Jacdac: Service-based Prototyping of Embedded Systems

Thomas Ball
tball@microsoft.com
Microsoft
USA

Peli de Halleux
jhalleux@microsoft.com
Microsoft
USA

James Devine
devinejames@microsoft.com
Microsoft
UK

Steve Hodges
shodges@microsoft.com
Microsoft
UK

Michal Moskal
mimoskal@microsoft.com
Microsoft
USA

ABSTRACT

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 program, or other service, but independently managed and updated.

We propose, design, implement and evaluate a service-based approach to prototyping embedded systems called Jacdac. 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 wire protocol (bus). 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 a third-party manufacturer.

KEYWORDS

embedded systems, services, plug-and-play, microcontrollers

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 [Corcoran 2013; Leens 2009; Semiconductors 2000]. 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 software abstractions such as those provided by Arduino [Severance 2014], ARM's Mbed [ARM 2017], and TinyOS [Levis et al. 2005] provide higher-level APIs for programmers, the end result is 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.

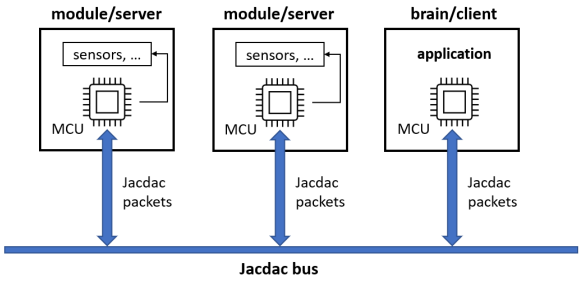


Figure 1: Jacdac devices (modules and brains) communicate with each other via packets sent over a bus.

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 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 [Dragoni et al. 2017].

We propose, design, implement, and evaluate a service-based approach to *prototyping* embedded systems called Jacdac. Our core 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 is novice programmers as well as programmers who are not familiar with embedded systems.

A service specification language, designed especially for embedded systems, is a key contribution of our work, along with a host of specifications for a variety of sensors and actuators. For example, as shown in the next section, an accelerometer service specification describes the basic interface to an accelerometer, regardless of the specific accelerometer hardware used.

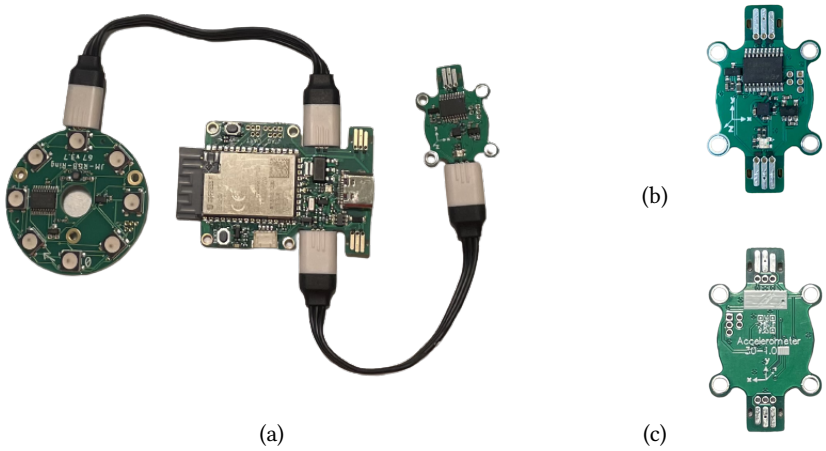


Figure 2: (a) A Jacdac system comprised of an ESP32 brain (center), connected via cables to an LED ring module (left) and an accelerometer module (right). The ESP32 brain has four Jacdac edge connectors, two of which are visible on the right side of the PCB. The LED ring has a single edge connector (not visible). (b) the front of the accelerometer module, which has two Jacdac edge connectors, allowing it to be daisy-chained.(c) Back of accelerometer module.

In Jacdac, service specifications provide a separation of concerns between application code (client) and driver code (server) that interfaces with hardware sensors and actuators. As shown in Figure 1, 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.

The Jacdac service abstraction 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 routing packets between Jacdac devices, and forwarding them 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 wire for 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 (for a module), or consumes the services (for a brain). 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. We support the Jacdac protocol for many sensors/actuators using 8-bit MCUs with 64 bytes of RAM that cost as little as US $0.03.

Figure 2(a) shows a small embedded system built from a Jacdac brain and two Jacdac modules, which are connected together to form a single 3-wire bus. The printed circuit boards (PCBs) of all three devices have one or more Jacdac 3-wire 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 (a separate paper will give more details of the design of the edge connector and cable).

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

- a large *library of service specifications*, with supporting server firmware, web-based simulators, and client bindings in a variety of languages[1]
- a growing *device catalog* of over 40 Jacdac modules (and a handful of brains) produced by us and others[2]
- 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 PADUAK MCU, the 32-bit ARM-based family of STM32x0 MCUs, as well as the ESP32 (which supports WiFi, TCP/IP and TLS) and RP2040 the last two being typically used as brains[3]
- additional implementations of the Jacdac protocol/runtime in Python, C#, TypeScript, and Static TypeScript [Ball et al. 2019], the subset of TypeScript supported by the MakeCode system [Devine et al. 2018], as well as a client library in each language (mostly code-generated) for each service;
- web site and tooling for compilation, flashing, monitoring, simulation, and debugging[4]

[1] https://microsoft.github.io/jacdac-docs/services/
[2] https://microsoft.github.io/jacdac-docs/devices/
[3] https://microsoft.github.io/jacdac-docs/ddk/
[4] https://microsoft.github.io/jacdac-docs/tools/

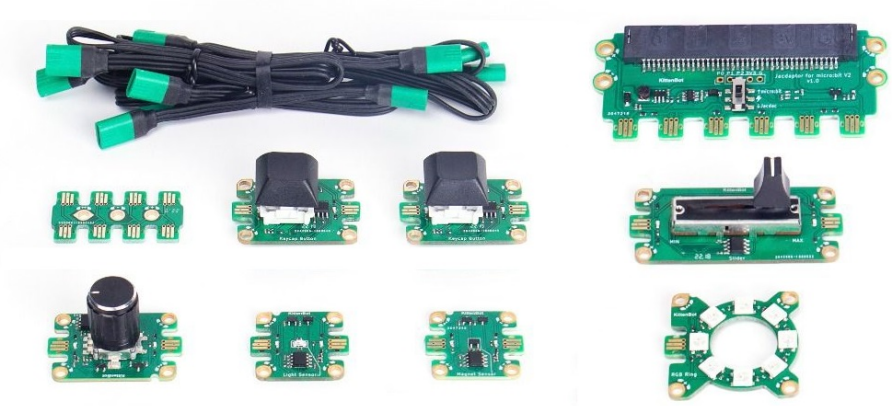


Figure 3: Jacdac modules created by KittenBot. From top left: Jacdac cables, Jacdac adapter for micro:bit, Jacdac hub, and seven Jacdac modules - two buttons, slider, rotary encoder, light and magnetic sensors, LED ring. Available at https://www.kittenbot.cc/collections/frontpage/products/kittenbot-jacdac-kit-for-micro-bit

Put altogether, the Jacdac stack effectively separates clients (brains) and server (modules) via services and 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. [Devine et al. 2022] As of the writing of this paper, a Jacdac kit of cables, seven modules and a Jacdac adapter for the popular micro:bit device, designed and manufactured by KittenBot, are commercially available, as shown in Figure 3.

In this paper, we focus on the design and technical implementation of the Jacdac platform and evaluate it with respect to the cost of the solution, its generality, and the overhead that the separation of client and server incurs. We also discuss the learnings from working with KittenBot to make Jacdac commercially available.

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 provider and/or client of services, allowing greater flexibility in application/system design than present in monolithic systems, and 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 4 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. Every Jacdac service is uniquely identified by a 32-bit identifier (line 3), also referred to as the *service class*, which is chosen randomly by the initial author of the service (we maintain a central repository of service specifications, which includes service class identifiers, and check for collisions in our tooling). 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.

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 base service specification. 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 data sheet for a sensor specifies its sensitivity (which may depend on environmental factors such as temperature), output resolution, and noise, among other characteristics. Line 16-17 of Figure 4 specifies 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 21-30 of Figure 4 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

```
# Accelerometer

    identifier: 0x1f140409
    extends: _sensor

## Registers

    ro forces @ reading {
        x: i12.20 g
        y: i12.20 g
        z: i12.20 g
    }

Indicates the current forces
acting on accelerometer.
    ro forces_error?: i12.20 g
       @ reading_error

## Events

    event face_up @ 0x85
    event face_down @ 0x86

Emitted when accelerometer is
laying flat in the given direction.

    event freefall @ 0x87
    event shake @ 0x8b
    ...
```

Figure 4: Jacdac accelerometer service (partial).

period. The `faceup` and `facedown` events are used in our running example.

Service specifications are stored in a GitHub repository which includes tools for generating various artifacts from the specifications, such as documentation, interface files for server code, and client libraries.

2.2 Accelerometer Module: Hardware

An accelerometer module we designed and produced appears in the upper-right of Figure 2(a). Figure 2(b) and (c) shows the module in greater detail. The large integrated circuit on the PCB 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 have a silk screen that show the direction of the three axes. The module has 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 the firmware is to make available the underlying specific hardware (the KXTJ3 accelerometer) via a Jacdac service. That is, the firmware abstracts over the particular accelerometer used by representing it using the Jacdac accelerometer service. The firmware allows the accelerometer module to join the Jacdac bus, advertise itself and that it supports 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 4, 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);
(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, as well as implementations of many services and drivers for various I2C/SPI hardware sensors, including the KXTJ3 accelerometer;
(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, as well as manufacturer and device name for easy user identification.

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 the MakeCode editors). 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 can also 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.

Figure 5 shows a small JavaScript program that works with the accelerometer and LED services. The Jacdac runtime (in this case, for the MakeCode programming environment) provides singletons (`accelerometer1` and `ledStrip1`) for working directly with modules exposing a single service (the mapping from names to device identifiters is discussed later).

The program has two event handlers for responding to the `face_down` and `face_up` events from the accelerometer module, where the first one sets the color of the LEDs to red and the second one sets the colors to green. As shown in the lower-left corner of Figure 5, Jacdac provides a simulator for the LED ring (the accelerometer simulator is not visible), which allows the user's program to be tested in the web browser before being deployed to the ESP32 brain.

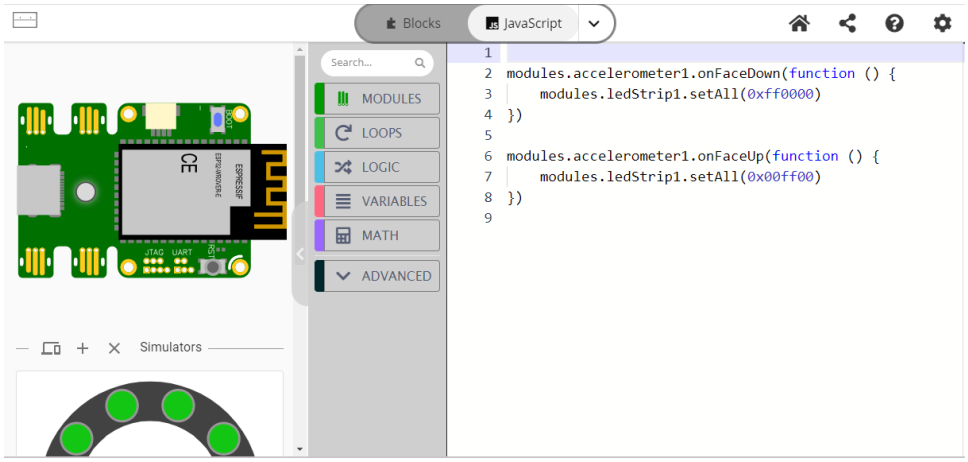


Figure 5: JavaScript program sets LED color based on accelerometer orientation (faceDown: red, faceUp: green). The coding environment is MakeCode, which we extended with support for Jacdac brains, as well as simulating Jacdac modules.

2.5 Dashboard and Digital Twins

Figure 6(a) shows the system from Figure 2 with the program deployed to the ESP32 brain and the accelerometer face up with the LED ring displaying green. The ESP32 brain is connected over USB to a host computer, extending the Jacdac bus over USB so it can be accessed from a web browser. Figure 6(b) shows the web-based Jacdac dashboard, which displays the digitals twins of the accelerometer module, LED module, and the ESP32 brain. Other tools available from the dashboard include a packet logger, details of device status, etc.

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

Service specifications describe the resources that a Jacdac device can share with other devices on the bus, by precisely defining the formats of data and requests/responses that can be interchanged between devices. 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 that uses them. For example, two modules with different accelerometer hardware can replace each other for an application (Figure 5) that depends on the accelerometer service (Figure 4).

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.

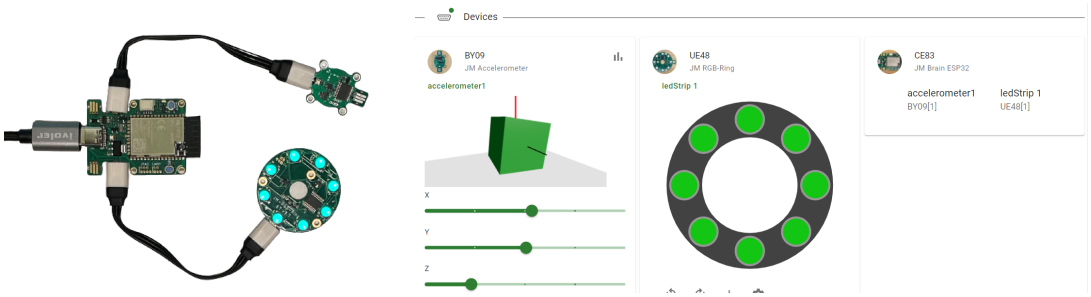
3.1 Service Members and Commands/Reports

A service specification consists of mainly of three kinds of members: register, action, and event declarations. We have seen examples of register and event declarations already in the accelerometer service. A question mark following a member's name indicates that the member is an optional feature of a service. By default, members are required (not optional) and must be implemented to conform to the service specification. More details on registers, actions, and events are given in the following subsections.

What is common to these declarations is that they are syntactic sugar over *commands* and *reports*, which have a direct translation into Jacdac packets. Commands are requests to devices on the Jacdac bus and reports are responses from devices. Commands and reports have a uniform base structure of an *operation code* and a *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).

While commands and reports are often paired, they need not be. For example, events are reports without an associated command. A command without a report is an instance of the "fire-and-forget" pattern (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.

Each command and report can carry a payload, which must be given a type. The type can be one of the core types, as listed in Figure 7, a record of fields (each with a core type), a homogeneous



(a) (b)

Figure 6: (a) system from Figure 2, programmed per Figure 5; (b) digital twins of accelerometer module, LED module, and ESP32 brain (from left to right).

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

Figure 7: Core types supported by Jacdac specification language. All types are little endian.

```
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
```

Figure 8: Jacdac distance service (partial).

sequence, or a record that ends with a homogeneous sequence. Each core type may be optionally annotated with a unit (a subset of SenML [5]) Lines 8-12 of Figure 4 declare the register `forces` to be a record with three fields (x,y, and z).

Command/report pairs are assigned unique operation codes based on their associated declaration's kind (registers, actions, and events each have their own range of numeric codes, not detailed here). Syntactic sugar is provided for common cases (such as sensor registers) so that the specification author need not remember the proper numeric ranges for operation codes. Commands are distinguished from reports in a Jacdac packet by a flag, as discussed in Section 4.

Various modifiers are available for specification members: `unique` commands are not idempotent (idempotent is the default semantics for all commands); `volatile` registers are described below.

3.2 Compile-time Declarations

For readability, the specification language provides declarations for naming of values and enumerations of values. Figure 8 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 code of the `reading` register to the register `distance`.

[5] https://www.iana.org/assignments/senml/senml.xhtml

Line 9 of Figure 8 declares an enumeration named `Variants`, listing the different kinds of distant sensors, which is then used in the optional register declaration of `variant` (option declarations have a '?' trailing the name). Enumerations are meant to be future-extensible (but not extensible by particular implementations of a service). Enumerations in services are primarily informational (eg., used in the Jacdac dashboard to visualize the sensor).

3.3 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 8 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, though there often is some conversion needed from the particular format supported by hardware to the register type. The `distance` register declared at Line 4 of Figure 8 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 9 presents an excerpt of the LED service that declares a read-write register `pixels` at line 6 which is a buffer of 24-bit RGB color entries (one per LED pixel).

```
        identifier: 0x1609d4f0

    A controller for small displays of individually
    controlled RGB LEDs.

        rw pixels: bytes @ value

    A buffer of 24bit RGB color entries for each LED,
    in R, G, B order.  When writing, if the buffer is
    too short, the remaining pixels are set to #000000;
    if the buffer is too long, the write may be ignored,
    or the additional pixels may be ignored.

        const num_pixels: u16 # @ 0x182

    Specifies the number of pixels in the strip/ring.
```

Figure 9: Jacdac LED service (partial).

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.

A register declaration is syntactic sugar for both a command (the request to read/write the register's value) and a corresponding report (the response with the new value of the register).[6]

3.4 User-defined Commands/Reports

As mentioned above, registers are syntactic sugar for a command/report pair, used to expose a devices' readable/writable memory. 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:

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

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.

3.5 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 4 (`freefall`, `shake`, ...). Jacdac client libraries provide APIs so that an application program can subscribe to a service event of a particular device. Events are reports that are given special treatment at the protocol level to ensure reliable delivery, as detailed in Section 4. An event may contain a payload.

[6] 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.

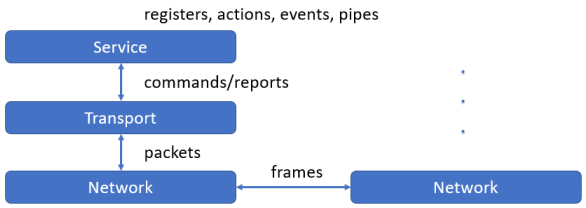


Figure 10: Three layers of Jacdac protocol

3.6 Pipes

Pipes are an application-level mechanism for establishing reliable one- and two-way point-to-point data links. Sometimes this corresponds to a data-stream in the underlaying service (eg., in a WiFi service a pipe may represent an open TCP/IP socket). More commonly, pipes are used to send a response that may not fit in a single packet. For example, again in a WiFi service, pipe may be used to return results of a WiFi network scan:

```
command list_known_networks @ 0x87 {
    results: pipe
}
pipe report network_results {
    channel: u8 {typical_min = 1, typical_max = 13}
    ssid: string
}
```

The client sends the command that creates the pipe. The server then reports each network (its name and channel number) in a separate packet sent over the established pipe, and then closes the pipe. The `pipe report` is scoped to share the pipe with the immediately preceding command.

3.7 Extending Services

If a hardware manufacturer wants to produce a device measuring cosmic background radiation, which is not currently covered by a service in the service catalog, they will need to define a new service (with a random service class for private testing). Our web tooling allows importing such new services for testing purposes. If the device is available publicly, the manufacturer is encouraged to submit the service specification to the catalog, so users can learn about it and client code can be auto-generated for various languages.

A more common situation is when an existing service does not cover all hardware features (eg., an accelerometer has additional de-noising settings). In this case, the manufacturer needs to implement the existing service, and also provide a *mixin service* and implementation covering the additional features. The existing service should still work in default settings for clients unaware of the mixin, while the mixin allows advanced users to access the special features.

4 PROTOCOL

This section visits the layers of the Jacdac protocol top-down, from the service layer, to the transport and network layers, as illustrated in Figure 10. 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 5 that uses the accelerometer and LED ring modules, represented by the accelerometer and LED services (Figure 4 and Figure 9, 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, actions, events, and pipes.

4.1.1 Control Service. For a device to be recognized on the Jacdac bus, it must run its own control service.[7] 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 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 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 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 5, the LED and accelerometer modules are represented by fixed "role" names available in the MakeCode runtime for Jacdac. The role name `accelerometer1` is a static instance of a Jacdac `SensorClient`, a client-side representation of a 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 and pipes.

Figure 11 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 `JD_FLAG_COMMAND` in the `flags` field is set, the packet is a command packet and `device_id` is the destination device receiving the packet; otherwise, the packet is a report packet and `device_idr` 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. Devices are also allowed to further restrict the maximum size of received commands.

An acknowledgement should be sent if the bit

`JD_FLAG_ACK_REQUESTED`

is set in `flags`. 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

`JD_FLAG_ID_IS_SERVICE_CLASS`

in the `flags` field is set then the `device_id` 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 [Postel et al. 1980], 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 [Postel et al. 1981], as discussed below.

[7] 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_id;         // 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
}

// COMMAND bit set: device_id is the receiver of a command
// COMMAND bit clear: the device_id is sender of a report
#define JD_FLAG_COMMAND 0x01

// an ACK should be issued with CRC upon reception
#define JD_FLAG_ACK_REQUESTED 0x02

// the device_id contains target service class number
#define JD_FLAG_ID_IS_SERVICE_CLASS 0x04
```

Figure 11: Jacdac packet structure (simplified).

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 Pipes. Pipes are unidirectional reliable streams. They are typically opened by the client sending a command to the server, which includes the client address and a *pipe port number*. The server then sends requested data as one or more packets to the client address, re-sending each packet (with 20ms gaps) until an ACK is received from the client (or timeout is reached). Pipe packets include the port number, as well as an incrementing counter. Typical use of pipes is to read a value of a "register" that does not fit in a single packet.

4.3.3 Running Example. Returning to the example program of Figure 5, once the program roles `accelerometer1` and `ledStrip1` are bound to the accelerometer module the LED ring module, the `face_up` and `face_down` events (reports) generated by the accelerometer module will be routed to the corresponding event handler 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.

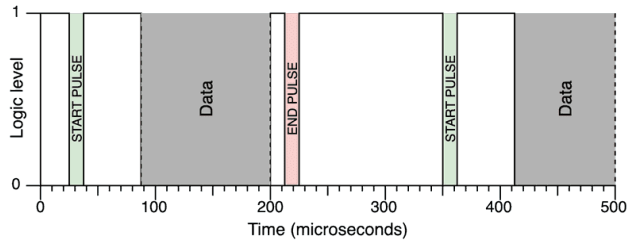


Figure 12: Jacdac transmission over the wire

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 (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 PADUAK 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 simple 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*), as shown in Figure 12. 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), 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 to 15 microseconds (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, there is no condition that 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.

A previous paper [?] evaluated the usability of Jacdac: we created and distributed 50 Jacdac kits, each with two brains and 8-10 modules, for an internal user trial with over 80 participants, over half of whom had non-technical backgrounds. They used Jacdac and MakeCode to build a variety of devices for the accessibility domain. Participants found the Jacdac modules easy to work with because of the immediate identification of modules and their services (via the integration of Jacdac into the MakeCode programming environment).

Since that paper, Jacdac modules have become commercially available via KittenBot. They produced 2000 kits containing seven Jacdac modules each, all of which passed our conformance tests, available via the Jacdac web dashboard (detai below).

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 (pre-pandemic Shenzhen pricing for 1k units or more). The KittenBot kit uses the PADUAK 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 or a push button, the total bill-of-materials (BoM) cost can be as little as ~US$0.10 in quantities of 1k units (pre-pandemic Shenzhen pricing). More sophisticated services may need more expensive sensors and/or a more capable MCU. For many of our prototypes we have 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. We have designed and deployed over 40 different Jacdac modules (some using the same set of services, but with different underlying hardware). We have also created various Jacdac adaptors so that various computers can act as Jacdac brains (BBC micro:bit, Raspberry Pi, laptop/desktop).

Table 1 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; such services may also be used for sensing (in particular, the rotary encoder service);
- **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 number of events it can raise; its implementation requires more code to identify the events.

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 struct providing pin mapping and other domain-specific information (services: button, buzzer, rotary encoder, switch, motion, servo, relay, motor);
- **analog sensor**: a few sensors provide a simple analog value, for which Jacdac provides an analog service based on an

Service	Kind	rw	ro	const	cmds.	events	LOC	Flash
Button	UX-in		2	1		3	69	300
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	248	758
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	49	162
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			85	204
Display	UX-out	3	1	4			144	624
Vibration motor	UX-out				1		85	168

Table 1: 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 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 compi service code occupies.

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 2, typically under 200 lines of C code (with no dependence on the Jacdac runtime).

As can be seen in Table 2, we have used a variety of hardware sensors for the same service (namely, accelerometer, air pressure, temperature, and humidity).

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 servers/modules using 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. In the past year we have not used STM32F0 as they are difficult and expensive to obtain, while the STM32G0 (produced using newer fabs with 90nm process) are readily available.

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 1 and 2);
- 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 compi 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. We had no need to further optimize the RAM usage of the C implementation, but it is possible.

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. Again, we have not looked at further size optimizations due to our switch to the larger and cheaper STM32G0 chips.

5.3.2 Paduak Servers. While the C99 code makes quite standard use of buffers and queues, the PADAUK implementation uses a very different approach. As the PADUAK 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

Hardware	Description	LOC	Flash
ADS1115	Analog-to-digital converter	265	727
KX023	Accelerometer	123	228
KXTJ3	Accelerometer	104	352
QMA7981	Accelerometer	218	290
LSM6DS	Accelerometer + gyroscope	161	648
CPS122	Air pressure	111	388
LPS33HWTR	Air pressure	203	652
MPL3115A2	Air pressure	151	494
SHT30	Temp. and humidity sensor	108	464
SHTC3	Temp. and humidity sensor	115	536
TH02	Temp. and humidity sensor	118	475
DS18B20	Temperature probe	91	357
MAX31855	Thermocouple interface	71	288
MAX6675	Thermocouple interface	71	272
AW86224FCR	Vibration motor controller	82	186
LTR390UV	Visible + UV light sensor	129	463
SGPC3	TVOC (air quality) sensor	177	650

Table 2: 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 compi code.

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 temperature register, would set a "temperature 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, implementation of various analog services, as well a button, fits 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 believe Jacdac could be implemented completely using custom silicon, without a general MCU, using strategies like the ones used in PADAUK.

The KittenBot modules all use a relatively high-end PADAUK (US$0.06 PMS131 with 96 bytes or RAM and 1500 words of ROM), allowing some of them to support two services at a time (rotary encoder exposes the built-in button as a separate button service), and others to use larger packets (8 s require 3 bytes each, resulting in 24 bytes payload and thus 40 byte packet).

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.

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.

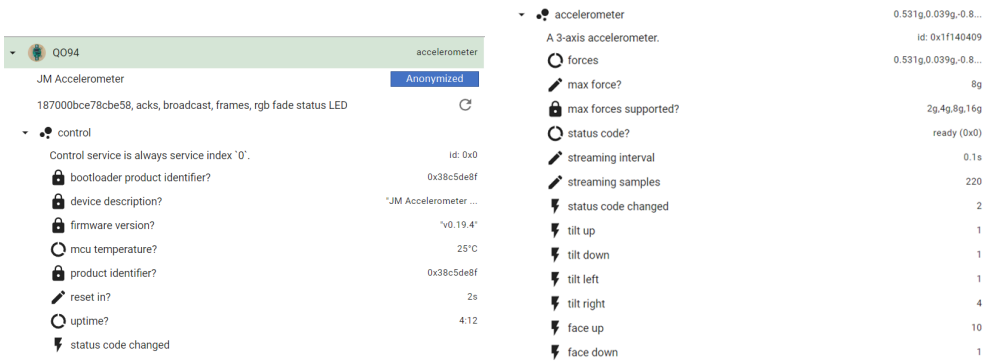
As for latency, the time between a module deciding to send a packet, and the packet being received by the client is typically under 500μs. This is sufficient for most use cases, but may not be fast enough for hard real-time use.

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 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.



(a) (b)

Figure 13: Device tree view of an accelerometer module: (a) shows the registers associated with the control service; (b) shows the registers and events associated with the accelerometer service.

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 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 (Chrome, Edge) 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 (compi to JSON), upon which a Jacdac web site is based, with the following features:

- the **device dashboard** displays digital twins of the Jacdac devices that are on the bus (see Figure 6);
- the **device tree** shows all the information about all connected devices and their services — Figure 13 shows a screen snapshot of the device tree for an accelerometer module;
- the **device tester** recognizes a device on the bus, and will run a set of automated and manual tests for its services, if any are available. Figure 14(a) shows the device tester when the ring module from KittenBot is present on the Jacdac bus.
- the **packet console** displays a filterable log of all the Jacdac packets sent over the bus.

The web-based service-aware tooling proved to be very useful for working with KittenBot, the hardware manufacturer based in China that produced the Jacdac modules shown in Figure 3. Figure 14(b) shows the ring modules being tested in the factory.

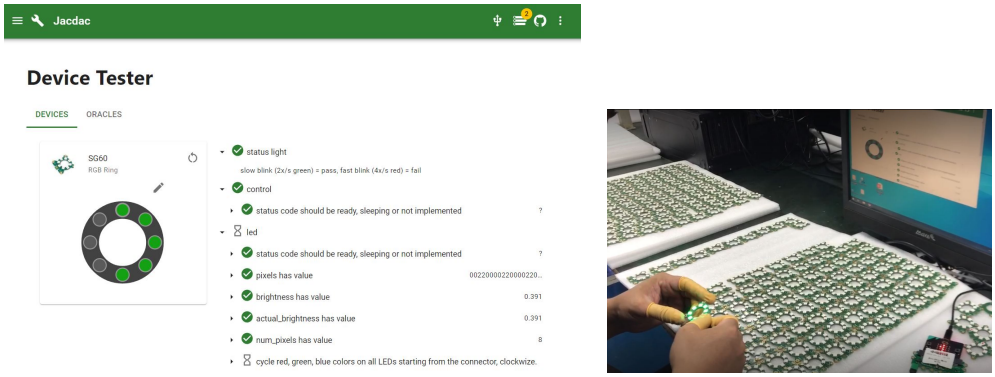
For each Jacdac service we generated a MakeCode extension (library) that allows all (web-based) MakeCode editors to work with Jacdac, as shown in Figure 5. The digital twins allow the MakeCode programmer to see what Jacdac modules have been attached to the bus; they are offered the option to load the extensions needed to work with those modules. Additionally, MakeCode's brain simulator allows the user's program to run in the web browser against the Jacdac modules on the bus, for a first-class debugging experience before compiling and flashing their application to the physical brain.

6 RELATED WORK

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

6.1 TinyOS

Perhaps the most closely related work to Jacdac in terms of core abstractions (though with fairly different goals and end-users in mind) is TinyOS [Gay et al. 2005; Levis et al. 2005], a framework for building embedded systems from a set of components, each described by a module interface in the nesC language. [Gay et al. 2003] Specifically, the Jacdac abstractions of commands and reports, which correspond to decoupled, asynchronous requests and



(a) (b)

Figure 14: (a)Web-based Jacdac device tester, showing tests for services on ring; (b) the device tester being used to test KittenBot ring modules at the factory.

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 [Gay et al. 2003], 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, application (client) and hardware-specific (servers), is combined together. [Levis 2012] 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 a number of benefits: true memory/fault isolation of client and server code, substitution of hardware modules without any change to client code (no recompilation). TinyOS, in comparison, uses a number of techniques to prevent a component's memory from being corrupted by the code of a different component [Cooprider et al. 2007] and requires recompilation when hardware needs to be changed.

Of course, it also is worth noting that two decades separate TinyOS and Jacdac. First, very low-cost and low-end, yet still capable of running Jacdac, MCUs are available, allowing us to place an MCU on each module. Second, application-level MCUs are much more powerful, while retaining a (US) one-dollar cost. This has led to a shift in how these devices are programmed: from assembly, via subsets of C, via full C and C++, towards high-level languages like Python and JavaScript, making embedded programming much more accessible. Jacdac is a continuation of that rise in abstraction levels to 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 [Exton et al. 1997] to stateless models with four-way transmission semantics such as WSDL [(W3C) [n.d.]]. Jacdac follows the WSDL paradigm with respect to transmission options, but 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 [Conway and Edwards 2004; Mérillon and Muller 2001; Mérillon et al. 2000; Ryzhyk et al. 2009; Sun et al. 2005]. Devil [Mérillon et al. 2000] 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: at the port level, the lowest level, there is a physical address space of bytes; on top of that, named registers are specified as constant width (untyped) bit vectors, at offsets off the base addresses of ports; finally, at the top-level, variables cast registers (or slices of registers and/or their concatenation) into an atomic C type (such as int), or are C structures consisting of fields similarly defined in terms of registers.

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 Construction 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 others 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 [Corcoran 2013; Leens 2009; Semiconductors 2000]. Driver writers use (statically assigned) peripheral addresses and adhere to individualized peripheral register maps to interact with and configure peripherals.

Almost as widely used as I2C and SPI, RS232, also known as UART (Universal Asynchronous Receiver Transmitter), is designed for point-to-point, full-duplex communications between two MCUs [Semiconductor 1998]. 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 [200 2000; Soltero et al. 2002].

One-wire brings both communication and power to low-cost MCU-based peripherals connected to a single wire bus [Awtrey 1997]. Each peripheral draws power from the bus, provided by a single host, storing charge that is used to temporarily power peripherals during communications.

USB (Universal Serial Bus) [Specification 2000] is designed for dynamically connecting peripherals to personal computers. Instead of providing just a physical transport like I2C, SPI, and UART, 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.

While protocols enable easy user interactions and fast, efficient communications between the embedded device and peripherals, the development and debugging experience requires specialist tools and knowledge. Jacdac aims to simplify the development and debugging process, and is inspired by the dynamism of USB, and the low-cost, simple, universal, and free-form communications of RS232.

6.3.2 Integrating 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 [Villar et al. 2012] supporting communication via UART, I2C and SPI. YAWN is based on UART and requires one host to control peripherals [Thar et al. 2018]. E-TAG and i*CATch peripherals are pre-programmed with unique I2C addresses [Lehn et al. 2004; Ngai et al. 2010]. Other work enhances I2C using additional protocols to add on-the-fly address allocation [Sankaran et al. 2009].

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 true service architecture can be achieved at very low cost and with acceptable overhead. Using modern web technologies, Jacdac also provides a universal development and debugging environment for beginner embedded systems developers.

While our focus was on supportying prototyping, Jacdac also makes it easier to manufacture tens or hundreds of identical instances of a device. Mass production techniques are typically cost-prohibitive at these quantities, while traditional prototyping with lots of wires is very difficult to repeat exactly. Of course, for large enough production runs it is more economical to use traditional mass production. Jacdac modules can be easily assembled due to standardized mounting holes.[8] The holes can also carry power and data, which allows for dispensing with cables. Jacdac brains can also include built-in sensors and actuators. These can be exposed internally, using the Jacdac service architecture, regardless if external Jacdac modules are present or not.

REFERENCES

2000. Selecting and Using RS-232 , RS-422 , and RS-485 Serial Data Standards.

ARM. 2017. The Arm Mbed IoT Device Platform. (2017). https://www.mbed.com/

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

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.

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

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)*. Association for Computing Machinery, New York, NY, USA, 205–218. https://doi.org/10.1145/1322263.1322283

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.

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*

[8] https://microsoft.github.io/jacdac-docs/ddk/design/ec30/

(Philadelphia, PA, USA) *(LCTES 2018)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3211332.3211335

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. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 6, 3 (2022), 110:1–110:30. https://doi.org/10.1145/3550317

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

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

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)*. Association for Computing Machinery, New York, NY, USA, 40–49. https://doi.org/10.1145/1065910.1065917

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

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

David I Lehn, Craig W Neely, Kevin Schoonover, Thomas L Martin, and Mark T Jones. 2004. e-TAGs: e-textile attached gadgets. (2004).

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

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

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

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.

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.

Jon Postel et al. 1980. User datagram protocol. STD 6, RFC 768, August.

Jon Postel et al. 1981. Transmission control protocol. STD 7, RFC 793, September.

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

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.

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

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

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

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

Universal Serial Bus Specification. 2000. Revision 2.0.

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

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.

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.

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