Jacdac: Service-Based Prototyping of Embedded Systems

THOMAS BALL, Microsoft, USA
PELI DE HALLEUX, Microsoft, USA
JAMES DEVINE, Microsoft, UK
STEVE HODGES, Lancaster University, UK
MICHAŁ MOSKAL, Microsoft, USA

The traditional approach to programming embedded systems is monolithic: firmware on a microcontroller contains both application code and the drivers needed to communicate with sensors and actuators, using low-level protocols such as I2C, SPI, and RS232. In comparison, software development for the cloud has moved to a service-based development and operation paradigm: a service provides a discrete unit of functionality that can be accessed remotely by an application, or other service, but is independently managed and updated.

We propose, design, implement, and evaluate a service-based approach to prototyping embedded systems called Jacdac. Jacdac defines a service specification language, designed especially for embedded systems, along with a host of specifications for a variety of sensors and actuators. With Jacdac, each sensor/actuator in a system is paired with a low-cost microcontroller that advertises the services that represent the functionality of the underlying hardware over an efficient and low-cost single-wire bus protocol. A separate microcontroller executes the user's application program, which is a client of the Jacdac services on the bus.

Our evaluation shows that Jacdac supports a service-based abstraction for sensors/actuators at low cost and reasonable performance, with many benefits for prototyping: ease of use via the automated discovery of devices and their capabilities, substitution of same-service devices for each other, as well as high-level programming, monitoring, and debugging. We also report on the experience of bringing Jacdac to commercial availability via third-party manufacturers.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; **Embedded software**; **Firmware**; • **Software and its engineering → Embedded software**; **Abstraction, modeling and modularity**.

Additional Key Words and Phrases: embedded systems, services, plug-and-play, microcontrollers

ACM Reference Format:
Thomas Ball, Peli de Halleux, James Devine, Steve Hodges, and Michał Moskal. 2024. Jacdac: Service-Based Prototyping of Embedded Systems. *Proc. ACM Program. Lang.* 8, PLDI, Article 175 (June 2024), 24 pages. https://doi.org/10.1145/3656405

1 INTRODUCTION

The traditional approach to programming embedded systems is monolithic: firmware on a microcontroller unit (MCU) contains both application code and the drivers needed to communicate with sensors, actuators, and other peripherals using low-level protocols such as I2C, SPI, and RS232 [9, 18, 30]. Such protocols were designed to provide a universal interconnect between microcontrollers and their peripherals; they are efficient but are low-level and use static addressing. While

Authors' addresses: Thomas Ball, tball@microsoft.com, Microsoft, USA; Peli de Halleux, jhalleux@microsoft.com, Microsoft, USA; James Devine, Microsoft, UK; Steve Hodges, steve.hodges@lancaster.ac.uk, Lancaster University, UK; Michał Moskal, mimoskal@microsoft.com, Microsoft, USA.

ACM 2475-1421/2024/6-ART175
https://doi.org/10.1145/3656405

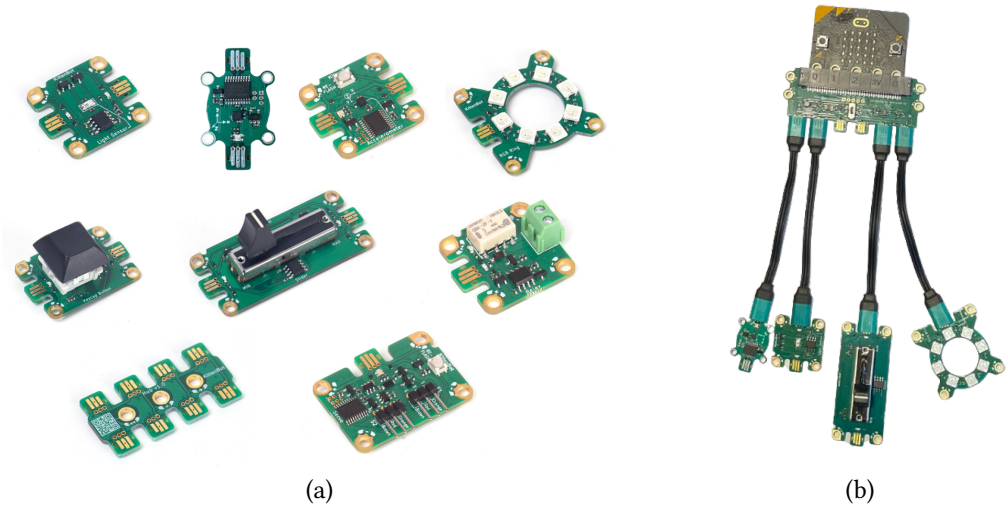


Fig. 1. (a) a variety of Jacdac modules: (first row) light sensor, two different accelerometers, LED ring module; (second row) button, slider, and relay modules ; (third row) passive hub and servo modules; (b) "night light" system—a BBC micro:bit slotted into a Jacdac adaptor, connected to four modules: accelerometer, light sensor, slider, and LED ring.

software abstractions such as those provided by Arduino [31], ARM's Mbed [2], and TinyOS [21] provide higher-level APIs for programmers, the result is still a monolithic system with a tight coupling between software and a static configuration of specific hardware components, as well as the underlying protocols they depend upon.

In comparison, in the world of the web and cloud, software development has largely transitioned from the delivery of monolithic layered systems to a service-based development and operation paradigm. A service provides a discrete unit of functionality that can be accessed remotely by an application program, or other service, but is independently managed and updated. Services have radically changed how software is produced, delivered, and operated. Microservices can be used to further decompose applications and services into smaller units of functionality [13].

We propose, design, implement, and evaluate a service-based approach to *prototyping* embedded systems called Jacdac. Our main contribution is to show that it is possible to efficiently map service-based abstractions to embedded systems, at low cost and reasonable performance. As a result, many benefits can be realized that support prototyping, including: ease of use via the automated discovery of devices and their capabilities, substitution of same-service devices for each other, and high-level programming. Our target audience ranges from complete beginners including schoolchildren exploring physical computing [17] to proficient programmers who are unfamiliar with embedded systems such as web developers.

Jacdac defines a service specification language, designed especially for embedded systems, along with a host of specifications for a variety of sensors and actuators. Service specifications provide a separation of concerns between application code (client) and driver code (server) that interfaces with hardware sensors and actuators. A Jacdac module (server) is a device that has one or more sensors/actuators and advertises the services that it supports over the Jacdac bus. A Jacdac brain (client) is a device that runs an application program, which consumes the services available on the bus.

Figure 1(a) shows a variety of Jacdac modules. The printed circuit boards (PCBs) of all modules have one or more Jacdac 3-pin edge connectors; these are double-sided and wired so that the Jacdac cable makes a stable connection, no matter which way it is plugged in. The PCB-based edge connector is low-cost and provides a consistent and reliable experience for at least 1500 plug/unplug cycles.

Figure 1(b) shows a small embeddable system built from a Jacdac brain (a BBC micro:bit [3], shown at the top), a Jacdac adaptor (in the middle), and four Jacdac modules (from left to right, accelerometer, light sensor, slider, and LED ring), which are connected to the adaptor via Jacdac cables to form a single 3-wire bus. We will use this small Jacdac system to build a "night light" that is only activated when the accelerometer is facing a given direction; the application logic then turns on the LED ring when the light level falls below a certain threshold; the slider is used to control the brightness of the LED ring (we do not consider here how to package the components into a more suitable form factor).

The Jacdac service specification language is supported by a three-layer protocol: the *service* layer represents all Jacdac services, including a set of common services for device discovery, advertisement of a device's services, power management, and firmware updating; the *transport* layer is responsible for forwarding packets to the appropriate service or application; the *network interface* layer deals with the transmission of Jacdac packets over the wire. Jacdac implements a "single wire serial" protocol, a UART-based data transmission protocol that uses one shared wire for half-duplex data, plus one for ground and one to supply power.

In our current implementation, each Jacdac module has a dedicated MCU with firmware that implements the three layers of the Jacdac protocol and exposes a module's on-board components via services on the bus. A Jacdac module's MCU abstracts over the specific hardware, adapting it to the appropriate Jacdac service; this is analogous to how web services were originally used to wrap legacy enterprise applications and make them available on the web. For many sensors and actuators we support the Jacdac protocol using 8-bit MCUs with 64 bytes of RAM that cost as little as US $0.03.

The Jacdac platform, both hardware designs and software, is open source[1] and includes:

- a large *library of service specifications*, with supporting server firmware, web-based simulators, and client bindings in a variety of languages;
- a growing *device catalog* of 80 Jacdac modules and several brains, produced by us and others;
- a platform-agnostic C99 implementation of the Jacdac protocol and a number of servers and drivers for I2C/SPI components;
- a *device development kit* (DDK) with hardware designs, and firmware source code for a range of MCUs, including the 8-bit PADAUK MCU, the 32-bit ARM-based family of STM32x0 MCUs, as well as the ESP32 (which supports WiFi, TCP/IP and TLS) and the RP2040;
- additional implementations of the Jacdac protocol/runtime in Python, C#, TypeScript, and Static TypeScript [6] (the subset of TypeScript supported by the MakeCode system [5, 11]);
- a (mostly code-generated) client library in each of the above languages for each service;
- a Jacdac website with monitoring/tracing, simulation, and testing tools.[2]

Altogether, the Jacdac stack effectively separates clients (brains) and server (modules) via services and the supporting protocol, enabling the dynamic creation and modification of the system. Programmers can choose from a variety of different programming languages to develop client/application code. A previous paper evaluated the user experience with Jacdac using modules designed and manufactured by us [12]. This paper focuses on the design and technical implementation of

[1]The main repo is https://github.com/microsoft/jacdac, which links to the other repos as well as the Jacdac website.
[2]The Jacdac website is at https://aka.ms/jacdac.

```
1  # Accelerometer
2
3      identifier: 0x1f140409
4      extends: _sensor
5
6  ## Registers
7
8      ro forces @ reading {
9          x: i12.20 g
10         y: i12.20 g
11         z: i12.20 g
12     }
13
14 Indicates the current forces
15 acting on accelerometer.
16     ro forces_error?: i12.20 g
17        @ reading_error
18
19 ## Events
20
21     event face_up @ 0x85
22     event face_down @ 0x86
23
24 Emitted when accelerometer is
25 laying flat in the given direction.
26
27     event freefall @ 0x87
28     event shake @ 0x8b
29     ...
```

Fig. 2. Jacdac accelerometer service (partial).

the Jacdac platform and evaluates it with respect to the cost of the solution, its generality, and the overhead that the separation of client and server incurs. Three Jacdac kits, comprising over twenty modules, have been produced by two third-party manufacturers[3] and are now commercially available, further validating our approach.

2 OVERVIEW

This section presents how a three-axis accelerometer is represented and programmed using Jacdac. This example illustrates how Jacdac supports prototyping through *standardized service specifications* — communication between devices is mediated via service specifications so that similar devices can act as drop-in replacements for one another. Using the *Jacdac protocol*, devices can be the providers and/or clients of services, allowing greater flexibility in application/system design than present in monolithic systems; devices and their services are discovered dynamically as they join the Jacdac bus. Finally, Jacdac supports *application/client programming* in a variety of high-level languages, with monitoring and debugging support provided by a web dashboard that joins the Jacdac bus using WebUSB or WebSerial.

2.1 Accelerometer Service

Figure 2 presents the (partial) source text of a Jacdac service specification for a three-axis accelerometer, which is specified using a simple markdown language where indented text represents the formal specification and non-indented text is descriptive.[4] Every Jacdac service is uniquely identified by a 32-bit identifier (line 3), also referred to as the *service class*, which is assigned randomly by the initial author of the service (we maintain a central repository of service specifications;[5] tools check for service class collisions in newly submitted service specifications). We don't expect there to be more than a few thousand service classes overall, compared to billions of device instances (which use 64-bit identifiers). Line 4 of the specification states that the accelerometer service extends the abstract sensor service, which defines a set of common registers for working with sensors.

[3]KittenBot (https://www.kittenbot.cc/) and Forward Education (https://forwardedu.com). The modules in Figure 1 (a) are a sample drawn from KittenBot's Kit A and Kit B, as well as one created by the authors (the "turtle" shaped accelerometer).
[4]Full source at https://microsoft.github.io/jacdac-docs/services/accelerometer/.
[5]The Jacdac service library and device catalog reside at https://github.com/microsoft/jacdac.

Line 8 defines a read-only register named `forces`, a record with three fields (`x`, `y`, and `z`). Every Jacdac register is assigned a unique numeric code: the "`@ reading`" annotation on line 8 states that the `forces` register is using the common code `0x101` as defined in the base service specification (that all services inherit from). Sharing numeric codes allows for common sensor-handling code, regardless of the specific service class. Lines 9, 10 and 11 define the type and units for the three fields (`x`, `y`, and `z`) of the `forces` register. The `i12.20` type is a signed 32-bit fixed point value, with 12 bits for the integer portion and 20 bits for the fractional part. Any data field, such as the three above, should be annotated with its unit. Jacdac supports a large set of units ("`g`" is earth gravity).

The datasheet for a sensor specifies its sensitivity (which may depend on environmental factors such as temperature), output resolution, and noise, among other characteristics. Lines 16-17 of Figure 2 specify a register named `forces_error` which exposes the expected error when reading the `forces` register.

The stream of values of any given sensor may give rise to a sequence of discrete events that capture various patterns in the stream. Lines 21-28 declare a handful of events that the accelerometer service raises: `freefall` is emitted when the total force acting on the accelerometer is much less than 1g, while `shake` is emitted when the forces change violently a few times in a short period. The `face_up` and `face_down` events are used in our running example.

2.2 Accelerometer Module: Hardware

A "turtle" shaped accelerometer module we designed and produced appears in the first row of Figure 1(a); to the immediate right of this is a square shaped accelerometer module from KittenBot. Both modules support the accelerometer service and are interchangeable. The large integrated circuit on the PCB of both modules is a STM32F030x4 MCU (16kB flash, 4kB RAM, running at 8MHz), connected via I2C to Kionix's KXTJ3 3-axis digital accelerometer, the square IC centered under the MCU on the PCB. Both the front and back of the PCB depict the direction of the three axes printed on the silkscreen. Both modules have two Jacdac edge connectors, which share the same three PCB traces (PWR, GND, DATA), to allow the module to be connected to the Jacdac bus and daisy-chained. Jacdac uses the UART capability on the MCU to allow the module to send and receive Jacdac packets over the bus.

2.3 Accelerometer Module: Firmware and Jacdac Protocol

The responsibility of a Jacdac module's firmware is to make its underlying hardware available as one or more Jacdac services. In our example, the firmware abstracts over the KXTJ3 accelerometer, exposing it via the Jacdac accelerometer service. The firmware includes the Jacdac runtime, enabling the accelerometer module to join the Jacdac bus, advertise itself and its support for the accelerometer service, respond to requests a client may send it, as well as generate events of interest. The firmware communicates with other Jacdac-aware devices using the Jacdac protocol, and communicates with the on-board KXTJ3 accelerometer over I2C. As the KXTJ3 uses a different representation of forces from the service of Figure 2, the firmware also converts to the specified representation before the register value is communicated via Jacdac.

The firmware for the accelerometer module is built using three layers:

(a) an *MCU-specific* C99 implementation of a hardware abstraction layer (HAL) that provides the necessary primitives needed to interface with common hardware interfaces (I2C, SPI) as well as those needed by the Jacdac protocol (UART);[6]

[6]See https://github.com/microsoft/jacdac-stm32x0.

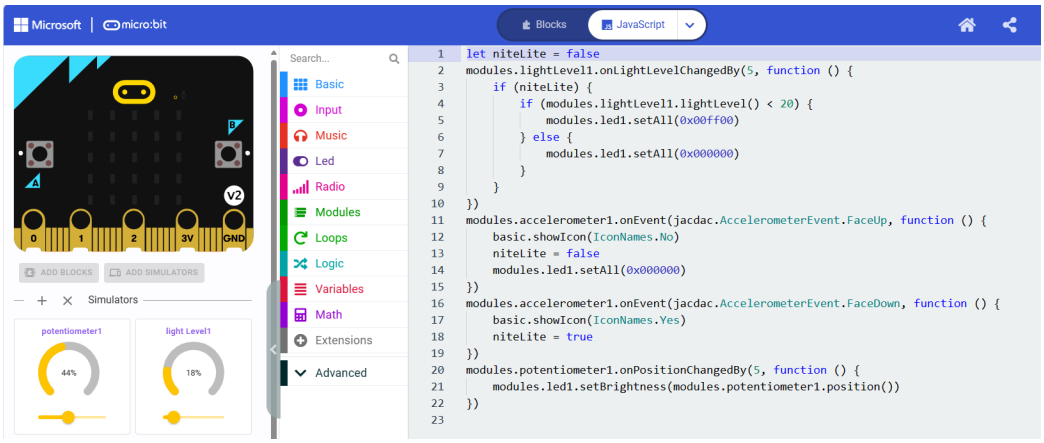


Fig. 3. JavaScript program for the "night light" running example. The coding environment is MakeCode for the micro:bit[5], extended with support for displaying the state of connected and/or simulated Jacdac modules/services.

(b) an *MCU-independent* C99 implementation of the Jacdac protocol (that relies on the HAL), including the Jacdac control service, which advertises the supported services, and implementations of many services and drivers for various I2C/SPI hardware sensors, including the KXTJ3 accelerometer;[7]

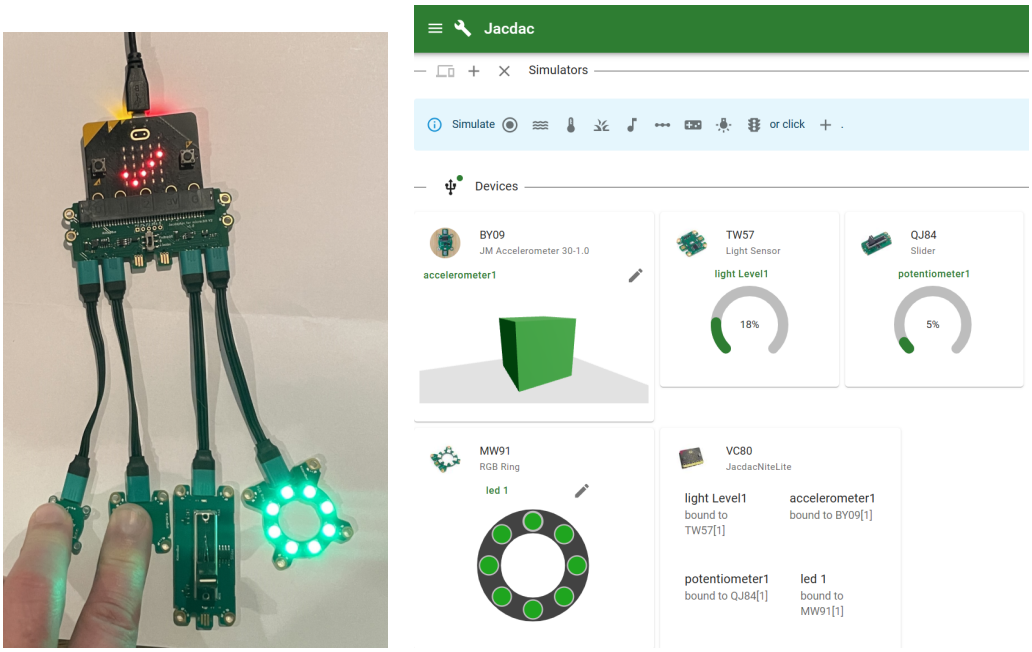
(c) finally, a small C driver that brings the above code together for the specific accelerometer module, specifying the hardware (KXTJ3) used, the orientation of the chip with respect to silk markings on the PCB, and the manufacturer and device name for easy user identification.[8]

2.4 Brain/Client Programming

We have ported the Jacdac protocol implementation to a variety of higher-level languages, including Python, C#, TypeScript, and Static TypeScript (the programming language of Microsoft's MakeCode editors for physical computing [5]). These ports primarily support the programming of brains/clients, with abstractions to hide the underlying asynchrony of the Jacdac protocol from the application programmer, although they also can be used to implement servers. Code generation tools compile each specification into high-level client APIs for each supported language, that call into the underlying runtime. In the running example, we make use of the MakeCode Jacdac extension/library[9], most of which was generated automatically including the code for working with the accelerometer service.[10]

Figure 3 shows a Static TypeScript program in the MakeCode editor for the micro:bit[11] that implements the logic of the "night light" application. The program is a client of accelerometer, light level sensor, LED ring, and slider modules/services. The MakeCode Jacdac runtime provides singletons (`accelerometer1`, `lightLevel1`, `potentiometer1`, `ledStrip1`) for working directly with modules exposing a single service (the mapping from names to device identifiers is discussed

[7] See https://github.com/microsoft/jacdac-c.
[8] See https://github.com/microsoft/jacdac-msr-modules/tree/main/targets/jm-accelerometer-30-1.0.
[9] See https://github.com/microsoft/pxt-jacdac.
[10] The following web page shows the status of each service including the status of automatically generated code for MakeCode: https://microsoft.github.io/jacdac-docs/tools/service-status/.
[11] See https://makecode.microbit.org.



(a) (b)

Fig. 4. (a) "Night light" system from Figure 1, programmed per Figure 3 - the accelerometer and light level module are being held face down, resulting in the LED ring turning on; (b) digital twins of the four modules and the BBC micro:bit brain. Note that the light level is the amount of light being read by the light level sensor while the potentiometer reflects the position of the slider, which controls the brightness of the LED ring.

later). The program has event handlers for responding to the `face_down` and `face_up` events from the accelerometer module: the handler for a face down event displays a check mark (`IconNames.Yes`) on the micro:bit screen and enables the night light logic; the handler for a face up event displays an X (`IconNames.No`), disables the night light logic, and turns off all LEDs on the LED ring. The program has two other event handlers for monitoring the value of the light level and slider/potentiometer and firing when the value has changed by the specified amount. When the light level is low, the LED ring is turned on (green). When the slider changes, the brightness of the LED ring is adjusted based on the new value.

2.5 Dashboard and Digital Twins

Figure 4(a) shows the system from Figure 1 with the MakeCode program deployed to the micro:bit brain and the accelerometer and light level modules being held face down. This turns on the night light logic (see check mark on micro:bit screen) and results in the LED ring turning on. The micro:bit is connected over USB to a host computer, extending the Jacdac bus over USB so it can be accessed from a web browser. Figure 4(b) shows the web-based Jacdac dashboard, which displays the digital twins of the micro:bit and the four modules connected to it. The current state of the modules is shown (light level is 18%) and the LED ring is displaying all green. Also, note that the actual devices have been identified (by lookup in the device catalog) and small images of them appear in the dashboard too.

2.6 Summary

This section presented the essential facets of Jacdac. A Jacdac brain executes client code that can discover Jacdac services on the bus, as advertised by Jacdac modules (servers). Communication among Jacdac devices takes place via a packet-based protocol that leverages the low-cost UART hardware available on MCUs. Each service represents a discrete unit of functionality, such as an accelerometer, that can be accessed remotely over the Jacdac bus. Standardized Jacdac service definitions abstract away the specific hardware that a module uses. As shown in the accelerometer example, the Jacdac bus can be extended over WebUSB to allow the web browser to join the conversation. This enables rapid prototyping of client programs in the web browser that work against physical Jacdac modules as well as virtual ones.

3 SERVICE SPECIFICATION LANGUAGE

As shown in the previous section, services provide abstract, standardized interfaces that can be used to work with physical hardware resources and permit devices with the same functionality but different hardware implementations to be substituted for one another without having to recompile the application (client code) that uses them. A service is globally and uniquely identified by its service class, which should be found in the service catalog, as discussed before. Once a service is marked `stable`, any changes to it must not break backward compatibility, as it may not be possible to update the firmware on devices that support the service.

The Jacdac service specification language has two related goals:

- to describe the resources that a Jacdac device acting as a server provides on the Jacdac bus, precisely defining the wire formats of data and requests/responses that the device will accept/provide;
- to provide meaningful abstract names and types for the above resources that enable client programming against a service at a high-level of abstraction.

In support of the two above goals, Jacdac provides tools that automatically generate C headers from service specifications; these headers parameterize the Jacdac runtime, which provides a suite of C helper functions for writing firmware. Other tools automatically generate client libraries from service specifications in a variety of languages, as mentioned previously. For example, most of the code for the MakeCode extension for Jacdac (see the subdirectories, one per service, in https://github.com/microsoft/pxt-jacdac) was automatically generated (the top-level code in this repo is the Jacdac runtime, written by hand).

The remainder of this section details the core abstractions of commands/reports that provide the foundation of Jacdac, as well as the other abstractions (actions, registers, and events) that are defined in terms of commands/reports. A service specification is written using markdown, which allows for interspersing informal English explanations of the service interface with the (indented) formal declarations of commands, reports, actions, registers, events, and other entities described below.

3.1 Commands and Reports

Commands are requests to devices on the Jacdac bus and reports are responses from devices. Definitions of commands and reports have a uniform base structure of a *readable identifier*, *numeric operation code* and a *statically-typed payload*. On the command side, payloads serve as arguments (for example, to a register write command). On the report side, payloads often serve as "return values" (in the case of the register read operation, for example). Commands are distinguished from reports in a Jacdac packet by a flag, as discussed in Section 4.

⟨*cmdrep*⟩ ::= (‘command’ | ‘report’) ⟨*ident*⟩[‘?’] ⟨*type-opcode*⟩

⟨*type-opcode*⟩ ::= ‘:’ ⟨*core-type*⟩ ⟨*unit*⟩ ‘@’ ⟨*opcode*⟩ | ‘@’ ⟨*opcode*⟩ [⟨*record-type*⟩]

⟨*opcode*⟩ ::= ⟨*hex-literal*⟩ ⟨*core-type*⟩ ::= See Table 1

⟨*record-type*⟩ ::= ‘{’ (⟨*ident*⟩ ‘:’ ⟨*core-type*⟩ ⟨*unit*⟩)* ‘}’

Fig. 5. Syntax of commands and reports.

Table 1. Core types supported by Jacdac specification language. All types are little endian.

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
bytes	byte buffer (until end of packet)
string	UTF-8 encoded string (until end of packet)
string0	NUL-terminated UTF-8 string
bool	a single byte; 0 = false, true otherwise

While commands and reports are often paired, they need not be. For example, events are reports without an associated command. A command without a corresponding report is an instance of the “fire-and-forget” pattern (that is, a request that doesn’t have an associated response). As discussed in Section 6, these communication patterns are also found in TinyOS and WSDL; in both these systems, as well as Jacdac, requests and responses are asynchronous operations.

Figure 5 gives the basic syntax of commands and report. A question mark following the identifier of a command/report indicates that it is an optional feature of a service, otherwise the command/report must be implemented to conform to the service specification.

Each command and report can carry a (possibly empty) payload. The payload can either be a value of *<core-type>* (as enumerated in Table 1) or of *<record-type>*. In the first case, the core type follows a ‘:’ and precedes the *<unit>* and *<opcode>*; in the second case, the record type follows the *<opcode>* (we found this makes the specification more readable). [12] Each core type must be annotated with a unit (a subset of SenML [13])

Commands can be used to direct a device to take some action. Here is the simplest form of a command/report pair, used to direct a sensor to perform calibration, with a corresponding report to acknowledge that calibration has taken place:

```
1       command calibrate @ 0x02 { }
2       report calibrate @ 0x02
```

In the above example, both the command and report use operation code `0x02`, where the command requests calibration and the report is a response indicating that calibration is complete. Note that an empty record “`{}`” is the same as having no type, as shown in the report declaration above.

Commands and reports are different domains; the names and operation codes within each domain must be unique, but as the above example shows may be reused across domains. An implementation of a service must be robust to invalid commands, reports, and payloads. That

[12] Jacdac also supports a record that ends with a homogeneous sequence, which is not described further here.
[13] See https://www.iana.org/assignments/senml/senml.xhtml.

```
1  identifier:  0x141a6b8a
2  extends: _sensor
3
4  ro distance: u16.16 m { typical_min=0.02, typical_max=4 } @ reading
5
6  const min_range?: u16.16 m @ min_reading
7  const max_range?: u16.16 m @ max_reading
8
9  enum Variant: u8 { Ultrasonic = 1, Infrared = 2, LiDAR = 3, Laser = 4 }
10 const variant?: Variant @ variant
```

Fig. 6. Jacdac distance service (partial).

```
1  identifier: 0x1609d4f0
2  rw pixels: bytes @ value
3  const num_pixels: u16 # @ 0x182
```

Fig. 7. Jacdac LED service (partial).

is, while code generation may ensure that the client creates only valid commands, reports and payloads, other devices (potentially buggy ones) may generate invalid traffic. The Jacdac runtime performs a certain amount of runtime validation on packets, given the C header generated from a service specification. Other implementations will need to do their own validation.

Operation codes (16-bit) are partitioned into the following ranges based on the class of operation: action (`0x0XXX`), register read (`0x1XXX`), register write (`0x2XXX`), reserved (`0x3XXX-0x7XXX`), and events (`0x8XXX-0xfXXX`). An action is a command with an optional associated report of the same name and code, given with the following syntactic sugar:

```
command calibrate @ 0x02 { } report { }
```

The following sections give more details on registers and events.

3.2 Registers

Registers are used for exposing necessary device state and have three forms:

- *const* registers do not change until module reset (which may put it into a new mode), though they most often will represent constraints imposed by the hardware that are forever the same. Lines 6 and 7 of Figure 6 use `const` to specify the minimum and maximum range of a distance sensor.
- *read-only* (`ro`) registers can be used to expose the value of relevant sensors. The `distance` register declared at Line 4 of Figure 6 is an example of a read-only register.
- *read-write* (`rw`) registers are generally used to configure the hardware and assignments to them are idempotent. Figure 7 presents an excerpt of the LED service that declares a read-write register `pixels` at line 2 which is a buffer of 24-bit RGB color entries (one per LED pixel).

Register declarations are (mostly) syntactic sugar over the command/report abstraction, given by the following grammar:

⟨*register*⟩ ::= ⟨*register-modifiers*⟩ ⟨*ident*⟩[‘?’] ⟨*type-code*⟩

⟨*register-modifiers*⟩ ::= ‘`const`’ | ([‘`volatile`’] (‘`ro`’ | ‘`rw`’))

A register declaration translates to two commands (to read/write the register's value) and one report (the response to a read with the value of the register). A report is only issued for the read request; for a write request, the client must issue a separate read command to confirm the value written. Also note that the service is not guaranteed to return the value written, as the underlying firmware may clamp or modify the written value. Furthermore, another write command may interleave between a client's write and subsequent read. Trying to write to a read-only register will be ignored by the server.

A `ro`/`rw` register may be annotated as `volatile` indicating that its value may change independently of any activity on the Jacdac bus. That is, a volatile register's value may change based on physical environmental conditions outside of programmatic control (the sensor service's `@ reading` register is implicitly volatile). This enables a caching strategy for non-volatile registers that flushes the client cache whenever there is some write to the service. For volatile registers, cached values will generally become stale very quickly.

3.3 Events

A Jacdac server may perform some computation over the stream of data from the sensor it encapsulates to detect a pattern. Events are a mechanism for notifying clients when such patterns are identified. We have seen examples of events with no payloads in the accelerometer service of Figure 2 (`freefall`, `shake`, ...). Jacdac client libraries provide APIs so that an application program can subscribe to a service event of a particular device. An event may contain a payload, as shown in the grammar:

⟨*event*⟩ ::= ‘`event`’ ⟨*ident*⟩ ⟨*type-code*⟩

Events translate to reports (with no associated command) that are given special treatment at the protocol level to ensure reliable delivery, as detailed in Section 4. The event opcode is limited to eight bits, with the remaining bits of the report opcode used for a counter for reliable delivery.

3.4 Compile Time Declarations and Hints

For readability, the specification language provides declarations for naming of values and enumerations of values. Figure 6 shows a simple specification of a distance sensor. Line 4 declares a read-only register `distance` with type `u16.16` whose value is in meters (`m`), as well as two compile-time constants: `typical_min` and `typical_max`, used when visualizing data in the Jacdac dashboard (eg., for axis scale for plots). The notation ‘`@ reading`’ that ends line 4 assigns the specific code of the `reading` register common to sensors to the register `distance`.

Line 9 of Figure 6 declares an enumeration named `Variants`, listing the different kinds of distant sensors, which is then used in the optional register declaration of `variant` (optional declarations have a '?' trailing the name). Enumerations are meant to be future-extensible and are primarily informational (eg., used in the Jacdac dashboard to visualize the sensor).

In addition to the **volatile** and **const** modifiers on registers, which are hints to the client runtime, one can add the modifier **unique** to a command, which means that the command is not idempotent; that is, multiple invocations with the same payload are different from a single invocation.

4 PROTOCOL

This section visits the layers of the Jacdac protocol top-down, from the service layer to the transport and network layers. Device identification is introduced between the descriptions of the service (device-unaware) and transport (device-aware) layers. We illustrate how the protocol maps to the

client program of Figure 3 that uses the accelerometer and LED ring modules, represented by the accelerometer and LED services (Figure 2 and Figure 7, respectively).

4.1 Service Layer

The service layer deals with the commands and reports specified by the service specification, as detailed in the previous section. Commands and reports are just Jacdac packets, provided to the service layer by the transport layer via a simple API. Helper functions provide access to the packet data structure via the abstractions of registers, events, and commands/reports.

4.1.1 Control Service. For a device to be recognized on the Jacdac bus, it must run its own control service.[14] The logic for the control service is generic, parameterized by the set of services a device supports, and is part of the Jacdac runtime.

The main job of the control service is to send a report every 500 milliseconds that advertises the device's presence on the bus and the list of services (via service class numbers) it supports. The other devices (clients) on the bus can inspect these advertisements and subsequently communicate with the advertised services. The advertisement also includes several flags indicating various protocol-level capabilities of the device, as well as a "restart counter" that monotonically increases and can be used to detect a device restart.

The control service also offers a set of common commands that can be used to query/inspect a Jacdac device. For example, the `identify` command causes a Jacdac device to perform an action that allows a user to locate it, usually through blinking an LED.

4.1.2 Processing Commands and Reports. A device acting as a server (of a particular service S) will receive commands from the transport layer for S and send reports back (it may also initiate sending of reports on its own). A device acting as a client of a service S will send commands (via the transport layer) and receive reports back. A device may act in the roles of both a client and a server. The transport layer is responsible for routing commands/reports to and from the proper services. Services are addressed by 6-bit indices referring to the position of the service class (32-bit number), as listed in the advertisement packet. The zero index is reserved for the control service. A server will generally maintain device-specific state for each of the services that it supports, usually via an array indexed by the service index; in the simplest case, a device has only two services (the control service and, say, a button service), and an array is not necessary.

4.2 Device Identification, Roles, and the Role Manager Service

Jacdac device identifiers are 64-bits in length and are used to determine the sending or receiving device, and for devices to remember one another on the bus. The Jacdac protocol does not support the allocation of unique device identifiers. Instead, each device must be assigned (or assign itself) a 64-bit device identifier; once assigned, a device's identifier must remain constant. As long as identifiers are generated with appropriate entropy (i.e., using a random number generator), there is little chance of identifier collision. If we consider one trillion Jacdac networks size of 200 devices with randomly chosen 64-bit identifiers, the probability of an identifier collision in at least one of the networks is 0.1%. The device identifier can be programmed at the factory, or the device can generate the identifier by itself upon first boot and store it in non-volatile memory. In either case, an appropriate source of randomness should be used.

In the example program of Figure 3, the four services of the four modules are represented by fixed "role" names available in the MakeCode runtime for Jacdac. For example, the role name `accelerometer1` is a static instance of a Jacdac `SensorClient`, a client-side representation of a

[14] See https://microsoft.github.io/jacdac-docs/services/control/.

```
struct jd_packet_t {
  uint16_t crc;                // crc and following 2 fields are from frame
  uint8_t  flags;              // various flags (see #defines below)
  uint64_t device_identifier;  // sending/receiving device, per flags

  uint8_t  service_index;      // which service does this packet refer to
  uint16_t service_opcode;     // the operation within the service
  uint8_t  service_size;       // size of the service payload
  uint8_t  data[236];          // payload
}
```

Fig. 8. Jacdac packet structure (simplified).

sensor-based service, specialized for the accelerometer service. Jacdac's role manager service keeps a mapping from role names to device identifiers (and the index of a particular service on that device). The role manager will eagerly map names to unmapped devices' services, unless directed otherwise by the programmer. Until the role name `accelerometer1` is bound to a device identifier providing an accelerometer service, the event handlers `onFaceDown` and `onFaceUp` in the program will not fire. In the case of multiple modules with the same set of services, the programmer can direct the role manager service to explicitly control the mapping of names to device identifiers.

4.3 Transport Layer

The transport layer deals with Jacdac packets and is responsible for generating acknowledgements, routing a packet to the correct service, as well as reliable events.

Figure 8 presents a simplified view of a Jacdac packet. A packet contains only one device identifier (rather than both source and destination identifiers, as in IP). If the bit `0x01` in the `flags` field is set, the packet is a command packet and `device_identifier` is the destination device receiving the packet; otherwise, the packet is a report packet and `device_identifier` is the source device broadcasting information on the bus. Sometimes, report packets will be broadcast without a preceding command (most prominently, in the case of advertisements and events). The maximum packet size is 252 bytes, which limits the size of a service payload to 236 bytes, which we find is sufficient for communication and control of many sensors and actuators (Jacdac has support for pipes, not discussed here, for working with data that does not fit in a single packet).

An acknowledgement should be sent if the bit `0x02` in `flags` is set. The acknowledgement packet includes the CRC of the packet being acknowledged. Finally, to support broadcast to all services on the bus (regardless of device), if the bit `0x04` in the `flags` field is set then the `device_identifier` field of the packet will be interpreted as a service class number, and the packet will be dispatched to all services on the bus with that class number.

Besides the support for acknowledgements, the transport layer is similar to UDP [25], as no delivery guarantees are provided. Since the two packets for a command/report pair may be separated by other packets, we provide support in the Jacdac runtime (a client of the protocol) to wait for the response (report) to a request (command). Support for TCP-like reliability and ordering also can be added [26] (as done with Jacdac pipes, which are unidirectional reliable streams).

4.3.1 Events. The transport layer has special queue-based logic for reliable sending of events. To communicate a discrete event reliably, the transport layer sends two identical repetitions of the event packet after the initial packet, with a 20-100ms gap between them. As typical packet loss is well under 1%, this ensures packet reception. The gaps between repetitions are relatively large to limit problems with reception queues at the client being temporarily full (which in our experience is

the main cause of packet loss), or interrupts being temporarily blocked. The event packets contain a per-device counter, incremented for every event sent (but not for the repetitions). This lets the client process the events in the correct order, even if some are lost.

4.3.2 Running Example. Returning to the "night light" program of Figure 3, once the four program roles (`accelerometer1`, etc.) are bound to the four modules, the `face_up` and `face_down` events (reports) generated by the accelerometer module will be routed to the corresponding event handlers in the program. The programmer does not need to know about the details of the protocol, the device identifiers, or the low-level encoding of the accelerometer service (the operation codes).

The method `setAll` of the `ledStrip1` client fills a pixel buffer with the given color and sends a register write command (packet) to the LED ring module that writes the buffer to the `pixel` register of the device. Again, the client wrapper abstracts over the details of the protocol. Two other "`ChangedBy`" event handlers in the program await changes to the reading register of the light level and the slider/potentiometer. This is implemented in the Jacdac runtime by asking the server to stream register reading packets at specified frequency and generating an event on the client side when the value has changed by the specified amount.

4.4 Network Layer: Single Wire Serial

A Jacdac frame contains a list of Jacdac packets (of length at least one), which all share the same device identifier and flags. The frame also contains a cyclic redundancy code (CRC). The Jacdac single wire serial (SWS) protocol is used to transmit a Jacdac frame over the wire, and requires an MCU with following basic functionality:

- transmitting and receiving UART-style bytes at 1Mbaud in half-duplex mode (as is standard, bytes are 10 bits long and are composed of 1 start bit, 8 data bits, and one stop bit);
- a GPIO with an internal or external 10-50kΩ pull up and support for interrupts, implemented in hardware or in software by spin waiting;
- the ability to keep time, via instruction counting or a hardware timer;

The MCU is not required to have UART hardware—we implemented SWS on PADAUK 8-bit MCUs (US $0.03) via bit-banging in software.

Any Jacdac device can initiate a transmission at any time. Because of this, devices must assert control over the bus before sending any bytes. This is where SWS differs from a traditional half-duplex UART: a device wanting to start transmitting checks if it is not in the middle of reception and that the line is not currently low; only then does the device bring the line low for 11 to 15 microseconds (*start pulse*). A collision is possible if another device at about the same time also determines the line to be high, and pulses it low. The window for such a collision is typically a few clock cycles (under 1μs s), resulting in typical collision rate of 0.1% for fully-utilized bus.

After the start pulse, the device waits at least 50μs (to allow other devices time to set up reception) and starts UART transmission of bytes, followed by an *end pulse* of 11-15μs (such pulses are recognized as break "characters" by UART hardware making them convenient frame markers). The receivers also have upper time limits on gaps between the start pulse, bytes of transmission, and the end pulse. If these are exceeded, any data is dropped, and the receivers go back to waiting for start pulse. Thus, no condition disrupts the bus for very long.

5 EVALUATION AND DISCUSSION

This section evaluates and discusses the impact of the design decisions on the cost, generality, and performance of Jacdac. Jacdac offers a tradeoff compared to using traditional embedded communication methods like I2C and SPI. Jacdac brings ease of use: dynamic device discovery, hot-plugging, error resilience and standardized services (Section 5.2). This is paid for by using

additional MCUs (Section 5.1) and additional wire time (Section 5.4). We argue that the costs are small and the benefits large, enabling more programmers to participate in building embedded systems.

5.1 Cost

From the outset, Jacdac was designed to ensure that its hardware implementation would be low cost and flexible. The protocol can be implemented on 8-bit MCUs such as the PADAUK PMS150C, PMS171B and PMS131 which run at 8MHz and have 64-96 bytes of RAM and 1000-1500 words of program memory. These processors don't provide UART hardware support, so our implementation uses bit-banging implemented via cycle-counted assembly language. These processors cost as little as US $0.03 (Shenzhen pricing for 1k units or more). The first kit produced by KittenBot uses the PADAUK PMS131 for all seven modules.

In addition to the MCU itself, a handful of discrete components are required to interface to the Jacdac bus for reliable operation: an RLC low-pass filter, a clamping diode and two electrostatic discharge protection diodes. If a server is powered from the Jacdac bus it typically also requires a low-dropout linear regulator (~US $0.03, Shenzhen pricing).

For modules that use a very low-cost peripheral such as an LED or a push button, the total bill-of-materials (BoM) cost can be as little as ~US $0.10 in quantities of 1k units (Shenzhen pricing). More sophisticated services may need more expensive sensors and/or a more capable MCU. For many of our prototypes, we used the STM32G030F6P6 (8kB RAM, 32kB ROM; US $0.51 ST Micro list price for 1k quantities). Our cheapest Jacdac brain is based on the RP2040 MCU and has a BoM cost of ~US $1.50.

5.2 Generality

Jacdac is a platform, so it is natural to consider how well it can support a range of hardware peripherals and how difficult it is to extend the platform to support new hardware. There are about 80 different Jacdac modules (some using the same set of services, but with different underlying hardware) designed and deployed, by us and others. We have also created various Jacdac adaptors so that various compute devices can act as Jacdac brains (BBC micro:bit, Raspberry Pi, laptop/desktop).

Table 2 provides an overview of 22 services we created to support these modules, categorizes them and describes how much code was needed to implement the server code supporting them. As shown in the second column of the table, we classify services into four basic kinds:

- **UX-in** services are mainly for user interface where we expect a person to take some action, such as push a button, twist a knob, or move a slider;
- **sensor** services are mainly for monitoring the environment, though a number of them may be used for user input (in particular, the accelerometer, flex, and motion services);
- **actuator** services generally cause some sort of motion to occur, though this may not be visible to the user;
- **UX-out** services are mainly for presenting information to the user.

Not surprisingly, UX-in and sensor services are described mainly by a few read-only registers and some events, though their operating envelope may need to be characterized by a few constant registers. Actuator and UX-out services, on the other hand, make more use of read-write registers and commands. Most of the services' logic is implemented by well under 100 lines of C code, especially for sensors, which have a fairly simple structure given by the abstract sensor service. The accelerometer service is noteworthy for the variety of events it can raise; its implementation requires more code to identify the events.

Table 2. Selection of services (22 out of 96, divided into four broad kinds) and their characteristics: columns **rw**, **ro**, **const** give the number of read-write, read-only and constant registers in the service, while **cmds** (commands) and **events** count the number of those service members, respectively; **LOC** is the number of lines of C code to implement the service logic (on top of the Jacdac protocol runtime), and **Flash** is the number of bytes the compiled service code occupies. The source code of these services is in https://github.com/microsoft/jacdac-c/tree/main/services.

Service	Kind	rw	ro	const	cmds	events	LOC	Flash
Button	UX-in		2	1		3	81	348
Potentiometer	UX-in		1	1			72	292
Rotary encoder	UX-in		1	2			120	406
Switch	UX-in		1	1		2	48	152
Accelerometer	sensor	1	2	1		12	251	798
Air Pressure	sensor		2				26	56
Flex	sensor		1	1			54	176
Humidity	sensor		2	2			25	56
Illuminance	sensor		2				26	56
Light level	sensor		2	1			72	292
Motion	sensor		1	3		1	51	184
Soil moisture	sensor		2	1			72	292
Temperature	sensor		2	3			26	56
TVOC	sensor		2	2			26	56
UV index	sensor		2	1			26	56
Motor	actuator	2		3			136	473
Relay	actuator	1		2			62	182
Servo	actuator	5	1	4			119	422
Buzzer	UX-out	1			2		87	228
Dot Matrix	UX-out	2		3			84	204
LED	UX-out	3	1	6			134	620
Vibration motor	UX-out			1	1		77	246

Figure 9 shows a service specification for a simple buzzer service and the C code of the integrated service and driver. The C code includes a header file for the buzzer service (automatically generated), as well as a header file of helper functions for working with Jacdac services. This reduces the amount of new code that needs to be written to a bare minimum. In particular, the developer of a new service only needs to implement several required functions, as shown in Figure 9, without having to know the details of the Jacdac protocol; they focus mainly on what state needs to be updated and what responses need to be generated for each function. Command line tools are available for creating the necessary header files locally from a new service specification. To make a new service widely available requires sending a pull request to the Jacdac team.

At its upper-edge, the service code uses the Jacdac runtime to communicate in the language of commands and reports. At the lower-edge, it communicates with specific hardware. The services are parameterized in one of three ways, based on the nature of the underlying hardware:

- **GPIO**: many modules have very simple hardware that can be accessed directly via general-purpose input/output (GPIO) pins; in these cases, the service initialization routine is parameterized by a C struct providing pin mapping and other domain-specific information (services: button, buzzer, rotary encoder, switch, motion, servo, relay, motor);

```
// jacdac/services/buzzer.md
identifier: 0x1b57b1d7
rw volume = 1: u0.8 / @ intensity
command play_tone @ 0x80 {
    period: u16 us
    duty: u16 us
    duration: u16 ms
}

// jm-v4.0/profile/buzzer-89.c
#include "jdprofile.h"
FIRMWARE_IDENTIFIER(0x3ff4e45b,
  "JM-Buzzer 89 v4.1");
void app_init_services() {
    buzzer_init(PA_4);
}

// jacdac-c/services/buzzer.c
#include "jd_services.h"
#include "jacdac/dist/c/buzzer.h"

struct srv_state {
    SRV_COMMON;
    uint8_t volume;
    // ^^^ see REG_DEFINITION()
    uint8_t pin;
    uint32_t end_tone_time;
    uint16_t period;
};

// jacdac-c/services/buzzer.c contd.
// define registers mapping to 'state'
REG_DEFINITION(buzzer_regs, REG_SRV_COMMON,
  REG_U8(JD_BUZZER_REG_VOLUME), // maps to state->volume
)
void buzzer_handle_packet(srv_t *state, jd_packet_t *pkt) {
  if (pkt->service_command == JD_BUZZER_CMD_PLAY_TONE &&
       pkt->service_size >= 6) {
    jd_buzzer_play_tone_t *d = (void *)pkt->data;
    state->end_tone_time = now + d->duration * 1000;
    state->period = d->period;
    d->duty = (d->duty * state->volume) >> 8;
    jd_pwm_pin(state->pin, state->period, d->duty);
  } else service_handle_register_final(state, pkt, buzzer_regs);
}
void buzzer_process(srv_t *state) {
  if (state->period && in_past(state->end_tone_time)) {
    jd_pwm_pin(state->pin, state->period, 0);
    state->period = 0;
  }
}
// bind buzzer_* functions to JD_SERVICE_CLASS_BUZZER:
SRV_DEF(buzzer, JD_SERVICE_CLASS_BUZZER);
void buzzer_init(uint8_t pin) {
    SRV_ALLOC(buzzer); // allocate 'srv_state'
    state->pin = pin;
    state->volume = 255; // volume defaults to 1.0
    pin_setup_output(state->pin);
}
```

Fig. 9. Jacdac service for a buzzer (buzzer.md) includes one command and one register. When using the `jacdac-c` framework, the firmware for every device is defined in one small file (buzzer-89.c here) which sets the numeric and string identifier and initializes specific services with relevant configuration information (the hardware pin number for the buzzer). The service implementation (buzzer.c) uses `#defines` generated from the service file (buzzer.h), defines the state structure (the value of the volume register and data of the currently playing sound), and implements callbacks to be run for every incoming packet, and on every "tick" (typically 10ms), as well as an initialization function. The packet function updates state for `play_tone` command and handles writes to registers via `service_handle_register_final()`, which uses the mapping defined by `REG_DEFINITION()` macro (the volume register is bound to the first byte (`U8`) of the state structure). The "tick" function stops the sound at the right time. The `SRV_DEV()` macro binds the three functions to the service class identifier of the buzzer (0x1b57b1d7).

- **analog sensor**: a few sensors provide a simple analog value, for which Jacdac provides an analog service based on an analog-to-digital converter (services: flex, light level, soil moisture, potentiometer);
- **complex sensor**: the remaining modules/services are generally more complicated sensors with their own integrated circuitry that is accessed via I2C or SPI, requiring driver code as shown in Table 3, typically under 200 lines of C code.

As shown in Table 3, we have used a variety of hardware sensors for the same service (namely, accelerometer, air pressure, temperature, and humidity). The Jacdac runtime provides class drivers for environmental sensors and other common classes of sensors that reduces the programming task to providing a C struct configuring the class driver.

5.3 Platform Code Size

For server code, there are two major implementations, one written in standard C99 and the other written in PADAUK macro-assembler. The C99 implementation is used mostly for Jacdac modules that use the STM32F0 and STM32G0 MCUs. The smallest in each family are STM32F030x4 with 4kB of RAM and 16kB of flash, and STM32G030x6 with 8kB of RAM and 32kB of flash.

Table 3. Jacdac-independent driver code for a variety of hardware sensors and actuators. **LOC** is the number of lines of C code and **Flash** is the number of bytes of the compiled code. The source code of these services is in https://github.com/microsoft/jacdac-c/tree/main/drivers.

Hardware	Description	LOC	Flash
ADS1115	Analog-to-digital converter	269	658
KX023	Accelerometer	137	327
KXTJ3	Accelerometer	118	443
QMA7981	Accelerometer	218	310
LSM6DS	Accelerometer + gyroscope	179	741
CPS122	Air pressure	111	396
LPS33HWTR	Air pressure	228	711
MPL3115A2	Air pressure	157	522
SHT30	Temperature and humidity sensor	123	537
SHTC3	Temperature and humidity sensor	123	636
TH02	Temperature and humidity sensor	125	503
DS18B20	Temperature probe	91	361
MAX31855	Thermocouple interface	71	292
MAX6675	Thermocouple interface	71	276
AW86224FCR	Vibration motor controller	110	172
LTR390UV	Visible + UV light sensor	137	517
SGPC3	TVOC (air quality) sensor	183	696

5.3.1 C99 Servers. As an example of code size, the C99 implementation of a temperature/humidity module with STM32G030 MCU includes:

- 0.6kB of service and driver code (as indicated in Tables 2 and 3);
- 0.9kB of generic sensor code;
- 4.8kB of service framework, control service, and various queues;
- 6.6kB of MCU-specific HAL code (RTC, ADC, I2C, UART, pins, startup);
- 0.3kB of glue code;
- 0.8kB of runtime support (integer division; the standard C library is not used);

for a total of 14kB of compiled code. At runtime, around 3kB of RAM are consumed, 1kB of which is debug logging buffer and 0.5kB is stack. The rest is mostly Jacdac queues.

The Jacdac implementation for STM32x0 also includes a bootloader, which allows for updating device firmware over Jacdac (from a web browser). The bootloader contains a very simplified Jacdac implementation and is 3kB in size. The bootloader must fit together with the module implementation in the flash of the MCU. Thus, for STM32F030x4 with 16kB of flash, we disable some optional features, resulting in firmware sizes of around 12-13kB.

5.3.2 PADAUK Servers. While the C99 code makes quite standard use of buffers and queues, the PADAUK implementation uses a very different approach.[15] As the PADAUK chips have 64-128 bytes of RAM, we only keep one packet buffer (of 24 bytes) for both reception and transmission. Only 6 bytes are allocated for stack, which is also used for interrupt handlers, so function calls are severely limited. The UART is implemented in software, with bit-banging.

After a packet is received it is immediately processed. A single bit of memory is allocated for every possible packet response (eg., a request to get the temperature register, would set a “temperature

[15] See https://github.com/microsoft/jacdac-padauk.

get pending" bit), with two additional bytes allocated for a single ACK. If any packet pending bits are set, and the transmission procedure successfully starts the low pulse, the remaining ~62μs before transmission of actual data are used to construct packet in memory based on the pending bit (which is cleared) and the state of the service.

In all, the implementations of various analog services, as well as a button all fit in the 1000-1500 words of one-time programmable (OTP) memory on the chip. Every word is a PADAUK instruction, so this would translate to around 2-3kB of ARM Thumb machine code. We also believe Jacdac could be implemented completely using custom silicon, without a general-purpose MCU, using strategies like the ones we used in our PADAUK implementation.

5.3.3 Clients. The size of client code vastly depends on the level of abstraction and programming language used. The simplest C implementation adds a few kilobytes, compared to server code. The MakeCode implementation, with a much higher abstraction level and less efficient translation from Static TypeScript to ARM machine code is tens of kilobytes. The TypeScript implementation for web browsers is hundreds of kilobytes. [16]

5.4 Performance

It is important to see Jacdac performance in light of its intended use: to create an embedded system from a small network of low-bandwidth sensors and actuators, with one to two handfuls of devices (modules and brain). It has been designed with robustness and ease of implementation in mind, rather than for low latency and high throughput.

5.4.1 Overhead. Sending a Jacdac packet using the Single Wire Serial (SWS) protocol of Section 4.4 takes, on average, 384μs of wire time plus 10μs for every byte of command payload. Often there is no payload, and otherwise it tends to be short, though it can be up to 236 bytes. This results from the SWS running at 1Mbaud (1 million bits per second, with 10 bits sent per byte due to start and stop bits), the wire arbitration protocol, and the packet structure.

For wire arbitration, SWS requires a start pulse (~12μs), ~50μs gap, data transmission, stop pulse (~12μs), and requires spacing between packets of 100-200μs (randomly chosen to avoid collisions). On average this comes to 224μs of overhead per packet. Jacdac packets have a 16-byte header, which includes the CRC, device identifier, other routing information, and command code (but not payload). This comes to 160μs. For example, an advertisement packet has at least an 8-byte payload, so takes 464μs, and is sent every 500ms. Thus, with 10 devices the bus is 1% saturated by advertisement packets, while 1000 devices would completely saturate the bus.

Typically, sensors that stream data are the largest users of wire time. A single sensor streaming at 2kHz would saturate the bus, so we advise streaming at not more than 50Hz. It is also possible to pack several readings in a single packet (or frame), to support sampling rates in the kHz range. As for latency, the time between a module deciding to send a packet, and the packet being received by the client is typically under 500μs. This is sufficient for most use cases, but may not be fast enough for hard real-time use.

5.4.2 Comparison to I2C and SPI. I2C typically runs at 100kHz or 400kHz (though faster modes are sometimes used). I2C latency at 100kHz is comparable to Jacdac on SWS, while I2C throughput at 400kHz is comparable to Jacdac using large payload sizes. I2C most often uses 7-bit addressing, so cannot support more than 127 devices, and in reality addresses are typically fixed for a given device type limiting usable networks to a handful. SPI can run at 50MHz or more, depending on MCU and peripherals. At these frequencies, the latency and throughput are much better than Jacdac over

[16] See https://github.com/microsoft/jacdac-c, https://github.com/microsoft/pxt-jacdac, and https://github.com/microsoft/jacdac-ts.

SWS. However, SPI typically requires separate addressing wires from the MCU, limiting network sizes to a handful of devices. Both SPI and I2C have severe limitations on cable lengths (typically under 30cm), whereas Jacdac on SWS can run over a few meters of wire.

5.4.3 Power Consumption. STM32G0-based sensors use around 50μA for the MCU and power regulation, plus whatever sensor is using (typically very little). Thus, a full system with a few sensors and a brain can be on the order of 1mA, which can run for months on a smartphone-sized battery. The low-cost PADAUK-based sensors use more current - around 1mA each. This is because they can't be put to sleep between incoming Jacdac packets, as they take a whole millisecond to wake up (unlike the STM32). This could be reduced dramatically with custom silicon support.

5.5 Security

The main attack surface introduced by the bus architecture of Jacdac is related to supply chains. A rogue device masquerading to be, say a button, could secretly listen to packets from other devices and even pretend to be the brain. Thus, sensitive services (typically ones related to the internet connection) should not be used on the same bus as untrusted devices. Typically, this is implemented by bundling the sensitive services with the brain and connecting them internally. We have recently introduced `restricted` modifier on packet specifications which instructs the brain to only accept them from a trusted connection (eg., USB to the computer) and never send them on the single-wire Jacdac bus. This allows for configuration of connection strings, WiFi passwords etc.

5.6 Working with Jacdac via the Web Browser

To get the most from Jacdac's service-based approach to working with sensors/actuators, we developed a web stack and site to make it possible to work with Jacdac without the need to download and install the tool chains or development environments usually associated with embedded development. A Jacdac module with USB-C connector is used to extend the Jacdac bus over USB so that a web browser that supports WebUSB can join the bus, sending and receiving Jacdac packets using the TypeScript port of the Jacdac runtime. We have created a set of React components that are parameterized by Jacdac service specification (compiled to JSON), upon which a Jacdac web site is based, as shown in Figure 4(b). The web-based service-aware tooling proved to be very useful for working with third-party hardware manufacturers based in China.

6 RELATED WORK

This section compares Jacdac with other approaches to composing embedded systems, interfacing with hardware, and connecting up microcontroller-based hardware.

6.1 TinyOS

Perhaps the most closely related work to Jacdac in terms of core abstractions, although with fairly different goals and end-users in mind, is TinyOS [15, 21]. TinyOS provides a framework for building embedded systems from a set of components, each described by a module interface in the nesC language [16]. Specifically, the Jacdac abstractions of commands and reports, which correspond to decoupled, asynchronous requests and responses, are quite similar to TinyOS "commands" and "events", which are termed a "split phase" interface. TinyOS is tightly dependent on the nesC programming language [16], in which the framework is written, while Jacdac adopts a neutral stance with respect to the client and server programming languages.

TinyOS's focus is on a modular framework that supports whole program optimization, which benefits from a static approach where all code—the client application and hardware-specific servers—is combined together [20]. Jacdac, on the other hand, focuses on dynamic discovery and hot-swapping to support rapid prototyping with hardware modules. The starting point for Jacdac is to separate the client code and server code on different MCUs, using a new wire protocol to join them on a bus. There are various benefits: true memory/fault isolation of client and server code, substitution of hardware modules without any change to client code or recompilation. TinyOS, in comparison, uses several techniques to prevent a component's memory from being corrupted by the code of a different component [8] and requires recompilation when hardware needs to be changed.

Of course, it also is worth noting that two decades separate TinyOS and Jacdac. Today, much lower-cost MCUs—nonetheless capable of running Jacdac—are now available, allowing us to place an MCU on each module. At the same time, modern US $1 MCUs are powerful enough to run Jacdac at the application level. This has led to a shift in how these devices are programmed: from assembly, through subsets of C, to full C and C++, and now towards high-level languages like Python and JavaScript. Jacdac is a continuation of this trend towards higher levels of abstraction in the hardware space.

6.2 IDLs for Distributed Computing and DSLs for Device Drivers

There is a long and rich history of interface definition languages (IDLs) for specifying the abstract interfaces to components/services in a language-independent manner, ranging from object-oriented models for distributed computing with (default) remote procedure call (RPC) semantics [14] to stateless models with four-way transmission semantics such as WSDL [37]. Jacdac follows the WSDL paradigm with respect to transmission options, simplified/adapted for embedded systems as discussed previously in Section 3.

Also relevant are domain specific languages (DSLs) for aiding the development of device drivers [7, 22, 23, 27, 34]. Devil [23] addresses the error-prone nature of writing the C programs that interface with specific hardware, especially as hardware documentation often is ambiguous or inaccurate, by providing a formal specification of the functional interface to hardware, from which C stubs can be generated. Devil models the interface to a device via three levels of abstractions: a physical address space of bytes; named registers; and finally, variables that cast registers into C types.

Many device driver DSLs follow this basic paradigm of interfacing to the C type system, as the goal is to aid the developer in writing correct C device drivers. In contrast, the goal of Jacdac specifications is to capture the functional interface to a wide class of devices at a higher-level of abstraction, while supporting a packet-based protocol (rather than a C interface). Towards this end, Jacdac provides a more expressive type system with support for units, uses a logical address space rather than physical, and provides support for actions and events, as well as registers.

6.3 Embedded Protocols and Toolkits

We analyze existing protocols with respect to three dimensions used to guide the design of Jacdac:

- *Standardized service interfaces*: Protocols such as USB (and Jacdac) abstract hardware via standard interfaces so that devices with similar functionality can act as drop-in replacements for one another. However, most of the interfaces provided by protocols for MCUs are low-level and do not provide this level of abstraction.
- *Communication paradigm*: While some communication protocols support only direct links between two devices (1:1), others define specific roles for devices on the network to reduce the complexity of peripherals therefore creating 1:N interconnects. To enable more flexible

peer-to-peer scenarios, some protocols adopt an N:M communication paradigm, as Jacdac does.
- *Dynamic device/service discovery*: Once a device has been connected, some protocols perform automatic service discovery to load the correct driver to operate a device. Without automatic service discovery, applications require prior knowledge of any software required to operate the device and its services. Applications also need to be recompiled to support new devices.

6.3.1 Wired Protocols. Widely used and highly efficient, I2C and SPI are the protocols of choice when connecting on-board peripherals to MCUs [9, 18, 30]. Driver writers use (statically assigned) peripheral addresses and adhere to individualized peripheral register maps to interact with and configure peripherals.

RS232, which is based on a universal asynchronous receiver transmitter (UART), has been extremely popular over the years. It is designed for point-to-point, full-duplex communications between two devices such as MCUs [29]. RS232 defines the format of bytes rather than the specification of packets, giving developers freedom over the packet structure. RS422 builds on RS232, but instead adopts a 1-to-many paradigm (1:N), and RS485 builds on both, applying a many-to-many (N:M) paradigm [1, 32].

Dallas 1-wire brings both communication and power to low-cost MCU-based peripherals connected to a single wire bus (with a second connection for ground) [4]. Each peripheral draws power from the bus, provided by a single host, storing charge that is used to temporarily power peripherals during communications.

USB (the Universal Serial Bus) [33] is designed for dynamically connecting peripherals to personal computers. Instead of providing just a physical transport like I2C, SPI, and RS232, USB contributes an entire stack that hides the complexities of address allocation and the transmission of packets to peripherals. The abstract driver model of USB enables the plug-and-play of peripherals and for driver reuse between devices, though at significantly higher cost than Jacdac.

While these protocols enable fast communications between the embedded device and peripherals, the development and debugging experience requires specialist tools and knowledge. Jacdac aims to simplify this experience while combining the dynamism of USB with the simplicity RS232.

6.3.2 Toolkits for Combining Embedded Devices and Peripherals. .NET Gadgeteer is a modular electronics toolkit that enables the integration of peripherals to a central MCU using a custom cable and socket system [36] supporting communication via UART, I2C and SPI. YAWN is based on UART and requires one host to control peripherals [35]. E-TAG and i*CATch peripherals are pre-programmed with unique I2C addresses [19, 24]. Other work enhances I2C using additional protocols to add on-the-fly address allocation [28]. While many of the above toolkits have succeeded in enabling the integration of embedded devices and peripherals, most of these solutions have worked within the constraints of static protocols and use higher-level APIs to simplify access to them, rather than changing the stack to support a true separation of concerns between client and server code, as done with Jacdac.

7 CONCLUSION

We have presented Jacdac, a platform for the dynamic composition of embedded systems from microcontrollers and hardware peripherals such as sensors and actuators. Central to the design of Jacdac is the specification of *services*, used to standardize the access to sensors/actuators and other hardware on the Jacdac bus, supported by a protocol that effectively separates application logic (on clients) from hardware (on servers), while enabling the dynamic discovery of devices and their services. As we have shown, a service architecture can be achieved at a very low cost and with acceptable overhead for many applications.

ACKNOWLEDGMENTS

We would like to thank Gabriele D'Amone, David Gakure, Joe Finney, Lorraine Underwood, Kobi Hartley, Paul Kos, and Matt Oppenheim for their many contributions to Jacdac. Also thanks to our hardware and education partners from the KittenBot, Forward Education, and MakeCode teams. Thanks to Angélica Moreira for her comments on the paper.

DATA-AVAILABILITY STATEMENT

An accompanying artifact is available [10].

REFERENCES

[1] Analog Devices Resource Library. 2000. Guide to Selecting and Using RS-232, RS-422, and RS-485 Serial Data Standards.

[2] Arm. 2017. The Arm Mbed IoT Device Platform. https://www.mbed.com/

[3] 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. https://doi.org/10.1145/3368856

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

[5] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michał Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN symposium on SPLASH-E*. ACM, 7–12. https://doi.org/10.1145/3358711.3361630

[6] Thomas Ball, Peli de Halleux, and Michał 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.). ACM, 105–116. https://doi.org/10.1145/3357390.3361032

[7] Christopher L. Conway and Stephen A. Edwards. 2004. NDL: a domain-specific language for device drivers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), Washington, DC, USA, June 11-13, 2004*, David B. Whalley and Ron Cytron (Eds.). ACM, 30–36. https://doi.org/10.1145/997163.997169

[8] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. 2007. Efficient Memory Safety for TinyOS. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems* (Sydney, Australia) *(SenSys '07)*. ACM, 205–218. https://doi.org/10.1145/1322263.1322283

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

[10] James Devine, Thomas Ball, Peli de Halleux, Michał Moskal, and Steve Hodges. 2024. *Jacdac: Service-based Prototyping of Embedded Systems (PLDI 2024 Artifact Evaluation)*. https://doi.org/10.5281/zenodo.10892762

[11] 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)*. ACM, 19–30. https://doi.org/10.1145/3211332.3211335

[12] James Devine, Michal Michał, 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. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 6, 3 (2022), 110:1–110:30. https://doi.org/10.1145/3550317

[13] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, Cham, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

[14] Chris Exton, Damien Watkins, and Dean Thompson. 1997. Comparisons between CORBA IDL & COM/DCOM MIDL: Interfaces for Distributed Computing. In *TOOLS 1997: 25th International Conference on Technology of Object-Oriented Languages and Systems, 24-28 November 1997, Melbourne, Australia*. IEEE Computer Society, 15–32. https://doi.org/10.1109/TOOLS.1997.681859

[15] David Gay, Phil Levis, and David Culler. 2005. Software Design Patterns for TinyOS. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Chicago, Illinois, USA) *(LCTES '05)*. ACM, 40–49. https://doi.org/10.1145/1065910.1065917

[16] David Gay, Philip Alexander Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003*

Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003, Ron Cytron and Rajiv Gupta (Eds.). ACM, 1–11. https://doi.org/10.1145/781131.781133

[17] Steve Hodges, Sue Sentance, Joe Finney, and Thomas Ball. 2020. Physical computing: A key element of modern computer science education. *Computer* 53, 4 (2020), 20–30. https://doi.org/10.1109/MC.2019.2935058

[18] Frédéric Leens. 2009. An introduction to I2C and SPI protocols. *IEEE Instrumentation & Measurement Magazine* 12, 1 (2009), 8–13.

[19] David I Lehn, Craig W Neely, Kevin Schoonover, Thomas L Martin, and Mark T Jones. 2004. e-TAGs: e-textile attached gadgets. (2004). http://hdl.handle.net/10919/80538

[20] Philip Alexander Levis. 2012. Experiences from a Decade of TinyOS Development. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 207–220. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/levis

[21] Philip Alexander Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason L. Hill, Matt Welsh, Eric A. Brewer, and David E. Culler. 2005. TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence*, Werner Weber, Jan M. Rabaey, and Emile H. L. Aarts (Eds.). Springer, 115–148. https://doi.org/10.1007/3-540-27139-2_7

[22] Fabrice Mérillon and Gilles Muller. 2001. Dealing with Hardware in Embedded Software: A General Framework Based on the Devil Language. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), June 22-23, 2001 / The Workshop on Optimization of Middleware and Distributed Systems (OM 2001), June 18, 2001, Snowbird, Utah, USA*, Seongsoo Hong and Santosh Pande (Eds.). ACM, 121–127. https://doi.org/10.1145/384197.384214

[23] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. 2000. Devil: An IDL for Hardware Programming. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (San Diego, California) *(OSDI'00)*. USENIX Association, USA, Article 2. http://dl.acm.org/citation.cfm?id=1251231

[24] 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. https://doi.org/10.1145/1753326.1753393

[25] Jon Postel et al. 1980. User datagram protocol. Internet Standard RFC 768.

[26] Jon Postel et al. 1981. Transmission control protocol. Internet Standard RFC 793.

[27] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic device driver synthesis with termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 73–86. https://doi.org/10.1145/1629575.1629583

[28] Rajesh Sankaran, Brygg Ullmer, Jagannathan Ramanujam, Karun Kallakuri, Srikanth Jandhyala, Cornelius Toole, and Christopher Laan. 2009. Decoupling interaction hardware design using libraries of reusable electronics. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*. ACM, 331–337. https://doi.org/10.1145/1517664.1517732

[29] Dallas Semiconductor. 1998. Fundamentals of RS-232 serial communications. (1998).

[30] Philips Semiconductors. 2000. The I2C-bus specification. *Philips Semiconductors* 9397, 750 (2000), 00954.

[31] Charles R. Severance. 2014. Massimo Banzi: Building Arduino. *IEEE Computer* 47, 1 (2014), 11–12. https://doi.org/10.1109/MC.2014.19

[32] Manny Soltero, Jing Zhang, Chris Cockrill, et al. 2002. 422 and 485 standards overview and system configurations. *Texas Instruments Application Report* (2002), 1–33.

[33] Universal Serial Bus Specification. 2000. Revision 2.0.

[34] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. 2005. HAIL: a language for easy and correct device access. In *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, Wayne H. Wolf (Ed.). ACM, 1–9. https://doi.org/10.1145/1086228.1086230

[35] Jan Thar, Sophy Stönner, Florian Heller, and Jan Borchers. 2018. YAWN: yet another wearable toolkit. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers*. ACM, 232–233.

[36] 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. https://doi.org/10.1007/978-3-642-31205-2_14

[37] World Wide Web Consortium (W3C). 2007. Web Services Description Language (WSDL) Version 2.0. https://www.w3.org/TR/wsdl/. W3C Recommendation 26 June 2007.

Received 2023-11-16; accepted 2024-03-31