be implemented on remarkably cheap microcontrollers (down to US$0.03) and potentially directly in silicon. Each of the three modules in Figure 1(a) includes a dedicated microcontroller with firmware that implements the Jacdac protocol and exposes the module's hardware via standardized Jacdac services over the Jacdac bus.

A TypeScript implementation of the Jacdac protocol allows web apps to join the Jacdac bus and communicate with Jacdac modules over WebUSB. A host of Jacdac web-based components for working with Jacdac hardware and services, including hardware simulators and on-screen digital twins [16], sit on top of the Jacdac TypeScript runtime, enabling it to be readily extended, leveraging numerous web frameworks and platforms.

To demonstrate and evaluate the potential of Jacdac, we have integrated it with micro:bit hardware and the Microsoft MakeCode programming environment. MakeCode was a natural choice because its web-based architecture, coupled with its open source license, provide a readily-extensible sandbox. It also facilitates evaluation of Jacdac by teachers as well as makers.

In the course of our research we have manufactured over 2000 Jacdac modules, representing around 40 different designs, along with 1000 Jacdac cables. These were used to create Jacdac kits that we distributed during an 80-participant one-week international hackathon where MakeCode novices used Jacdac to build a variety of physical computing devices for accessibility applications. We conducted five follow-up interviews with participants to obtain their firsthand experiences with the Jacdac platform. We also interviewed a panel of four educators, experienced MakeCode users, to gain insights into the potential of Jacdac in a classroom environment.

The core contribution of our work is to show that it is possible to leverage a service-based architecture, implemented on very low cost microcontrollers, to ease the construction of physical computing systems. A second major contribution is showing how web technologies and platforms can support the programming and debugging of composable physical computing systems, without the need for installation of specialized toolchains. As of the publication of this paper, Jacdac hardware is commercially available; more information about the open source Jacdac platform can be found at https://aka.ms/jacdac

We believe that Jacdac can help the full gamut of physical computing users, from beginners to more experienced creators, from all backgrounds including students, educators, makers, digital artists, and citizen developers, targeting a range of applications from interactive art to IoT-like home automation solutions. In the rest of this paper, we report on the design and our evaluation of Jacdac.

2 RELATED WORK

This section starts by analyzing the properties of USB—a protocol that is not often associated with physical computing, but which enables the plug-and-play composition of hardware. We then contrast USB with the wired protocols and physical interfaces commonly used in physical computing before looking at how these enable physical computing experiences for makers and novices. Our analysis informs the design goals for Jacdac, described in the following section.

2.1 Plug-and-play with USB

The Universal Serial Bus (USB) allows people to connect hardware to a PC really easily—plug a mouse in, for example, and it's instantly recognised and ready to use. USB has several properties that enable this:

(1) *Cables and connectors*: The USB cable and connector system—especially with the emergence of the reversible USB-C connector—makes connecting USB devices together and powering them very straightforward.
(2) *Dynamic composition*: Unlike many other wired protocols, USB is designed so that users can connect devices at any time. When a user connects a device, its properties are automatically detected by the host. The protocol is designed to recover from, or at the very least indicate, errors to the user.
(3) *Abstraction*: Common devices share a common driver interface so that users can expect a mouse, for example, to work with any personal computer without installing any additional software.

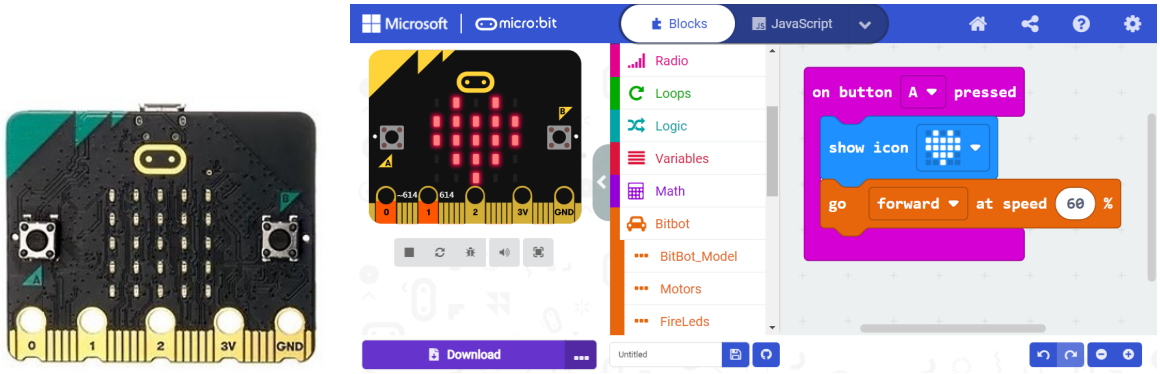
USB is not widely used in physical computing scenarios. Although described as a bus, USB connections are essentially point-to-point: a host is always required to direct communication and intelligent hubs are needed to connect USB devices together. Also, USB cables and connectors are more bulky and rigid than conventional prototyping wires. These constraints all conspire to increase cost, size and power consumption, which collectively limit the situations where USB is suitable for physical computing.

2.2 Standardized Communications Protocols

Several wired protocols that are relatively simple to implement in silicon form the basis of MCU-to-peripheral communication both within printed circuit boards (PCBs) and between PCBs. RS232 [35] is an asynchronous protocol that does not require a clock line to be shared between devices. Instead, devices agree on a communication rate ahead of time and decode data according to that rate. RS232 communications can be implemented in full-duplex mode using three wires (one data line for each direction of communication and one for ground), or in half-duplex mode with two wires (one shared data line and one ground).

I2C and SPI are synchronous protocols [24], where all devices share a clock line. At any point in time, communication is directed by a single device that defines the communication rate and initiates all data transfers. Data is written to or read from peripherals using a register abstraction; registers are defined on a per-implementation basis, so it is not common for similar register definitions to be shared by two peripherals even if they have very similar functionality (e.g. two accelerometers). This results in a need for manual configuration and inconsistent APIs. Neither protocol supports the dynamic connection of additional peripherals; connecting a peripheral dynamically can halt communications.

OneWire [3], CAN [33, 37] and MIDI [25] are other well-known wired protocols. OneWire, which uses a single wire to provide power and to communicate, works at a data rate of 16.3 kbps. However, its dynamic device discovery and low complexity do enable simple plug-and-play physical computing applications [39]. CAN



(a) (b)

Fig. 3. (a) The BBC micro:bit. (b) The MakeCode web app for the micro:bit. The MakeCode simulator provides virtual versions of most micro:bit features, allowing functional testing/debugging of programs in the browser before deploying to real hardware. A software extension for the 'Bitbot' accessory from Figure 2 has been added to the MakeCode toolbox.

operates as a broadcast bus and allows multi-host communication, up to 1 Mbps in high speed mode, or 125 kbps in low speed mode. Because of its reliability, CAN is the de facto communication protocol for critical control systems but to our knowledge is not used in physical computing. MIDI, a communications protocol for musical devices, was successfully re-purposed for physical computing in the Bloctopus project [34].

2.3 Standardized Physical Interfaces

Seeed's Grove, Adafruit's Stemma, and SparkFun's Qwiic [6, 22] are modular solutions with broadly similar implementations. Each ecosystem aligns on standard cabling and connectors and allows multiple peripherals to be interconnected with common physical computing platforms. Qwiic modules adopt a four pin connector and four wire cable, using I2C for communication. The Stemma ecosystem takes a similar approach, while Stemma QT provides a smaller connector (like Qwiic) to reduce board size. With each of these ecosystems, the protocol limitations of I2C still apply—software drivers are specific, peripherals cannot be automatically detected and must therefore be manually configured, and hot-swapping can cause problems, limiting composability.

Shields [20] and HATs [1] simplify the experience of connecting peripherals to Arduino and Raspberry Pi, respectively, by providing standardized device pinouts. Arduino shields require manual configuration whereas Pi HAT producers can optionally add a hardware component to support automatic identification and driver loading when the Raspberry Pi first boots up. Neither can be hot-swapped.

2.4 Physical Computing Tools and Platforms

Researchers and engineers have created a wide variety of physical computing tools and platforms, often building on top of the interfaces and protocols described above, abstracting away some of their shortcomings and creating a simpler user experience. It is not possible to review all of this prior work here, so instead we have called out a small number of representative products and research projects to give a flavor of what exists. For more detail we refer the reader to [22].

The many variants of Arduino and Raspberry Pi hardware and software provide a baseline of established practices for physical computing. A more recent addition to the physical computing arena is the BBC micro:bit and the popular MakeCode programming environment, shown in Figure 3. The micro:bit integrates an MCU with a handful of on-board inputs and outputs, making it easy to get started. Much like Arduino and Raspberry

Pi, additional peripherals are supported via a proprietary connector that exposes established electrical interfaces such as general purpose I/O (GPIO) pins, analog inputs, I2C, SPI and UART.

In terms of software, MakeCode runs in a web browser and supports a visual language similar to Scratch [26, 32], in addition to text-based coding in Javascript and Python, see Figure 3(b). Many web browsers now support access to USB devices connected to their host computers over WebUSB, allowing a micro:bit program, compiled to machine code in the browser by MakeCode [4], to be copied directly onto the micro:bit. Unfortunately the simulator does not support accessories and different accessory vendors have implemented inconsistent APIs.

Phidgets [15], which started as a research project and subsequently transitioned to a successful product, provides a set of fully composable hardware modules for building novel interactive interfaces to PCs. Some Phidgets modules connect directly to the PC over USB while others use a proprietary Phidgets protocol called VINT to attach to a special hub, which in turn interfaces to USB. Native USB modules can be dynamically detected for a true plug-and-play experience whereas VINT modules require manual configuration. Although Phidgets does provide hardware simulation, the use of USB and the need for a PC to run the user's program increases cost, solution size, and limits the possible application scenarios.

The Calder toolkit [23] helps guide the creation of hardware interfaces, providing simple analog and digital I/O modules to users. It is designed to be hot-pluggable and offers a number of IDE integrations for simulation. A hub acts as the main connection point for all modules. The hub propagates connection events and interactions detected by peripherals over USB to Calder applications running on a PC. The d.tools system [17] was also conceived to reduce the technical complexity of prototyping functional devices. The d.tools authoring environment provides a visual programming interface that supports drag-and-drop specification of interaction models for tangible user interfaces, allowing the quick construction of functional prototypes.

More recently, .NET Gadgeteer [19, 40] is designed to make it easier to build self-contained physical computing devices. A Gadgeteer device consists of a main processor board connected to modular peripherals and programmed via an integrated environment. Peripherals include simple digital and analog I/O to more complex sensors that rely on protocols like I2C, SPI, UART and USB to high resolution displays. Peripheral boards are connected to the main board by a cable with a non-reversible 10 pin connector. On the main board, there are multiple ports for connecting to multiple peripherals. Users must manually configure which peripherals are connected to which ports and there is no hardware simulation support.

3 JACDAC: DESIGN AND ARCHITECTURE

While several of the technologies and toolkits described in the previous section go some way towards tackling the limitations of physical computing outlined in Section 1.2, none of them provide the plug-and-play experience that we aspire to. Arguably Phidgets comes the closest, but its reliance on a PC and USB—not just for programming and debugging, but for on-going operation—is counter to the embedded, integrated, and low-cost nature of popular physical computing platforms.

We propose a new approach, which we have implemented in a platform called Jacdac. The design goals for Jacdac are inspired by a key insight from Sadler et al. [34], namely *the importance of making interfacing a one-step process*. In physical computing, this concept relates to both hardware and software. At the same time, we aspire to a solution which is realistic rather than idealistic, and as such we have an additional design goal for Jacdac around implementation cost. The three main goals that result are listed below; the rest of this section describes how Jacdac meets them:

DG1 **Easy hardware composition**: Hardware devices should be easy and intuitive to construct, with no arbitrary physical constraints—unlike many existing systems that require careful connection of individual wires to specific pins. The number of devices should not be limited and all physical topologies should be supported.

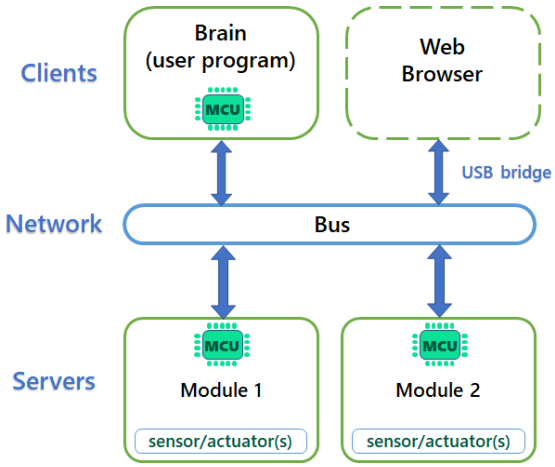


Fig. 4. A typical Jacdac configuration consists of a brain device (upper left), running a user's program on its MCU, which is a client of Jacdac services advertised by one or more Jacdac modules (bottom). Each module has its own MCU that acts as a server, exposing sensors/actuators connected to that MCU via Jacdac services. Our example CO_2 alarm from Figure 1 uses the micro:bit as its brain and connects three modules that expose services. A web browser (upper right) may join the Jacdac network via a USB/Jacdac bridge for purposes of configuring, programming, monitoring and debugging. In Figure 1 the micro:bit also acts at the USB/Jacdac bridge.

DG2 **Plug-and-play software abstraction**: Dynamic addition and removal of devices should be possible, along with automatic discovery of the functionality and capability of each connected device. APIs to devices should be common and standardized, giving the flexibility to substitute one device with another of similar functionality. Software simulation of hardware components (digital twins) should also be supported.

DG3 **Low cost**: It should be possible to achieve the above features using components that are economically viable in commercial implementations. The ideal solution should minimise barriers to adoption and have a range of implementation options where sophisticated functionality can be traded-off against cost.

Unlike USB and many other embedded systems communications protocols, Jacdac inherently supports a distributed systems architecture made up of one or more clients and one or more servers, communicating via services over a shared bus. This architecture is naturally composable and scalable (DG1), following the same principles used in established client/server solutions.

As shown in Figure 4, a Jacdac *module* (or *server*) is simply a Jacdac device that presents one or more Jacdac services on the bus—for example a push button, temperature sensor or LED. A server might provide multiple different services. For example the CO_2 module in the alarm example shown in Figure 1 actually provides four separate services: CO_2, total volatile organic compound (TVOC), temperature and humidity. Services can be used to query or control both physical hardware resources and purely virtual resources, e.g. the state of a video game. Multiple similar servers—for example multiple identical buttons—may be connected to the same bus; in this case a *role manager* service helps name modules with identical service profiles.

Services provide abstract, standardized interfaces that support plug-and-play dynamism. This means that devices with different hardware implementations but the same overall functionality can replace one another with no other changes needed (DG2). Jacdac services are a contract between Jacdac device manufacturers and the clients (and users) of these devices. As such, it is important to have a stable public *service catalog* that can be used by Jacdac software tooling, for example to generate the client/server bindings. Device manufacturers can submit

new services to the catalog, and some tools support import of local service specifications. At run time, devices advertise their services to others on the bus (DG2).

A Jacdac *client* is a device, again physical or virtual, that consumes one or more services. Often a client can be reconfigured or re-programmed to alter its behavior to suit a particular application. There may be multiple clients on the same Jacdac bus, in which case they may consume exactly the same set of services or different combinations of services. Sometimes a client will also provide services itself.

The following two sections describe the major technical contributions of Jacdac: (1) a set of easily composed hardware elements, including connectors, cables and devices; (2) a plug-and-play software abstraction, including services and underlying protocols. These contributions are somewhat independent—the service abstraction could run over different protocols, and the protocol could use different hardware—but they were designed to work well together. The Jacdac hardware and software platform is open source on GitHub[1].

4 JACDAC: HARDWARE

Jacdac's client/server architecture naturally results in several categories of device hardware:

- **Modules (servers)**. A *module* can be thought of as a very small server that makes one or more sensors, actuators or other peripherals available via one or more services, respectively, on the Jacdac bus. Figure 1 shows three such modules: a slider, an LED ring, and a CO_2 module (which makes available three other sensors in addition to the CO_2 sensor).
- **Brains (clients)**. A *brain* runs application code on an MCU and makes use of services that are available on the bus to achieve the required end-user scenario. Brains can either be *dedicated* or can be *integrated* with on-board peripherals. The micro:bit in Figure 1 is the brain of that device.
- **Adaptors**. *Adaptors* can provide a convenient way to add functionality to existing electronics prototyping platforms such as micro:bit, Raspberry Pi and Arduino. Figure 1 shows an adaptor for the micro:bit. We also have built other adaptors: one for the Raspberry Pi and another for USB. The latter allows the Jacdac bus to be connected to a laptop or desktop, whereupon browser-based applications can connect to Jacdac devices via WebUSB or WebSerial. Jacdac *hubs* are a special type of adaptor that provide additional Jacdac sockets.
- **Power supplies**. The Jacdac bus supports power sharing in addition to communications. Jacdac devices may be designed to consume power from the bus, to provide power to the bus, to be independently powered, or to alternate between these modes. Jacdac power supplies are often integrated in brains or adaptors.

We next consider how Jacdac devices can be connected together physically.

4.1 Jacdac Bus and PCB-based Edge Connector

At the electrical level, Jacdac relies on a 3-wire bus for power delivery and data transfer. One wire is used for ground, one for data and one for power. In the simplest configuration, all devices on the Jacdac bus are connected directly to these three wires.

Jacdac provides a purpose-built PCB-based connector that is optimized for cost, performance and user experience (DG1, DG3). As shown in Figure 5, one mating half of the connector takes the form of a double-sided three-conductor edge connector that is incorporated into the PCB design of Jacdac devices. The edge connector is trivial to integrate into existing PCB designs and introduces no component dependency for manufacture (DG3).

To better support dynamic plug-and-play (DG2), the data pin is shorter than the power and ground pins (see Figure 5)—this ensures that a Jacdac device is powered before electrical contact with the data line is made, minimizing the chance that hot plug/unplug operations will disrupt the bus. For easier composition (DG1), the edge connector is designed to be reversible so that cables can be plugged in either way up; the order of pins is

[1]See https://microsoft.github.io/jacdac-docs/github/.

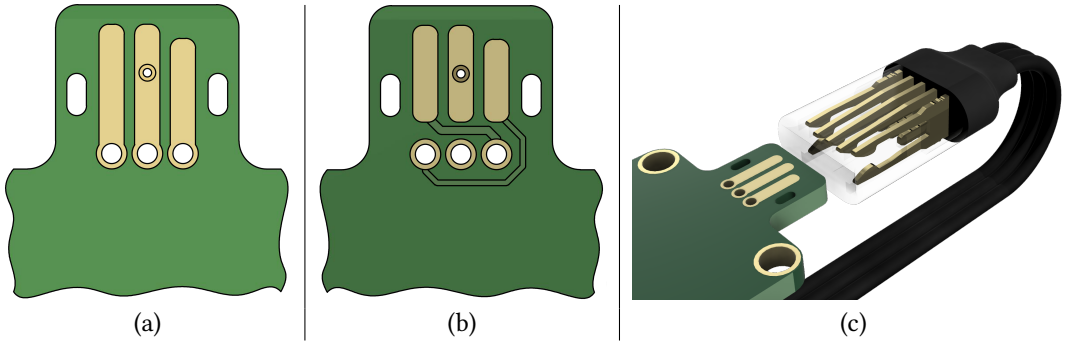


Fig. 5. PCB-based double-sided Jacdac edge and cable connectors. (a) The “top” of the edge connector has three pins, from left to right - 1) power, 2) ground, and 3) data. (b) The “bottom” of the edge connector also has three pins, from left to right: power, ground, and data - see text for more detail about wiring between top and bottom sides. (c) A 3D model of the cable connector, with housing made transparent to show internal metal fingers. See Figure 1 for photos of real examples.

flipped on the bottom side of the edge connector to support this. Mechanical slots on either side of the electrical contacts support connector attachment.

4.2 Jacdac Cable Connector

As shown in Figure 5(c), a purpose-built Jacdac cable connector interfaces with the Jacdac PCB edge connector described above. The cable connector is reversible but internally it only requires three sprung contacts to connect electrically due to the double-sided design of the mating Jacdac PCB edge connector. This keeps the cost of the cable low (DG3). Each contact has a maximum resistance of 30mΩ and can carry up to 1A.

Four sprung mechanical hooks inside the cable connector engage with the slots on the edge connector. The purpose of these mechanical hooks is two-fold: (1) to ensure mechanical and therefore electrical stability and (2) to provide "click-like" haptic feedback. The click makes the user aware that the cable is fully and correctly seated (DG1). The mating and unmating force of the connector is specified at 5-10N.

Figure 1(a) shows three 10cm Jacdac cables (we’ve produced cables up to 150cm; longer cables are also possible). Figure 1(b) shows three modules connected to a micro:bit Jacdac adaptor. Note that the cables can be plugged into any of the five available edge connectors on the adaptor—operationally they are all equivalent as they share the same three traces on the PCB. The edge connectors at each end of the slider module also use the same three traces, enabling daisy chaining (DG1).

4.3 UART-based Communication

The Jacdac network layer builds on the foundations of RS232, operating in half-duplex mode to create a shared bus using just three wires (ground, power and data). Jacdac defines a fixed frequency and packet structure and adds bus arbitration so that any device can initiate communications (DG2). The process of bus arbitration is simple to implement and is achieved through a Jacdac device bringing the bus from the default logical one (3.3V) to a logical zero (0V) for a short period at the beginning and end of transmissions.

Jacdac communications can be implemented on nearly all MCUs, including very low cost ones (DG3). Key requirements are:

- the ability to transmit and receive data on a single wire 10 bits at a time (1 start bit, 8 data bits, 1 stop bit);
- transmission at 1Mbaud;
- MCU control of the single wire, optionally with interrupt capabilities;

- the ability to approximate time, either through instruction counting or a hardware timer.

Note that built-in UART hardware is not mandatory; we have, for example, implemented Jacdac on an US$0.03 8-bit MCU via bit-banging. Similarly, if an MCU has separate UART transmit and receive pins, these can usually be combined off-chip to operate in half-duplex mode. A simple resistor-capacitor-ferrite filter is used to reduce electromagnetic emissions to meet certification requirements and a diode clamp protects the MCU from unexpected transients.

4.4 Device Identification

Jacdac device identifiers are 64-bits in length and are used to determine the sending or receiving device, and for devices to identify one another on the bus. Each hardware device must be assigned (or assign itself) a 64-bit identifier, which must remain constant once assigned. As long as identifiers are generated with appropriate entropy (i.e., using a truly random number generator), the chance of identifier collision is astronomically small.

4.5 Power Sharing

The Jacdac bus supports power sharing; devices that have their own power source—perhaps via a USB connector or from a built-in battery—can share power onto the bus. *Power providers* allow other devices (e.g., sensors, LEDs) to act as *power consumers* that are powered directly from the bus. Power providers fall into two categories: *low current providers* supply up to 100mA at 3.7V-5.2V and only require a small and cheap current limiting chip; *high current providers* supply up to 1A and must run a Jacdac power provider service that ensures only one is active at any one time, so as to respect the maximum current capabilities of Jacdac cables. Power consumers are even cheaper to implement, requiring little more than a 3.3V low-dropout linear regulator IC to power their circuitry.

4.6 Prototype Hardware

We designed and produced over 40 distinct Jacdac modules and adaptors, grouped into the following categories and shown in Figure 6:

- **Adaptors**: A micro:bit adaptor makes five Jacdac edge connectors available and is capable of low-current power sharing. We also have USB adaptors based on the STM32F4 and RP2040 MCUs that are high current power providers. The USB adaptors can also expose a HID service to the Jacdac bus through which clients can emulate HID events like mouse clicks. A 'Click' adaptor that allows the MikroElektronika Click ecosystem of modules [27] to be used with Jacdac.
- **User inputs**: These include individual buttons of various types, sliders, joysticks, rotary controls, game pads and keypads.
- **Outputs**: We have built a range of RGB LED modules for Jacdac. A single LED, ring of 8 LEDs and strip of 10 LEDs are all power consumers. Additionally, we have a module with a separate power input that interfaces to a NeoPixel strip of any length. We have also prototyped a speech synthesis module, a relay and a servo motor driver.
- **Sensors**: In addition to the combined CO_2/TVOC/temperature/humidity sensor from Figure 1, we have built a cheaper temperature and humidity sensor, a flex sensor, soil temperature and moisture sensors, and light level sensors.
- **Wireless interfaces**: We have prototyped both Bluetooth and WiFi interfaces using nRF52 and ESP32 MCUs respectively.

Each of the modules in Figure 6, except for the micro:bit adaptor, contains a low-cost MCU that interfaces with the underlying sensor/input/output and makes it available via one or more Jacdac services. In addition to an MCU, these basic modules typically only require a linear power regulator and a handful of passive components,

resulting in a total bill-of-materials (BoM) cost of as little as US$0.10 for a button or LED module in quantities of 1k units, based on pre-pandemic Shenzhen pricing.

More sophisticated services may need more expensive sensors and/or a more capable MCU. Power providers require a current limiter IC (~US$0.10 pre-pandemic) and the Jacdac micro:bit adaptor also needs a DC-DC converter (US$0.15 or less pre-pandemic). Our cheapest Jacdac brain is based on the RP2040 MCU and has a BoM cost of ~US$1.50 (pre-pandemic Shenzhen pricing).

The CO_2 alarm example from Figure 1 leverages our micro:bit adaptor, one user input (a slider), one output (ring of 8 LEDs) and one sensor. Note that the slider has two edge connectors, allowing it to be daisy-chained, while the other modules have just a single edge connector.

5 JACDAC: SOFTWARE

Jacdac devices share resources with other devices on the bus through services. Services provide abstract, standardized interfaces that can be used to interact with physical hardware resources using a client/server communication mechanism [28] similar to the service-oriented architectures popular in software engineering [12]. This supports substitutability and enables the creation of digital twins [16] of devices and simulators (DG2), for example via a web-based user interface. A software stack (Section 5.3) implements the Jacdac protocol for both MCUs and web browsers.

5.1 Service Specification Language

The Jacdac service specification language standardizes service interfaces and was designed to:

(1) describe the low-level interface to hardware;
(2) allow code generation for both clients and servers across a variety of languages;
(3) have an efficient runtime representation over the wire;
(4) be extensible, as the need for new services arises.

To date, we have used the specification language to create over 80 Jacdac service specifications. The specifications are used to generate a variety of artifacts, including: documentation, C header files and client libraries in a variety of languages. These services are independent of the particular underlying hardware.

We give a brief overview of the salient aspects of the service specification language via a few examples that pertain to the slider and CO_2 modules from Figure 1. Jacdac services are specified in a customized Markdown language with support for declaring registers, actions, and events—service primitives—as well as the types that parameterize these three entities.

5.1.1 A potentiometer service. Figure 7(a) presents the source text of the specification for a simple potentiometer service, implemented by the slider module from our running example in Figure 1. Every Jacdac service is uniquely identified by a 32-bit identifier (line 1), also referred to as the *service class*. Line 2 of the specification states that the potentiometer service inherits from (extends) the sensor interface, which defines a set of common registers for working with sensors. Line 3 provides some meta-data about the service: the `group` attribute tags the service with the keyword "input", for the purpose of filtering searches over services.

Every service that implements the sensor interface must declare the name and type of the read-only "reading" register, which provides the value of the sensor. Line 5 of the potentiometer specification contains this declaration, which gives the friendly name "position" to the reading register and declares that its type is an unsigned fixed point number (between 0 and 1) with 16 bits of precision (via the type `u0.16`). The unit for register is "fraction" (signified by the "/" following the type).

Figure 7(b) lists the core Jacdac types available for describing register values as well as the payloads associated with actions and events. Each core type may be optionally annotated with a unit, for which we use a subset of

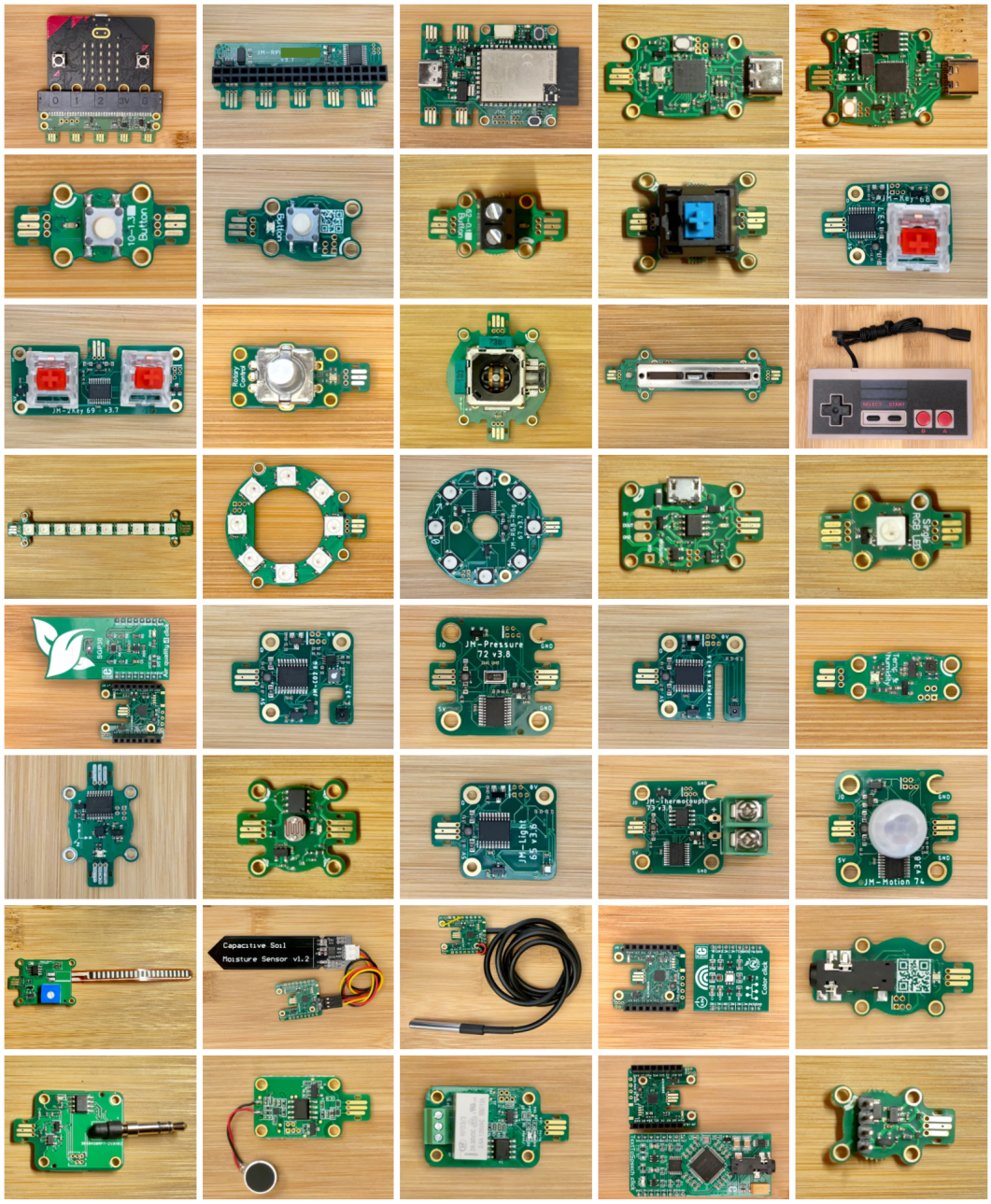


Fig. 6. Some of the Jacdac hardware we have prototyped and evaluated. Adaptors (top left) for micro:bit and Raspberry Pi. Brains (top right): ESP32 WiFi brain, STM32F4 brain and RP2040 brain. Various push-buttons (row 2), input devices (row 3) and RGB LEDs (row 4). Atmospheric sensors (row 5) including CO_2, pressure and temperature/humidity. Other sensors (rows 6 and 7) include accelerometer, light level, thermocouple, PIR motion, flex, soil moisture/temperature, color, and access switch input. Actuators (bottom row) include access switch output, haptic/vibration output, relay and speech synthesis. We have also produced a generic breakout board containing only the bus interface filter circuit (bottom right).

```
1  identifier: 0x1f274746
2  extends: _sensor
3  group: input
4
5  ro position: u0.16 / @ reading
6
7  enum Variant: u8 {
8      Slider = 1,
9      Rotary = 2,
10 }
11 const variant?: Variant @ variant
```

(a)

u8, u16, u32, u64	unsigned (1, 2, 4, and 8 bytes)
uM.N	unsigned fixed point ($M + N \in \{8, 16, 32\}$)
i8, i16, i32, i64	signed (1, 2, 4, 8 bytes)
iM.N	signed fixed point ($M + N \in \{8, 16, 32\}$)
f32, f64	IEEE float and double
b	byte buffer (until end of packet)
s	UTF-8 encoded string (until end of packet)
z	NUL-terminated UTF-8 string

(b)

Fig. 7. (a) Potentiometer sensor service written in customized Markdown. (b) The core types supported by Jacdac.

```
1   identifier: 0x169c9dc6
2   extends: _sensor
3   group: environment
4
5   ro e_CO2: u22.10 ppm { typical_min=400, typical_max=8192 } @ reading
6
7   ro e_CO2_error?: u22.10 ppm  @ reading_error
8
9   const min_e_CO2: u22.10 ppm  @ min_reading
10  const max_e_CO2: u22.10 ppm  @ max_reading
```

Fig. 8. Equivalent CO_2 sensor service.

SenML [9]. The Jacdac type system also permits a record type with named fields of core types. Via this small type system, we have been able to specify a wide range of services.

Lines 7-11 of Figure 7(a) contain information relevant to presenting the digital twin of a physical potentiometer device; via the optional `variant` constant, a device can declare that its potentiometer service corresponds to a slider (as in Figure 1) or a rotary dial.

5.1.2 A CO_2 sensor service. The potentiometer service is about the simplest sensor in the Jacdac service catalog. Figure 8 presents the specification for an equivalent CO_2 sensor supported by the sensor in our CO_2 alarm. This illustrates a few other salient aspects of the specification language. Note that line 5 of the specification declares the reading register (named `e_CO2`) to be a 32-bit unsigned fixed point with 22 bits to the left of the "." and 10 bits to the right. The units of the register are declared to be parts-per-million (via the acronym `ppm`, per SenML). This declaration also provides typical minimum and maximum values of equivalent CO_2, which is useful for simulation.

The datasheet for a sensor specifies its sensitivity, which may depend on environmental factors such as temperature, output resolution, and noise, among other characteristics. Line 7 of Figure 8 declares an optional register named `e_CO2_error` which exposes the expected error when reading the `e_CO2` register. Lines 9 and 10 provide access to the actual minimum and maximum values supported by the underlying hardware.

If a particular sensor supports configuration or other functions not specified in the standard service specification, the manufacturer should expose the standard service, and an additional so called *mix-in service* that exposes the specific functionality (with reasonable defaults, so it can be ignored by the user if so desired).

5.1.3 The control service. Every device acting as a server must run its own control service (service class 0), which emits advertisement packets roughly every 500 milliseconds. Each advertisement contains the device's unique identifier as well a list of services offered by the device (DG2); other devices on the bus can inspect the advertisement. If a client does not receive an advertisement from a device for several advertisement periods (1-2 seconds), it assumes that the device has left the bus.

5.1.4 The role manager service. Jacdac allows multiple devices that provide the same service. Consider a system that uses four slider modules to create a miniature mixing desk. The user can define four roles (or names) with the slider service class and the role manager service assigns specific devices to these, so that the program can refer to each device by its role name. The user can re-configure these assignments by interacting with the role manager service, or alternatively swap the physical devices.

5.2 Packet-based Protocol

Jacdac uses a packet-based broadcast protocol; all Jacdac devices on the same bus receive all packets. Although we previously classified Jacdac devices as "modules" (servers) and "brains" (clients), this distinction is not made at the protocol level: any device can act as both client and server, although often will assume just one of these roles.

A Jacdac packet is either a *command* or a *report*. A command is targeted at a particular device on the bus, while a report is a broadcast of information from a particular device. This distinction is necessary because a Jacdac packet has space for only one device identifier.

In general, a Jacdac server will listen on the bus for service commands that are addressed to it and respond by broadcasting a service report. So, for example, a register read command is a request to a service for the value of a particular register, while a register read report is a response from that service with the value for that register.

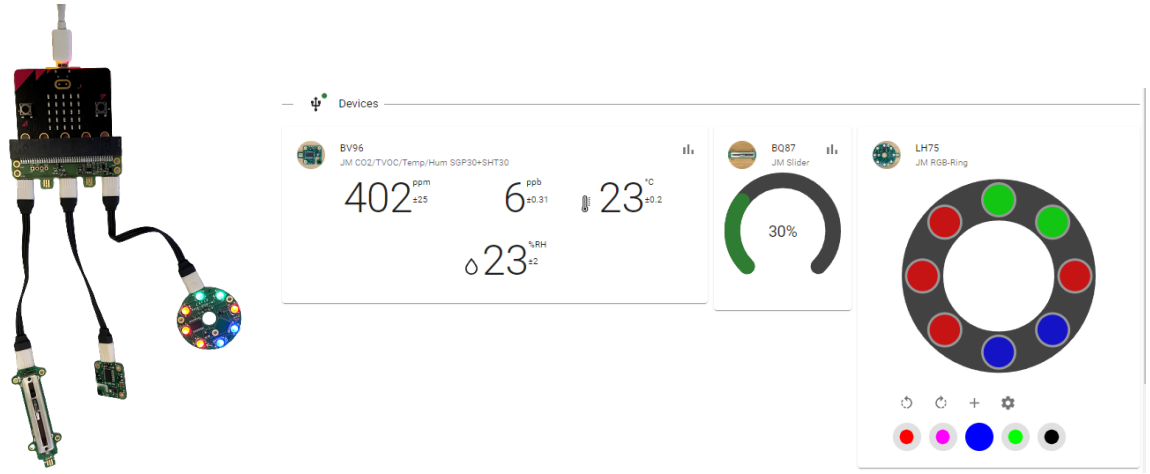
The sender of a command can request an acknowledgement; otherwise, the Jacdac protocol is similar to UDP, the user datagram protocol [30], in that no delivery guarantees are provided. Also, because a command/report pair (two packets) may be separated by other packets on the Jacdac bus, the Jacdac runtime provides support for waiting for the response (report) to a request (command). Additional TCP-like functionality [31] can be added to support reliability and ordering.

A server will also issue some reports such as events and advertisements without prompting. Since servers do not generally issue commands to other devices, they need not track the other devices that are on the bus, leading to resource savings on the server.

5.3 Implementation

The Jacdac software stack consists of firmware that runs on a variety of MCUs plus a completely separate TypeScript protocol implementation that can run in the web browser. It also has web components for digital twinning and simulation of Jacdac services.

The firmware for modules (servers) is split into two: a platform-agnostic C99 implementation of the Jacdac protocol, I2C drivers and Jacdac services; and platform-specific implementations of low-level primitives (half duplex RS232, I2C, SPI, GPIOs, etc.). We use a separate monolithic assembly language implementation of the Jacdac protocol and various services for the ultra low-cost PADAUK family of 8-bit MCUs (in particular the PADAUK PMS150C, PMS171B, and PMS131 which cost as little as US$0.03, pre-pandemic Shenzhen pricing for 1k units or more).



(a) (b)

Fig. 9. (a) A simple CO_2 alarm is comprised of a micro:bit connected to a desktop computer over USB and also to a slider, CO_2 sensor and LED ring via a Jacdac adaptor. (b) The Jacdac web-based dashboard shows digital twins of the three modules: the first two show the CO_2 reading (and the values of three other associated sensors) and the slider, while the last allows the user to set the LED ring colors and displays current state.

Servers have different resource requirements based on the services implemented. For example, the button service on an 8-bit PADAUK MCU needs 2500 bytes of ROM and 64 bytes of RAM. The C99 implementation is more complete and therefore requires more resources. For example, an implementation of a sensor service, hardware runtime, Jacdac protocol, I2C driver, and a 4kB Jacdac bootloader fits comfortably in the 32kB of ROM and 8kB RAM available on the smallest STM32G030 (US$0.51, ST Micro list price for 1k quantities).

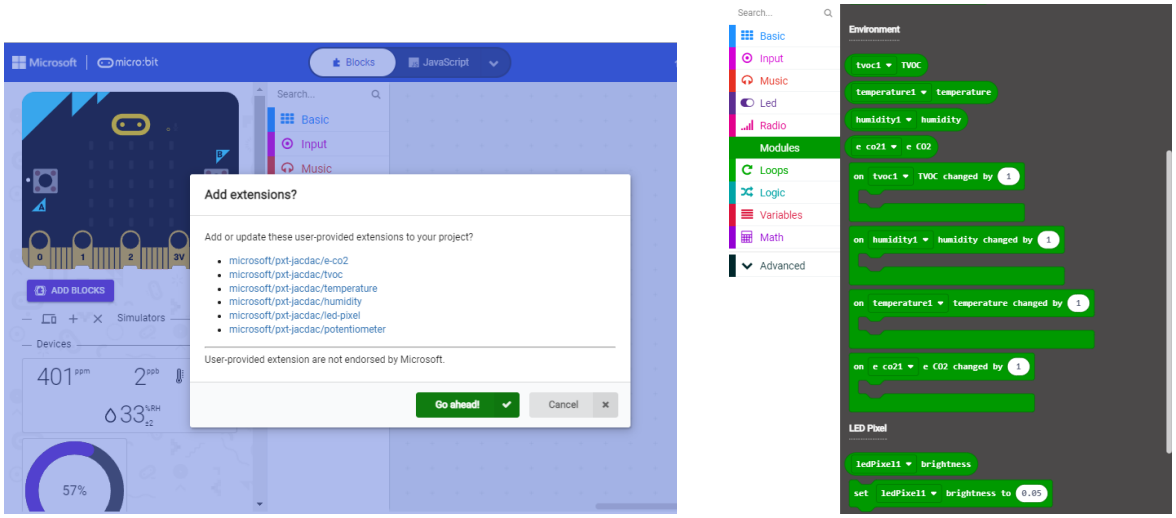
The web implementation of Jacdac also has several layers: a TypeScript implementation of the Jacdac protocol and a Jacdac object model that provides an RPC abstraction; a port of the above to Static TypeScript [4], the language supported by MakeCode; a set of React components for providing digital twins and simulators for Jacdac modules/services. The first two layers allow any Jacdac device implementing the corresponding USB interface (including the micro:bit) to talk to a web browser when connected to a USB port. Figure 9 presents this stack in action. We also implemented Jacdac for Python, .NET, and Node.Js.

6 PUTTING IT ALL TOGETHER TO MAKE A WORKING DEVICE

In this section, we revisit the running example from Figure 1 and describe how we program it to make a working device via the integration of the Jacdac platform with the MakeCode editor for the micro:bit, which forms the basis of our evaluation in the subsequent sections.

Jacdac is integrated into MakeCode as an extension, which includes the Static TypeScript implementation of Jacdac, configuration information that loads the Jacdac dashboard into the MakeCode simulator panel, and separate extensions for each Jacdac service. We created a tool that automatically translates each Jacdac service into a (separately loadable) MakeCode extension, providing high-level APIs for programming against each service. This means users can program via both visual blocks and JavaScript. For example, for each sensor service, we generate an “on change” event handler that fires when the relevant register changes by a specified amount.

As with any micro:bit accessory, to get started with Jacdac the user simply adds the top-level Jacdac extension to their MakeCode program. This activates the Jacdac dashboard and connects MakeCode to the Jacdac bus over



(a) (b)

Fig. 10. (a) Jacdac dashboard integrated in MakeCode (lower-left), with modal dialog for adding MakeCode extensions for the services advertised by the Jacdac modules; (b) partial view of blocks available for the CO_2 module (with its three other sensors), and the LED ring.

WebUSB. A new category named "Modules" is also added to the MakeCode toolbox, but no service-specific APIs are present—they are loaded on demand to keep from overwhelming the user with blocks that are not relevant to the modules they are planning to use, which in the case of Figure 1's CO_2 alarm, are a slider, an LED module and the CO_2 sensor itself.

Figure 10(a) shows how the Jacdac dashboard appears in the MakeCode micro:bit editor when the three modules shown in Figure 1(b) are plugged into the micro:bit and recognized. Digital twins of the modules are visible below the micro:bit simulator. The purple "Add Blocks" button (directly under the micro:bit simulator) allows users to add MakeCode blocks corresponding to the connected modules. When the user presses this button, MakeCode reflectively looks at the services connected to the bus, automatically finding the appropriate additional extensions, listed in the modal dialog. Figure 10(b) shows all the blocks associated with the four services of the CO_2 module, and a few of the blocks associated with the LED service. The user can now directly program against the hardware using these blocks.

The dashboard—in addition to showing the digital twin of any connected module—has the ability to run simulators for all Jacdac services. This allows the user to select a set of simulators and program against them even if the relevant hardware isn't readily available.

Figure 11 shows a block-based program that underpins our running example CO_2 alarm. It adjusts the LED ring brightness based on the slider and sets the color of the LED ring depending on value read from the CO_2 sensor. In this case, there is no actual hardware connected. Instead, simulators have been started via the dashboard, allowing the user to test their program without having to recreate the physical conditions required to trigger code. The CO_2 simulator is a great example of this—via a virtual slider the user can change the sensor value to test their program.

With MakeCode, we can compile the program to machine code in the browser and download it to the micro:bit, so that the whole system in Figure 1(b) can run standalone from battery power.

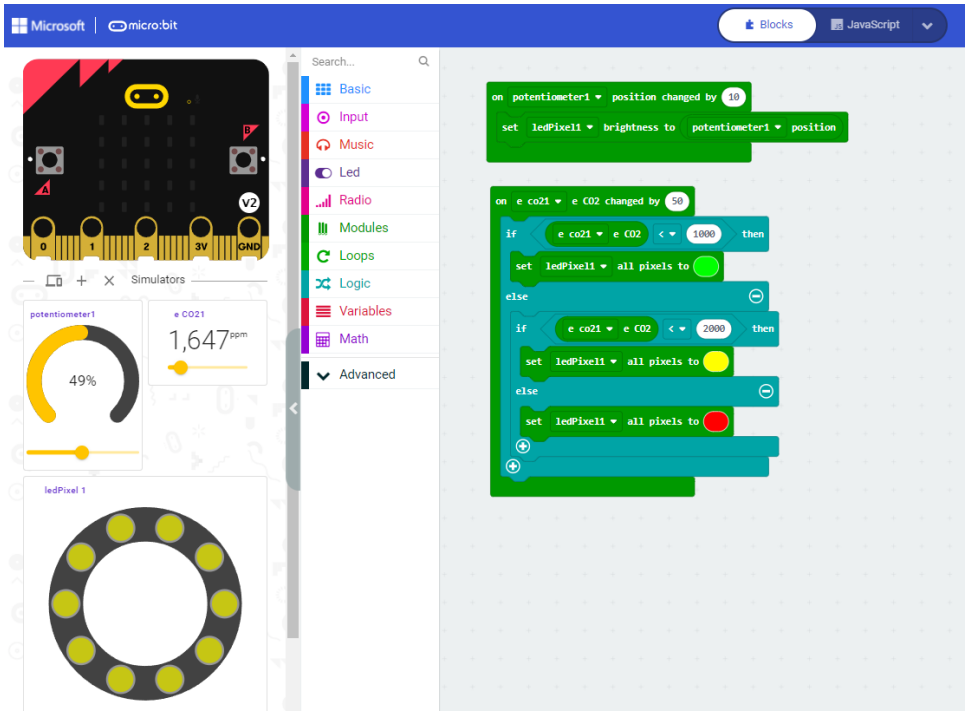


Fig. 11. A Makecode program that uses the slider, CO_2 sensor and LED ring to simulate the CO_2 alarm from Figure 1 without hardware.

7 JACDAC HACKATHON

New physical computing platforms are often evaluated in the context of a workshop led by the researchers who create them [8, 29], but due to the COVID-19 pandemic we were unable to do this for Jacdac. Instead, we took advantage of an opportunity to evaluate Jacdac in an entirely remote, international three-day hackathon with a focus on accessibility. Our primary goal was to see how easily newcomers to physical computing and MakeCode could use Jacdac to build custom devices.

Hackathon participants were self selecting and discovered Jacdac through a hackathon project page we created. Participants were from a diverse range of backgrounds: 28% software developers, 7% firmware engineers, 10% hardware engineers, and 55% from non-technical backgrounds. 47% of participants were from America, 5% from EMEA, and 48% from Asia. 62% of participants were male, 38% were female.

Participants were sent kits (see Figure 12) that included a set of Jacdac modules selected to suit planned projects in the accessibility domain, suggested by participants as part of the hackathon registration process. In total, we sent out 50 kits for use by over 80 participants (some participants were co-located) spread across timezones from GMT-8 to GMT+8. Participants grouped together into 12 timezone-compatible workstreams, and met over video conferencing. Section 10.1 in the Appendix gives details of the particpant on-boarding experience.

The following sections overview three artifacts created during the hackathon by various workstreams. The complexity of these artifacts demonstrates how, with minimal introduction, participants used Jacdac with great success.

Fig. 12. A Jacdac hardware kit for the micro:bit V2, as used by hackathon participants.

7.1 Video-conferencing Accessibility Controller (VACO)

The VACO workstream focused on making the user interface of a video conferencing application accessible to those with limited movement. People with limited movement might struggle to maneouver the mouse cursor or perform the required keyboard presses to toggle mute, toggle video, raise/lower hand, and trigger emoticons. The VACO box (see Figure 13) consisted of a micro:bit, Jacdac adaptor, five buttons, two sliders, and six optional access switch input modules. Buttons across the middle can be used to signal emoticons, and a large central button supports quick mute toggling. The left-hand slider raises and lowers a virtual hand, and the right-hand slider switches video on and off. A cutout in the top case allows status display via the micro:bit. Six access switch input modules can be used to control all functions for users who require accessibility switches.

7.2 Button-to-speech Device

One of the hackathon participants has a daughter, Paloma, with cerebral palsy. The condition impacts her movement and her speech, meaning that automated voice assistants often cannot understand her. Using a micro:bit, Jacdac adaptor, a speech synthesis module, a speaker, a button, and a rotary control, this workstream built Paloma a portable text-to-speech device, shown in Figure 14. Three removable wooden legs mean that the device can be positioned stably over a Google Home Mini smart assistant. A simple website allows Paloma to load and save phrasebooks to the micro:bit. The rotary control integrated into the device allows Paloma to select the phrase she wants to use, and a simple button press utters that phrase aloud.

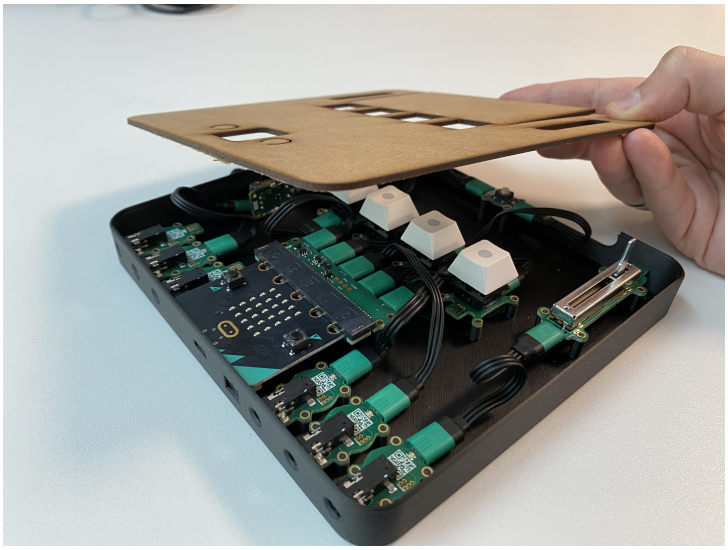


Fig. 13. A video conferencing accessibility controller. A 3D-printed enclosure contains mounting pillars that hold the Jacdac modules neatly in place and the laser-cut top cover includes a large 'living hinge' button (right hand-side).

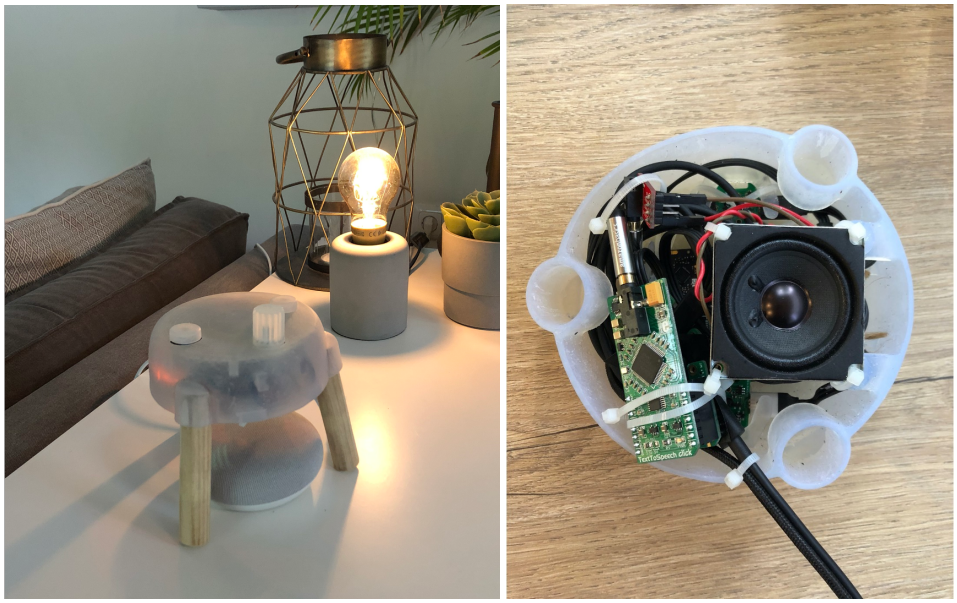


Fig. 14. A button-to-speech device. (a) The completed device positioned above a Google Home Mini smart assistant. (b) The internals of the device; the Jacdac speech synthesis module and speaker can been seen but the other Jacdac modules are mostly obscured.

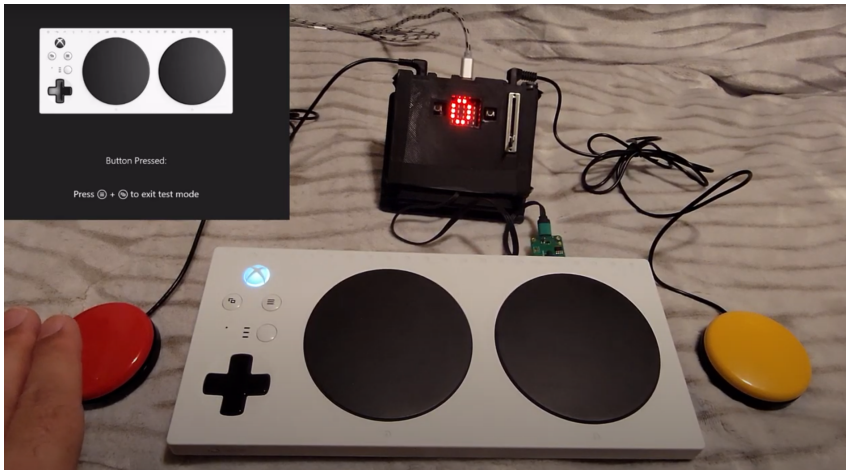


Fig. 15. The Jacdac-enhanced Xbox Adaptive Controller.

7.3 Enhancing the Xbox Adaptive Controller

The Xbox Adaptive Controller (XAC) was released in 2018 to allow more people with accessibility issues to participate in gaming [14]. Using established and readily available access switches (that have a user friendly 3.5mm audio jack connector), occupational therapists and allies of people with disabilities can easily create custom gaming setups for them.

One participant used the micro:bit's wireless capability to decouple access switches, which are often connected by a long trail of wires, from the XAC. Another hackathon participant made use of the Jacdac access switch output module to create a six-mode trigger for those who can't easily tap, double-tap and/or hold down a regular switch. They combined a micro:bit, an access switch output module connected to the XAC, and two access switch input modules connected to access switches. One access switch was used to toggle between modes, and the other was used to trigger the XAC input in different ways, for example simulating it being held down.

8 USER PERSPECTIVES

Due to COVID-19 restrictions, the highly distributed nature of a global hackathon and the large number of particpants, direct observation was not possible. However, we still wanted to deepen our understanding of the strengths and weaknesses of Jacdac. We therefore arranged follow-up interviews with five hackathon participants. Interviews were conducted via a recorded video call that lasted up to 30 minutes.

We also conducted preliminary interviews with four educators familiar with MakeCode to see how they perceived Jacdac. We recruited them through local networks known to one of the authors and sent them each a small Jacdac kit. Accompanying each kit was a set of instructions for adding Jacdac to MakeCode, and a guided "getting started" task (see Appendix). Again, direct observation was not possible due to restrictions imposed by COVID-19, but a video call interview of up to 45 minutes with each educator captured their feedback.

Both sets of interviews focused on how the participants reacted to and experienced Jacdac. Interviews were semi-structured, giving participants the opportunity to deviate from the pro-forma questions, helping to uncover additional insights. Interviews for educators and hackathon participants followed a similar outline (see Appendix for full list of interview questions) except when enquiring about a particular application domain (i.e. accessibility or education).

ID	Gender	Country	Occupation/background	Experience with physical computing
P1	M	Canada	Disability Organisation Lead	Some experience building prototypes with MakeCode, Arduino, Raspberry Pi.
P2	F	US	UX Designer	None.
P3	M	US	Accessibility Researcher	Experience designing 3D models, soldering, building systems with Arduino.
P4	F	US	Software Engineer	Used Arduino a few times in the past.
P5	M	US	Software Engineer	Electronic engineering degree, has done PCB design and used Raspberry Pi, micro:bit.
E1	M	UK	Computing & Physics Teacher (ages 11-14)	Uses micro:bit, MakeCode and Raspberry Pi to teach computing concepts.
E2	F	UK	Computing Teacher (ages 11-18)	Primarily uses Arduino to teach computing concepts, also uses micro:bit, MakeCode.
E3	F	UK	Computing and IT Teacher (ages 11-18)	Hosts physical computing club, uses micro:bit, MakeCode, Raspberry Pi.
E4	F	UK	Technology & Computing Teacher (ages 11-18)	Uses micro:bit, MakeCode, Picoboard and Raspberry Pi to teach computing concepts.

Table 1. Details of interviewees who shared their thoughts and experiences relating to Jacdac.

All interviews were transcribed for analysis, and themes were extracted using a hybrid-coding approach [13]. Deductive codes centered on: the MakeCode integration with Jacdac, experiences with other prototyping toolkits, how Jacdac compares to other prototyping toolkits, and Jacdac's suitability for particular application domains (e.g. education or accessibility). Further analysis of semi-structured interviews generated additional themes via inductive coding.

Details regarding the background, location, occupation and physical computing experience of all nine of our interviewees are summarized in Table 1.

8.1 Hackathon Participants' Perspectives

Initial reactions. All five hackathon participants were impressed by the ease of use offered by Jacdac and MakeCode. *"You just plug it in and it works"* (P1); *"I liked the fact that I was able to plug it in and MakeCode knew what I plugged in"* (P3); *"Anybody can pick it up. It doesn't matter if you are software engineering, hardware engineering, designer or project manager"* (P4); *"It was super easy...it was just plugging things together and seeing what happens"* (P5). After using Jacdac and MakeCode, P2 said that they were *"less intimidated by hardware"*. Interestingly, P5 volunteered that they could *"totally see this being introduced [at school] in the third grade and beyond"*.

Identifying modules. In order to help users identify Jacdac modules, each PCB is labelled with a brief description, e.g. 'button' or 'accelerometer'. Despite this, three participants (P2, P4, P5) reported that it was hard to discover the functionality of certain modules. For those less experienced with hardware (P2, P4), there was a clear domain-language gap. For example, the names 'potentiometer' (used for both the slider and flex sensor modules) and 'accelerometer' are technical terms for sensors which they did not recognize. However, although these terms presented a barrier to these users, the digital twins made the module functionality discoverable. P5 is an experienced electronics engineer, but they too found it difficult to identify the function of certain modules. For example, it wasn't until P5 connected the accelerometer to MakeCode and the digital twin appeared that they recognised the module's purpose.

P3 embraced a *"plug it in and see what happens"* approach to discovery. Reflecting on their previous experience with Arduino, they said *"Even after plugging in some wires ... you still have to go find a code sample before you can get that 'Hello world'."* P3 then went on to say that the *"plug it in and it does something by default"* experience of Jacdac is an entirely different proposition.

Using MakeCode. In general, participants all got on well with MakeCode and were successful creating programs using blocks. P1 reported that *"Sometimes you'd miss adding the extension and it wasn't clear whether Jacdac was on or not"*. Adding the Jacdac extension was required due to the extension framework offered by MakeCode. It is possible that in the future, MakeCode will support Jacdac natively, making a deeper integration possible.

P2 commented that it was confusing having to add blocks in MakeCode separately after connecting the modules *"After connecting it, it was confusing to realize you don't have the blocks"*. This design decision was driven by two factors: first, we did not want to bloat the toolbox with blocks when users are only experimenting; second, for security reasons MakeCode does not allow the extensions to be added without direct permission from the user.

Accelerating accessibility. The two hackathon participants we interviewed who are accessibility experts (P1 and P3) saw the potential benefits Jacdac could bring to the accessibility community. P1 compared Jacdac to their prior experience with Arduino *"With Jacdac, you're not running a marathon like you are when using Arduino ... Jacdac is modular and can really help the gap in accessibility technology through community innovation."* P3 believes *"Jacdac is more approachable than a typical Arduino for occupational therapists and allies"*. P3 also thinks that therapists could have *"a kit like this ... and use it to prototype accessibility devices with those with disabilities in real-time."* P1 and P3 both commented that when you eclipse a certain number of Jacdac modules, your prototype turns into *"cable soup"* reducing practicality for the end user. P2 also pointed out the limitations of exposed PCBs in the context of people with disabilities.

We have been thinking how to move "beyond the prototype" [21] with Jacdac. In addition to simplifying prototyping, Jacdac's three-wire bus can potentially also make it easier to manufacture devices comprising of Jacdac modules that are soldered down to a carrier board rather than being connected together with cables. We are working to enable an experience where, after a device has been successfully prototyped, an optimized and functionally equivalent PCB-based design can be generated automatically. P3 saw this potential for themselves and wished for *"a magic button"* that would allow *"ten magic PCBs [to be] delivered with an enclosure that fits."*

8.2 Educators' Perspectives

Initial reactions. All educators could see the benefits Jacdac brings to physical computing. E1 commented that they *"could see this straight away in my physical computing unit"* and E2 thought *"It would be good to add Jacdac"* to their micro:bit toolbox. E3 and E4 shared similar sentiments *"it definitely is something that I would use in the classroom"*. E4 went on to say, Jacdac *"is a lot easier, a heck of a lot easier. It's just plug in."*

Easy plug-and-play. We purposefully did not provide detail about how to use Jacdac cables and connectors, in order to gauge how intuitive they were to use. No-one experienced any issues. E4 found the cables *"so much easier"* and E3 especially enjoyed the fact that they did not *"have to worry about what pin that you're putting it on"*. E1 also liked the design of the cable and connector. E3 particularly valued the reversibility of the connector and thought it would avoid student questions like: *"Can you check this? Have I lined it up correctly?"*

Interestingly, E1—who has a lot of experience with electronics—revealed they didn't realise that the cable was reversible. Upon receiving the kit, they were not sure which way modules should be connected and inspected the internals of the cable connector. They incorrectly concluded that the cable could only be connected one way (i.e. to the top in Figure 5) because the internal pins of the cable are only on one side: *"I'm fairly sure the kids will put the cable in the wrong way and find the things don't work."* They were surprised and pleased to learn that the connector is, in fact, reversible and we conclude E1's prior experience with other physical computing experiences may have biased their assumptions about Jacdac cables and connectors.

Inspiring excitement and creativity. All educators saw the value of Jacdac in the context of the micro:bit ecosystem. Educators believed that not only would Jacdac provide more excitement for their students, but they themselves were inspired and excited.

E4 praised the potential to create more complicated projects: *"We could go deeper, where you couldn't go deep before. With more complicated sensors you can get more detail."* E4 explained that more advanced students want a path to more sophisticated applications and that Jacdac could provide this.

E2 was excited about the potential of combining the modules together to control real-world actuators like servos *"Could the slider move an object? Could it, could it? Could it be moving motors and things like that?"* E2 revelled in the potentials of leaving her students to explore and that Jacdac would potentially *"bring creativity into our classroom"*, increasing participation from *"students who choose to go and do art and photography"* rather than computing. E3 recognised the same effect *"I think you'll get a lot more girls interested because they're very creative and they want to design things"* and Jacdac *"is the way to go."*

Valuing extensibility. Two educators, E1 and E2, recognised how the extensibility of Jacdac could benefit the micro:bit ecosystem. Both educators gave extending the micro:bit with additional buttons as their first example. A popular activity with the micro:bit is a reaction timer. It is a collaborative yet competitive project that involves two users pressing button A or button B when a symbol is displayed on the micro:bit. The fastest to press their button wins the round. E1 saw the value in adding extra buttons via Jacdac to extend the activity to more children. E2 also mentioned being *"restricted"* when incorporating the micro:bit into design and technology (D&T) projects; often, students want to hide the micro:bit but that restricts access to the micro:bit's integrated buttons. E2 believes that not only would having additional buttons available via Jacdac improve D&T projects, but it would encourage students to think about design and user experience when programming.

Assimilating simulation. The educators all reacted positively to digital twins and our device simulators. E3 saw one of the benefits as being sure that hardware is connected together correctly. They mentioned the frustrations of being a computing educator is that *"You demonstrate, they go away, and it's like 'Oh Miss, it's not working!' OK, well have you checked this?..."* With Jacdac, E3 saw the simulators as being an immediate smoke test of whether everything is operational *"Once it was slotted in that's it."* E1 liked *"the idea of the having emulators on the screen and in real life in front"* of them commenting that the digital twins got their *"attention big time."* They especially enjoyed that digital twins update in real time along with changes in the hardware.

9 DISCUSSION

9.1 Limitations

We start the discussion by reflecting on some of the key limitations we see with Jacdac. From a technical point of view, we want to flag four:

- **Custom Jacdac connector.** Early prototypes of Jacdac used the ubiquitous 3.5mm stereo audio jack, which is readily available and remarkably cheap. But we found that many stereo audio cables had a relatively high resistance, and the constraints of detecting and accommodating inadvertent connection between a Jacdac device and a legacy device using the same connector were too constraining. We also experimented with 3-pin JST SH 1mm pitch connectors (similar to 4-pin Qwiic connectors), but users found them fiddly and sometimes fragile. We therefore decided to design a custom connector for Jacdac to more effectively meet our design goals. By making one mating half a PCB edge connector we removed component sourcing effort and cost, but it's still necessary for adopters to source custom Jacdac cables and unless/until these are manufactured in reasonable volume, they will be more expensive than comparable stereo audio cables.
- **Fixed data rate.** Jacdac communications occur at 1Mbaud, a good fit for sensor streaming applications and relatively simple and cheap to implement. However, it's not ideal for data-intensive devices like cameras; not only would camera frames transfer slowly, but other devices would struggle to take control of the bus to communicate their data.
- **No delivery guarantees.** Like other packet-based approaches such as UDP [30], the Jacdac physical layer provides no reliability guarantees—reliability must be layered on top. A reliable delivery layer is available in the Jacdac stack, however, it requires more memory and therefore cannot be supported by the very lowest cost devices.

- **No topology detection.** Jacdac's bus topology provides easy and flexible prototyping but doesn't natively support topology discovery. If a particular application requires device roles be assigned based on where those devices are connected, additional electronic components and associated firmware are required. Instead, a manual configuration step is required.

In addition to the technical limitations listed above, we also want to highlight two limitations of our research methodology:

- **Direct observation not possible.** We believe our hackathon-based evaluation provides many insights about Jacdac from users' hands-on experiences. However, it's likely that further insights were missed because COVID-19 concerns meant we couldn't observe our participants directly. On the flip-side, by supporting remote participants we were able to support 80 participants from around the globe.
- **Limited interview feedback.** Although all our interview participants clearly expressed the view that Jacdac lowered the barrier to entry to physical computing, with just 9 interviewees the generalizability of our findings could be called into question.

9.2 Current and Future Work

Some of the limitations listed above naturally point to future work. We are, for example, working on a simple topology detection scheme that could be added to certain modules. Similarly, we plan to observe users working with Jacdac to gain more detailed insights into what works particularly well and what is challenging for them.

Over the last year, we worked with manufacturing partners to make Jacdac available to a wider audience, with commercial availability of the first kit announced in June 2022. We are also strengthening Jacdac's interoperability with platforms like the Raspberry Pi and extending our software integration to other tools and languages. As mentioned in Section 8.1 we are exploring ways in which Jacdac might support a transition from prototypes such as our CO_2 alarm to reliable low-volume production. Finally, we are developing ways to embed Jacdac modules and devices in enclosures, based on requests from some of our hackathon participants.

In addition to our own plans to further develop Jacdac, we hope that the open-source nature of Jacdac will encourage other researchers and practitioners to adapt and extend what we have built so far and presented here.

9.3 Conclusion

Over the past two decades, we have seen physical computing move from a niche activity associated with a handful of researchers, artists and makers, to a mainstream pursuit that now extends to the classroom and beyond. As the field continues to develop, the technologies and platforms that underpin it will inevitably evolve too. In this paper, we presented some of the existing protocols and products that are well-established in the field of physical computing, reviewed research relating to the design and evaluation of physical computing platforms, and identified three design goals (DG1-DG3) for future physical computing solutions. We then introduced Jacdac, an open-source platform for low-cost, plug-and-play physical computing that we believe epitomizes a natural evolution.

Jacdac is designed for **easy hardware composition** (DG1). A purpose-built and easy-to-use connector supports the dynamic addition of modules whose capabilities are automatically detected, leveraging a **truly plug-and-play software abstraction** with standardized services (DG2), allowing devices of common functionality to replace one another at run-time. Finally, with each design decision, cost has been an important consideration, resulting in a **low cost** (DG3) plug-and-play platform for physical computing. The core contribution of our work is to demonstrate how a service-based architecture, implemented on very low cost MCUs, can simplfy the construction of physical computing solutions. A second major contribution is illustrating the power of web technologies and platforms for programming and debugging the resulting systems.

To prove the viability of Jacdac, we manufactured over two thousand Jacdac modules, representing around forty different designs, along with one thousand cables. These were used to evaluate Jacdac via an 80-participant hackathon and we subsequently interviewed five of our hackathon participants along with four educators. The artifacts built by hackathon participants and the follow-up interviews demonstrated that the plug-and-play experience offered by Jacdac lowers the barrier to entry to physical computing. The preliminary interviews we conducted with educators also demonstrate the potential of Jacdac in the classroom.

We hope that technologies like Jacdac, which make physical computing easier and more powerful without compromising on cost, will further extend the reach of physical computing to new and more diverse communities, driving inclusion and adoption. In the future, perhaps people with disabilities and their allies will use physical computing as a matter of course to build customized devices that suit individual needs, as the hackathon participants did. Ultimately, we believe that Jacdac can help a broad range of users, beginners and more experienced creators from all backgrounds, build interactive devices for a range of applications from art pieces to IoT-like home automation solutions.

The true test of whether Jacdac meets our design goals—and indeed if our design goals address the key limitations of today's physical computing platforms—will be whether or not it is adopted by the community over the coming years.

ACKNOWLEDGEMENTS

We would like to thank Mike Hall, Brannon Zahand, John Helmes, Masaaki Fukumoto, Weishung Liu, Lason Jiang, Shuling Fang, Jorge Garza, Richard Lin, as well as the organizations Mustard Tek, Makers Making Change, and Microsoft Garage for their contributions to the work presented in this paper.

REFERENCES

[1] James Adams. 2022. Introducing Raspberry Pi HATs - Raspberry Pi. https://www.raspberrypi.org/blog/introducing-raspberry-pi-hats/

[2] 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.

[3] Dan Awtrey and Dallas Semiconductor. 1997. Transmitting data and power over a one-wire bus. *Sensors-The Journal of Applied Sensing Technology* 14, 2 (1997), 48–51.

[4] Thomas Ball, Peli de Halleux, and Michal Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*, Antony L. Hosking and Irene Finocchi (Eds.). 105–116.

[5] Massimo Banzi and Michael Shiloh. 2014. *Getting started with Arduino: the open source electronics prototyping platform.* Maker Media, Inc.

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

[7] Rebecca F. Bruce, J. Dean Brock, and Susan L. Reiser. 2015. Make space for the Pi. *Conference Proceedings - IEEE SOUTHEASTCON* 2015-June, June (2015). https://doi.org/10.1109/SECON.2015.7132994

[8] Leah Buechley, Mike Eisenberg, Jaime Catchen, and Ali Crockett. 2008. The LilyPad Arduino: using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 423–432.

[9] Zach Shelby Cullen Jennings, Jari Arkko, Ari Keränen, and Carsten Bormann. 2018. *Media Types for Sensor Measurement Lists (SenML).* Technical Report. Internet-Draft draft-ietf-core-senml-11, Internet Engineering Task Force.

[10] James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: Intuitive and Efficient Embedded Systems Programming for Education. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Philadelphia, PA, USA) *(LCTES 2018)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3211332.3211335

[11] Dale Dougherty. 2012. The maker movement. *Display & Design Ideas : DDI* 27, 4 (2012), 80–85. https://doi.org/10.1162/INOV_a_00135

[12] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.

[13] Jennifer Fereday and Eimear Muir-Cochrane. 2006. Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development. *International Journal of Qualitative Methods* 5, 1 (2006), 80–92. https://doi.org/10.1177/

160940690600500107 arXiv:https://doi.org/10.1177/160940690600500107
[14] Camille Godineau. 2018. The new Xbox adaptive controller, another step towards digital inclusion? https://mastersofmedia.hum.uva.nl/blog/2018/09/23/the-new-xbox-adaptive-controller-another-step-towards-digital-inclusion/
[15] Saul Greenberg and Chester Fitchett. 2001. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*. ACM, 209–218.
[16] Michael Grieves and John Vickers. 2017. Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. In *Transdisciplinary perspectives on complex systems*. Springer, 85–113.
[17] Björn Hartmann, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. 2006. Reflective Physical Prototyping through Integrated Design, Test, and Analysis. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (Montreux, Switzerland) *(UIST '06)*. Association for Computing Machinery, New York, NY, USA, 299–308. https://doi.org/10.1145/1166253.1166300
[18] Steve Hodges, Sue Sentance, Joe Finney, and Thomas Ball. 2020. Physical Computing: A Key Element of Modern Computer Science Education. *IEEE Computer* 53, 4 (2020), 20–30. https://doi.org/10.1109/MC.2019.2935058
[19] Steve Hodges, Stuart Taylor, Nicolas Villar, James Scott, Dominik Bial, and Patrick Tobias Fischer. 2013. Prototyping connected devices for the Internet of Things. *Computer* 46, 2 (2013), 26–34. https://doi.org/10.1109/MC.2012.394
[20] JIMBLOM. 2022. Arduino Shields - learn.sparkfun.com. https://learn.sparkfun.com/tutorials/arduino-shields/all
[21] Rushil Khurana and Steve Hodges. 2020. Beyond the Prototype: Understanding the Challenge of Scaling Hardware Device Production. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–11.
[22] 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. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 2, Article 70 (Jun 2021), 24 pages. https://doi.org/10.1145/3463523
[23] J.C. Lee, D. Avrahami, S.E. Hudson, J. Forlizzi, P. Dietz, and D. Leigh. 2004. The calder toolkit: wired and wireless components for rapidly prototyping physical computing devices. *Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques* 04 (2004), 167–175. https://doi.org/10.1145/1013115.1013139
[24] Frédéric Leens. 2009. An introduction to I2C and SPI protocols. *IEEE Instrumentation & Measurement Magazine* 12, 1 (2009), 8–13.
[25] Gareth Loy. 1985. Musicians make a standard: the MIDI phenomenon. *Computer Music Journal* 9, 4 (1985), 8–26.
[26] John Maloney, Kylie Peppler, Yasmin B. Kafai, Mitchel Resnick, and Natalie Rusk. 2008. Programming by choice: urban youth learning programming with scratch. *SIGCSE '08 Proceedings of the 39th SIGCSE technical symposium on Computer science education* (2008), 367–371. https://doi.org/10.1145/1352135.1352260
[27] MikroElektronika. 2022. Click Boards - MikroElektronika. https://www.mikroe.com/click-boards
[28] Bruce Jay Nelson. 1981. *Remote procedure call*. Carnegie Mellon University.
[29] Grace Ngai, Stephen CF Chan, Vincent TY Ng, Joey CY Cheung, Sam SS Choy, Winnie WY Lau, and Jason TP Tse. 2010. i* CATch: a scalable plug-n-play wearable computing framework for novices and children. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 443–452.
[30] Jon Postel et al. 1980. User datagram protocol. STD 6, RFC 768, August.
[31] Jon Postel et al. 1981. Transmission control protocol. STD 7, RFC 793, September.
[32] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52 (2009), 60–67. https://doi.org/10.1145/1592761.1592779 arXiv:-
[33] José Rufino. 1997. An overview of the controller area network. In *Proceedings of the CiA Forum CAN for Newcomers*.
[34] 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* (Stanford, California, USA, 2015-01-15) *(TEI '15)*. Association for Computing Machinery, New York, NY, USA, 347–354. https://doi.org/10.1145/2677199.2680581
[35] Dallas Semiconductor. 1998. Fundamentals of RS-232 serial communications. (1998).
[36] Charles R. Severance. 2014. Massimo Banzi: Building Arduino. *IEEE Computer* 47, 1 (2014), 11–12. https://doi.org/10.1109/MC.2014.19
[37] ISO Standard. 1993. 11898: Road vehicles—interchange of digital information—controller area network (can) for high-speed communication. *International Standards Organization, Switzerland* (1993).
[38] Arduino Team. 2021. Introducing the Arduino UNO Mini Limited Edition: Pre-orders now open. https://blog.arduino.cc/2021/11/24/introducing-the-arduino-uno-mini-limited-edition-pre-orders-now-open/
[39] Nicolas Villar, Kiel Gilleade, Devina Ramdunyellis, and Hans Gellersen. 2007. The VoodooIO gaming kit: a real-time adaptable gaming controller. *Computers in Entertainment (CIE)* 5, 3 (2007), 7.
[40] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. 2012. .NET Gadgeteer: A platform for custom devices. In *International Conference on Pervasive Computing*. Springer, 216–233.
[41] Wikipedia. 2022. Raspberry Pi - Wikipedia. https://en.wikipedia.org/wiki/Raspberry_Pi

10 APPENDIX

10.1 Overview of Hackathon On-boarding Process

On the first day of our hackathon participants were welcomed, given an overview of the coming three days and assigned to workstreams. They were also invited to a MakeCode and Jacdac training session following the welcome presentation. After the training session, participants were invited to attend their assigned workstream meetings; participants were also able to work independently if desired. The following two days largely followed the same structure. Chat channels and ad-hoc video calls were used by participants to collaborate within their workstreams, seek technical help, and to provide feedback.

10.2 Hackathon Participant Interview Questions

Overall experience

(1) What is your background?
(2) Which hackathon workstream did you select?
(3) What was your contribution to the selected workstream?

Prototyping process

(1) Can you briefly describe how you approached this hackathon? Did you plan things out before hand? Did you just hack things together as you went along?
(2) Were there any points where you had to alter or change your approach, if so why?

Jacdac and MakeCode

(1) Were there any particular qualities of Jacdac that made this prototyping experience easier or harder?
(2) What are your thoughts on the Jacdac and MakeCode workflow?
(3) Did you encounter any problems? If so, what?

Comparison to other prototyping tools

(1) Do you have experience with other prototyping tools?
(2) If so – can you please contrast your experience with Jacdac and those tools?

The hackathon

(1) What are your key takeaways from the hackathon?
(2) How did you find the hackathon overall?
(3) What were your personal highlights?
(4) Was there anything that could have made the hackathon better?

Specific challenges

(1) Did you at any point think about enclosures for the artifact your built?
(2) Do you have any thoughts on Jacdac as a prototyping tool?
(3) Do you have any thoughts on Jacdac as a tool for designing accessibility devices?

10.3 Overview of Educator On-boarding and Tasks

Tutorial

1. *Open MakeCode for micro:bit.*
2. *Create a new project.*
3. *Connect MakeCode to your micro:bit.*

4. Add the Jacdac extension to MakeCode. After completing this step, you should see digital twins of any modules you have connected to your micro:bit in MakeCode.

5. Programming Jacdac modules. After following the steps above, you should now have a green modules category in the MakeCode Blocks toolbox. However, if you expand the "Modules" blocks category, you will not find any blocks for the modules you have connected to your micro:bit. This is because one extra step is required. To add the blocks for the modules you have connected click the "add blocks" button. Now you know the steps to build a program with Jacdac modules! Hooray!

N.B. A GIF showing the step to be performed was displayed alongside each of the above steps.

Task one

This task is a step by step guide to plot the value of the slider module using the micro:bit's display. The hardware required for this task is:

- a micro:bit V2,
- a USB cable connecting the micro:bit V2 to your computer,
- a micro:bit V2 Jacdac adapter,
- a Jacdac slider module, and
- a Jacdac cable to connect the slider module to the micro:bit Jacdac adapter.

Step 1. Please create a new MakeCode project, and follow the steps above to add Jacdac to your project. (If you've just created a new project using the steps above, there is no need to repeat the steps.)

Step 2. If you've not already connected your Jacdac slider, connect it to your micro:bit. With the slider connected, check it has appeared in the MakeCode editor and click "Add blocks".

Step 3. Grab the 'on potentiometer1 position changed by 5' from the modules block category. Then, take a 'plot bar graph' block from the "LED" category and place it inside the 'on potentiometer1 position changed by 5' block.

Step 4. From the modules category, grab the "potentiometer1 position" and place it on the first "0" of the "plot bar graph" block.

Step 5. Slider values range from 0-100. So, let's change the "up to" portion of the plot bar graph block to 100.

Step 6. That's it! Now, when you move the position of your physical slider, you will see a proportional change to the number of LEDs displayed on the MakeCode simulator display.

(optional) Step 7. Just like any MakeCode program, you can click the download button to see the same program running on your physical micro:bit. When the program is downloaded, the physical slider will change the number of LEDs displayed on the physical micro:bit's display.

Task 2

This next task will be less structured and will require the slider module and the 10-RGB LED bar module. Please connect these two modules to the micro:bit Jacdac adapter.

Now, with those modules connected, create a program that: when the slider value changes, the colour of the RGB LED bar module changes.

10.4 Educator Interview Questions

Background

(1) What subjects do you teach?
(2) What do you identify as?

(3) How old are you?

Micro:bit/MakeCode experience

(1) How many years have you been teaching?
(2) How long have you been using micro:bit and MakeCode?
(3) Have you used any hardware accessories with micro:bit and MakeCode? Which ones?
(4) What do you see as the pros of micro:bit/MakeCode?
(5) What do you see as the cons of micro:bit/MakeCode?
(6) Have you used any other technology in the classroom?
(7) What other physical computing ecosystems have you used (e.g. Arduino, Raspberry Pi)?
(8) How does micro:bit/MakeCode compare to these other ecosystems?
(9) Have you used an accessories with these other ecosystems? How did you find that?

Jacdac experience

(1) How did you get on with each task?
(2) How did you find connecting Jacdac devices together?
(3) How useful were the Jacdac device twins ("simulators")?
(4) Was there anything you found challenging when using Jacdac?
(5) What was your highlight of the Jacdac experience (if any)?
(6) How does Jacdac compare to other solutions you've used in the classroom?
(7) Would you feel comfortable and confident using Jacdac in the classroom?
(8) What year group would you use Jacdac with?
(9) What Jacdac modules would you like to see?