Enabling Intuitive and Efficient Physical Computing

James Devine

School of Computing and Communications
Lancaster University

This thesis is submitted for the degree of
Doctor of Philosophy

September 2020

To my love, Elizabeth, who has been my rock through the many ups and downs of this journey. And to my parents, who with their love and support, have given me many opportunities in life.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This thesis contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

I ensure that any user research contained in this thesis was conducted ethically and with consent. Any user research conducted on behalf of Lancaster University followed all internal processes and guidelines and is documented under ethics application number *FST16091*. Where external organisations presided, I guarantee that user research was conducted in line with their processes and guidelines.

James Devine
September 2020

Acknowledgements

First and foremost, I would like to acknowledge Microsoft Research (Cambridge) and Lancaster University for providing me with the funding to complete my PhD. I would like to especially thank Steve Hodges for procuring this funding at short notice.

I would like to thank Joe Finney for the opportunities he has provided me and his sage guidance over the years. Thank you for encouraging me to do what is right, rather than what is easiest. I would also like to thank my close collaborators: Tom Ball, Steve Hodges, Peli de Halleux, Michal Moskał, and Teddy Seyed. Who with their support, encouragement, and wisdom, shaped my thinking and the contributions of this thesis. Extra special thanks must also be given to Angie Chandler and Jason Alexander who gave me feedback on large portions of this thesis.

I would also like to thank those who provided support and encouragement at a distance. The MakeCode team who acted as a nexus of support: Jacqueline, Abhijith, Sam, Richard, Daryl, and Joey. My fellow band members who helped to musically vent frustrations: Geoff Coulson, Nigel Davies, Amanda Ross, and Kevin Huggett. My office mates, who helped me to retain my sanity: Oliver, Kelly, Kathy, Matt, Christina, and Helena. And the many other members of Infolab21 but in particular: Adrian Friday, Mike Hazas, Matthew Broadbent, and Karen Coupe, who greeted me with a smiling face each morning.

Finally, I would like to thank my close friends and family. My partner Elizabeth who brought light to the end of each day. My parents, Janice and Colum, and my siblings, Cameron and Emily, who helped me to remember home. My extended family Lorraine, Michael, Daniel, Mark, and Matthew, who acted like a second home. And my close friends Sam, Hugh, and Dan, who regularly provided comic relief through the years.

Abstract

Making tools for technology accessible to everyone is important for diverse and inclusive innovation. Significant effort has already been made to make software innovation more accessible, and this effort has created a movement of *citizen developers*. These citizen developers have the drive to create, but not necessarily the technical skill to innovate with technology.

Software, however, has limited impact in the real world compared to hardware and here, *physical computing* is democratising access to technological innovation. Using microcontroller programming and networking, citizens can now build interactive devices and systems that respond to the real world. But building a physical computing device is riddled with complexity. Memory efficient but hard to use low-level programming languages are used to program microcontrollers, implementation efficient but hard to use wired protocols are used to compose microcontrollers and peripherals, and energy efficient but hard to configure wireless protocols are used to network devices to each other and to the Internet. This consistent trade off between efficiency and ease of use means that physical computing is inaccessible to some.

This thesis seeks to democratise microcontroller programming and networking in order to make physical computing accessible to all. It provides a deep exploration of three areas fundamental to physical computing: *programming*, *hardware composition*, and *wireless networking*, drawing parallels with consumer technologies throughout. Based upon these parallels, it presents requirements for each area that may lead to a more intuitive physical computing experience. It uses these requirements to compare existing work in the space and concludes that no existing technology correctly strikes the balance between efficient operation for microcontrollers and intuitive experiences for citizen developers. It therefore goes onto describe and evaluate three new technologies designed to make physical computing accessible to everyone.

Table of contents

Collaborations

Much of the work in this thesis was created whilst collaborating with other researchers and organisations. This foreword is intended to clarify and outline these collaborations.

CODAL was created in close collaboration with Microsoft Research. Microsoft Research acted as one of the end users for the work and defined a number of requirements for the work to fulfil. My supervisor, Professor Joe Finney acted in an advisory capacity throughout its development.

JACDAC was created as a result of an internship at Microsoft Research where I was tasked with creating a protocol for on-the-go multiplayer gaming. Initial development of the protocol occurred in isolation with others acting as a sounding board for ideas. A later internship with Microsoft research led to the deployment of JACDAC at a fashion show in Brooklyn New york. This stage required close collaboration with researchers from UC Calgary and Microsoft Research.

Finally, Droplet was created as part of the Energy in Schools project. In this project, I collaborated with Samsung to create an educational IoT solution. My focus within the project was to create a reliable yet intuitive networking protocol for teachers and students to use and connect to the Internet. As part of this remit, I helped to define server infrastructure and APIs suitable for consumption by the micro:bit. Development of Droplet itself occurred in isolation.

Publications

[101] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli de Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale. The BBC micro:bit—from the UK to the World. *Communications of the ACM*, 2020

[143] James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. *Journal of Systems Architecture*, 2019

[142] James Devine, Joe Finney, Michał Moskal, Peli de Halleux, Thomas Ball, and Steve Hodges. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. 2018

[227] Kathy New, James Devine, Taylor Woodcock, Sophie Beck, Joe Finney, Mike Hazas, Nick Banks, Karen Smith, and Tim Bailey. Energy in Schools: Promoting global change through social technical deployments. *In Living in the Internet of Things: Harnessing Economic Value*, 2019

[258] Teddy Seyed, Peli de Halleux, Michal Moskal, James Devine, Joe Finney, Steve Hodges, and Thomas Ball. Makerarcade: Using gaming and physical computing for playful making, learning, and creativity. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, page LBW0174. ACM, 2019

[191] Bran Knowles, Sophie Beck, Joe Finney, James Devine, and Joseph Lindley. A scenario-based methodology for exploring risks: Children and programmable IoT. In *Proceedings of the 2019 on Designing Interactive Systems Conference*, pages 751–761. ACM, 2019

[192] Bran Knowles Knowles, Sophie Beck, Georgia Newmarch, Joe Finney, and James Devine. IoT4Kids: Strategies for mitigating against risks of IoT for children. *In proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019

[190] Bran Knowles, Joe Finney, Sophie Beck, and James Devine. What children's imagined uses of the BBC micro:bit tells us about designing for their iot privacy, security and safety. *In Living in the Internet of Things: Cybersecurity of the IoT*, 2018

List of figures

List of tables

Chapter 1

Introduction

1.1 The rise of the citizen developer

Making technology more intuitive leads to greater opportunities for diverse and inclusive innovation. The data science community has thrived upon simpler modern scripting languages, like Python [248], and mathematically verified software libraries, like SciPy [287]. This makes programming intuitive to mathematicians and complex mathematics intuitive to non-mathematicians. PyTorch [234], Keras [131], and TensorFlow [91] bring clean abstractions and simple Application Programming Interfaces (APIs), allowing anyone to learn about, train or deploy machine learning models. Advances in cloud computing have enabled anyone to deploy software across the globe without significant infrastructure investment, creating opportunities for global innovation. Tools like Microsoft PowerApps [203] and Ionic Studio [167] leverage cloud computing to deploy web applications with global reach, created in environments that strive to build applications without writing a single line of code.

The tools above democratise access software development, allowing many citizens to transform the digital world. To transform the physical world however, *hardware* is required. Here, a similar trend of democratisation is taking place. Widely available *embedded development boards*—Printed Circuit Boards (PCBs) that feature a Microcontroller Unit (MCU) with exposed General Purpose Input/Output (GPIO)—are allowing citizens to interact with the physical world using microcontroller programming and networking. This combination—known as *physical computing*—makes it easier than ever before for citizens to build interactive devices that respond and interact with the real-world—*and they are doing just that*.

Citizen makers and hobbyists are using physical computing for fun and experimentation. Physical computing ecosystems, like Arduino [107], supply embedded development boards

with exposed GPIO and an accompanying programming Integrated Development Environment (IDE) with simple C/C++ APIs. Combined, Arduino makes it easy to interact with external circuits and modular peripherals to create physical computing devices. This more intuitive route to microcontroller programming and electronics continues to play a critical role in the *maker movement* [145], a movement that seeks to democratise access to physical computing.

Teachers are also using physical computing for computer science education. Embedded development boards, like the Raspberry Pi [279], primarily act as a personal computer with a full operating system and built-in Internet connectivity. Through exposed GPIO however, citizens can combine the physical world with the digital in order to build physical computing devices. This process has proven itself to be a valuable pedagogical approach for computer science education [232] and teachers with technical expertise are now using physical computing to teach students fundamental computer science concepts.

Citizen scientists and innovators are also using physical computing for scientific experimentation and innovation. Embedded development boards and ubiquitous wireless networking are enabling citizens to build physical computing devices for the Internet of Things (IoT) [99]. The IoT stems from the idea that through inter-networking devices (things) together we can create a smarter world. Applying physical computing to the IoT therefore creates new opportunities for innovation and empowers more people to solve real-world problems using technology.

This trend—democratising access to technological innovation and experimentation—is creating a movement of *citizen developers*. This thesis characterises citizen developers by the following properties:

Ubiquity Anyone, not just those with technical expertise, can have an idea that can transform society. Citizen developers are typically not professional software or hardware developers, but all have the drive to innovate and create. Harnessing this hunger for innovation with intuitive technology will allow problems to be solved on a grander scale, propelling society forward.

Diversity Citizen developers do not follow a single profile. They come from many walks of life, have diverse backgrounds and personal experiences, and specific domain expertise that does not typically intersect with technology. Making technical communities more diverse therefore brings new perspectives to existing problems, leading to more opportunities for innovation.

Creativity There are many domains yet to benefit from true technological democratisation. Introducing citizens to intuitive tools for technological innovation will reveal new and creative applications for technology. This is already happening across many domains and contexts, and there will be many more to come as further technological democratisation takes place.

1.2 Physical computing

Physical computing combines software and hardware to build physical computing devices and systems that can sense and respond to the real world [177]. Despite much effort to make physical computing more intuitive, the process is steeped in technical complexity and even now only the technically capable can partake. Complexity stems from a long history of valuing efficiency over simplicity which has led to a wealth of highly efficient, but hard to use technologies.

This section discusses the core technologies used to build and prototype physical computing devices. For each technology, we identify the mismatch between the needs of the citizen and the existing approaches in the space. We start with a discussion of embedded development boards (Section 1.2.1), progress onto how they are programmed (Section 1.2.2), and how citizens connect peripheral sensors to embedded development boards to extend the functionality of embedded boards (Section 1.2.3). We then discuss the technologies used to network physical computing devices to each other and to the Internet (Section 1.2.4).

1.2.1 Embedded development boards

Although designed as a means for embedded systems specialists to evaluate new microcontrollers, embedded development boards are now used to enable the flexible, rapid, and efficient prototyping of physical computing devices. An embedded development board is a Printed Circuit Board (PCB) with a re-programmable microcontroller and easy to connect General Purpose Input/Output (GPIO). An embedded development board becomes a physical computing device when it is connected to sensors and loaded with an application that enables interactivity.

There are many styles of embedded development board and one of the most recognisable style of board are those designed for singular device prototyping. *Prototyping* boards have no on-board peripherals and let users connect peripheral sensors via PCB mounted break out connectors. Peripherals are usually mated directly with connectors or with an intermediary breadboard via wires. The Arduino Uno and its modern equivalent, the Arduino Zero,

are popular examples of prototyping boards and are a core technology used in the maker movement [145].

As Moore's law [251] continues to slow and demand for Internet connectivity continues to rise, more embedded development boards are featuring 32-bit microcontrollers with built-in wireless networking. This style of board—*networked* embedded development boards—either feature a microcontroller with built-in wireless capability or make use of a dedicated co-processor to perform wireless communications. Networked development boards let users connect peripherals in much the same way as prototyping boards, but the end result of prototyping is usually an Internet connected physical computing device. This style of board lets users rapidly prototype devices for the Internet of Things (IoT) [99] and popular examples of this type of board are the Raspberry Pi Zero Wifi [78], Particle Xenon [75], and the ESP32 [59].

However, the style of embedded development board that provides the most accessible base for citizens are those that come with both sensors and networking built in. The focus of *integrated development boards* is therefore not necessarily to enable flexible and efficient prototyping, but to give a simpler user experience. Integrated sensors and wireless peripherals mean that a great many applications can be realised without the use of wires and breadboards. The Circuit Playground Express (CPX) and BBC micro:bit are great examples of this style of device, seeing large adoption as physical computing devices for computer science education [232].

1.2.2 Programming

Programming languages, in the context of physical computing devices, are used to write applications for microcontrollers mounted on embedded development boards. Languages range in complexity from low-level and intricate programming languages like C/C++, to high-level programming languages like Python [248] and JavaScript [152], and even simple visual programming languages like microblocks [125] and OzoBlockly [154]. Resulting applications define the purpose and functionality of a physical computing device.

Programming language complexity correlates with more efficient memory and processor utilisation. Low-level programming languages, therefore, offer the greatest levels of efficiency and it is for this reason that they reign supreme in the resource-constrained world of the microcontroller. It is significantly harder to write programs in low-level programming languages due to complex syntax and few language features (i.e. memory management). The light feature set and efficient program compilation to binary instructions, however, leads to

efficient processor utilisation and real time behaviour, critical when building an *interactive* system.

Higher level languages offer greater usability at the expense of efficiency and in recent years, popular higher-level programming languages like Python and JavaScript have been re-written for resource-constrained microcontrollers. The flexibility and feature set of these languages has been shown to make programming more intuitive, and their move to microcontrollers is expected to have a similar effect. Programs written in these languages are compiled to byte-code and are interpreted—converted from cross platform byte-code to binary instructions—at run time by microcontrollers. Interpreting byte-code adds significant memory and execution overhead. This has the side effect of increasing energy consumption through greater processor utilisation and decreasing real time behaviour through increased execution time.

Visual programming languages are a type of higher level programming language that are more intuitive than text-based programming languages. They are now a proven tool for computer science education and are used by educators to teach the fundamental concepts of computer science. Scratch [245] is one of the most prominent and intuitive visual programming environments in use today and it engages young minds through the creation of on-screen 2D games. Scratch lets students combine pre-defined blocks of code to create programs, allowing them to focus on the structure of code rather than the syntax. As of May 2020, the TIOBE index reports Scratch as the 19th most used programming language in the world, with one million projects created every month [274].

Development environments for the above programming languages have begun to move to the web, and web browsers are quickly becoming the modern development environment of choice. The web browser lets users access programming from any Internet capable device, making software innovation and experimentation more intuitive to all. Teachers and students can now create on-screen games with Scratch 3.0 without installing any software, and embedded systems professionals and makers can write C/C++ applications for microcontrollers using ARM mbed [97] and Arduino Cloud [128] without a local compilation tool chain.

Despite these advances, microcontroller programming is currently beyond the reach of many citizen developers.

Citizen developers require intuitive programming. With little technical knowledge, the programming languages in use today are too complex for citizen developers to use. Regardless of efficiency, text-based languages like C/C++, Python, and JavaScript have been shown to have a high barrier to entry compared to visual programming languages like Scratch. The

event-based and visual programming paradigms offered by Scratch is evidence that these programming abstractions make programming intuitive to all.

Citizen developers require efficiency. Interactive devices may be powered from battery and use any embedded development board as their foundation. Memory and energy efficiency is therefore of great importance. The use of simpler programming abstractions and higher level languages however, comes at the cost of memory efficiency, energy efficiency, and real time behaviour. Resource constrained microcontrollers, and their use in interactive physical computing devices, however, demand all three.

Citizen developers require installation-free programming. Lack of technical knowledge means that software installation is a barrier for citizen developers. The recent trend towards web-based development environments is evidence that installation-free programming is key to making programming intuitive to all.

1.2.3 Hardware composition

Hardware composition is the process of connecting external sensors and peripherals to embedded development boards. Buttons, accelerometers, and displays are all examples of peripherals that can be connected to embedded development boards, and their use specialises an embedded development board to a particular use case. Peripherals are either mounted directly or connected externally to embedded development boards, and communicate with microcontrollers using a combination of conductive copper traces and wiring. Widely available external hardware modules fitted with sensors and peripherals can be mated directly to PCB mounted connectors for rugged device composition. Connected sensors and peripherals can then be used by microcontroller applications to turn an embedded development board into a physical computing device.

Complexity of communication correlates with peripheral capability. Buttons use simple digital communications to indicate button presses and change the logical line level between GND and VCC when pressed (depending on electrical orientation). Simple analogue sensors, like thermometers, express a range of values by modifying line voltage with respect to sensed value. For example, the maximum value of the thermometer would be expressed as VCC and the minimum as GND, with other values falling between this range. More complex sensing tasks, however, require a more expressive means of communication. Accelerometers, for example, each report acceleration in the x, y, or z axis. Whilst each axis could be expressed

as an individual analogue values over separate GPIO, it is far more efficient and extensible to standardise electrical communications over a single medium.

Wired protocols standardise electrical signalling and software interfaces for communicating between advanced peripherals and microcontrollers. Each wired protocol is designed for a particular purpose, ranging from dynamic protocols designed to make it easier to connect devices to personal computers like USB [264], to highly static protocols, like I2C [255, 137] and SPI [199], that are designed to efficiently connect advanced peripherals to microcontrollers.

The more dynamic the protocol, the more complex the implementation and for microcontroller manufacturers, this translates to more silicon and an increase in cost. Dynamic protocols like USB are therefore considered a premium feature unlike simpler protocols, such as I2C and SPI, which are generally available on all microcontrollers.

Despite hardware composition being a dynamic process, disparity in protocol availability means that protocols designed for efficient communication between PCB mounted peripherals (I2C and SPI) are being used for composing physical computing devices. Iterative development practices however demand that protocols support connecting more than one of the same peripheral at a time and that peripherals can be dynamically connected and removed (hot plugged). Because of their assumptions of static environments, neither of these use cases are supported by I2C and SPI, artificially constraining the hardware composition process.

Making hardware composition more intuitive and dynamic has long been a goal of the ubiquitous computing research community. Using a mixture of the protocols above and many more beyond that, modular toolkits have been proven to make hardware composition more intuitive [166, 285, 176, 228, 123]. However, despite advances put forward by these toolkits, many have not moved beyond research prototypes. Hardware composition, therefore, continues to be challenging for citizen developers.

The static and efficient protocols of the past are now hindering the citizen developers of the future.

Citizen developers require dynamic composition. Heavy use of I2C and SPI are making hardware composition more difficult through their assumptions of static and unchanging environments. A trend towards modular hardware in both industry and research means that hardware composition is becoming inherently more iterative and dynamic. Citizen developers therefore require protocols that better support dynamic, plug-and-play, hardware composition.

Citizen developers require simplicity. There are more restrictions and limitations to I2C and SPI than static design assumptions. These protocols require at least four wires to provide power and data to peripherals increasing the complexity of connecting peripherals together. Moreover, no more than one of the same peripheral may be connected to a microcontroller (without deep technical configuration), and only one microcontroller can be connected to a peripheral at a time. Increased infrastructure cost (wiring) and protocol restrictions make hardware composition difficult for citizens.

1.2.4 Wireless networking

The Internet of Things (IoT) is granting new opportunities for innovation and experimentation by connecting everyday devices and objects to each other and to the Internet. Prototyped physical computing devices are quickly becoming part of the IoT through more intuitive hardware prototyping, microcontroller programming, and integrated wireless networking. Wireless networking protocols provide the means to interconnect objects and standardise wireless signals for universal communication between devices.

Bluetooth Low Energy (BLE) [113] has seen wide adoption as a point-to-point wireless protocol for connecting small form factor electronics to more capable devices. Small form factors are achieved through the minification of electronics, including batteries, and great care must therefore be taken during operation to consume as little energy as possible. Reducing energy consumption however is challenging when wireless protocols are involved, as increased radio activity directly correlates with increased energy consumption. BLE, therefore, is heavily optimised to minimise radio activity and its point-to-point operation and short range (30 metres) mean BLE is best suited to personal networking.

WiFi is perhaps the most ubiquitous wireless protocol in use today and is designed for high data throughput, low latency—and therefore real time—communication. It operates over a short range (30 metres) and with its wide adoption, Internet connectivity is but a password away. High throughput and low latency comes at the cost of energy efficiency however, and WiFi is therefore not well suited to battery powered IoT devices.

Low power mesh and ad-hoc networking protocols like Zigbee [187] have seen wide adoption in the IoT. Reduced energy consumptions comes from the use of different physical standards, like 802.15.4, and higher level protocols that apply strategies to reduce radio on time. Ad-hoc protocols allow for on demand network creation and extend the range of single-hop communications by allowing devices to forward packets through a network to

reach their destination. To enable such dynamism and flexibility without compromising energy efficiency, packets must be optimally routed across a network.

Competing strategies for routing packets across ad-hoc networks has fuelled decades of research [185, 278, 187, 146, 262]. Static routing approaches are highly energy efficient, but require a large amount of configuration and reduce network flexibility. Dynamic routing approaches trade some efficiency for increased flexibility by creating and maintaining routes between devices. Regular route maintenance requires additional transmissions that reduce overall energy efficiency. Flood-based approaches cause devices to repeat packets until a single message has been propagated through an entire network. Little to no routing state leads to highly flexible networking, but packet propagation incurs significant energy cost.

Recent research has highlighted a new approach, known as *concurrent flooding*. Concurrent flooding promises to vastly improve the energy efficiency of flood-based approaches and reduces radio on time by parallelising transmissions through simultaneously transmitting the same data. Since its introduction in Glossy [148], concurrent flooding has been shown to enable flexible and energy efficient networking.

Decades of research focussed on energy efficiency means that ad-hoc networking protocols are not well-aligned to the needs of the citizen developer.

Citizen developers require simplicity. Ad-hoc networking protocols are best suited to the iterative development and deployment practices of the citizen developer, but most require configuration and rely upon a fixed infrastructure to operate. This requires deep technical expertise that is incompatible with the majority of citizen developers.

Citizen developers require interactivity Many existing protocols are designed to suit highly specific scenarios. Most ad-hoc networking protocols are designed for low throughput, energy efficient operation. However, there is no fixed set of IoT applications that citizen developers create and many demand a level of interactivity that is not well catered to by current ad-hoc networking protocols.

Citizen developers still require efficiency. Prototyped physical computing devices are often deployed in environment on battery power and use low resource microcontrollers. It is therefore important that wireless ad-hoc networking protocols support interactive applications whilst maintaining energy and memory efficiency.

1.3 Research questions

As noted in the previous sections, for a long time, many citizen developers have been excluded from partaking in physical computing across the domains of making, the IoT, and education (and many more). This exclusive practice has not only stifled innovation but has reduced diversity in technical communities. Exclusivity is not a conscious practice, it is rooted in the complex process of building a physical computing device.

Complexity stems from a drive to make the fundamental building blocks of creating a physical computing device as efficient as possible. Memory and processor efficient low-level programming languages are used to create applications for physical computing devices. Efficient and statically designed wired protocols are used for interconnecting microcontrollers and peripherals during hardware composition. Highly energy efficient ad-hoc wireless networking protocols are used to bring Internet connectivity and local networking to prototyped physical computing devices. Ultimately, each technology *trades ease of use for efficiency*.

This thesis questions this trade-off and explores whether efficiency does need to come at the expense of ease of use. More specifically, it explores the following research questions (RQs) in the context of programming (RQ1), hardware composition (RQ2), and wireless networking (RQ3):

RQ1 What are the capabilities, characteristics and limitations of event-based visual programming languages when applied to microcontrollers? How do these languages' capabilities and performance compare with the state-of-the-art in supporting citizen developers?

RQ2 Do single wire approaches to modular hardware composition simplify hardware integration for citizen developers, and what are the performance implications of these approaches?

RQ3 Do concurrent flooding approaches simplify ad-hoc wireless networking for citizen developers and what are the performance implications of these approaches?

1.4 Contributions

This thesis contributes three technologies spanning the domains of programming, hardware composition, and wireless networking—the fundamental building blocks for prototyping physical computing devices. Each technology was designed to provide citizen developers with a more intuitive experience without compromising efficiency.

1.4.1 Programming (RQ1)

In Chapters 2 and 3 we derive three design requirements to make programming more intuitive to citizen developers:

P1 Visual programming

P2 Event-based programming

P3 Installation-free

Informed by P1—P3, in Chapter 4 we contribute **CODAL**, a runtime environment designed to enable higher-level programming languages to run efficiently on resource-constrained microcontrollers. Microsoft MakeCode leverages CODAL to create an intuitive, installation-free, visual and event-based programming environment for physical computing devices. We show that MakeCode and CODAL are more efficient than other approaches to running higher level languages on physical computing devices. Millions of citizen developers now intuitively create applications using MakeCode and CODAL every month.

1.4.2 Hardware composition (RQ2)

In Chapters 2 and 3 we derive three design requirements to make hardware composition more intuitive to citizen developers:

HC1 Dynamic connectivity

HC2 Dynamic device discovery

HC3 Hardware abstraction

Informed by HC1—HC3, Chapter 5 contributes **JACDAC**, a wired protocol designed to support the iterative development practices of citizen developers through dynamic hardware composition. We show that JACDAC is as efficient as existing protocols like I2C and that it can be applied to many microcontrollers without any additional cost. Through a field trial at a fashion show in New York, we show that JACDAC is intuitive for citizen developers.

1.4.3 Wireless networking (RQ3)

In Chapters 2 and 3 we derive three design requirements to make wireless networking more intuitive to citizen developers:

WN1 No configuration

WN2 No infrastructure

WN3 Supports interactivity

Informed by WN1—WN3, Chapter 6 contributes **Droplet**, a no configuration, no infrastructure ad-hoc networking protocol for physical computing devices. We show that Droplet is more supportive of interactivity at the expense of a small amount of energy efficiency when compared to BLE. Through a deployment in 30 schools across the UK, we show that Droplet is intuitive for citizen developers.

1.4.4 Guiding principles

Each of the above technologies were created by methodologically applying the following four Guiding Principles (GP):

GP1 *Intuitive*: Technologies must be easy to use for citizen developers with little experience of hardware and software development.

GP2 *General*: Technologies must be suitable for different application domains and the spectrum of embedded hardware.

GP3 *Extensible*: Technologies must support the easy addition of new functionality and interoperate with existing tools and standards.

GP4 *Efficient*: Technologies must be highly efficient, but not at the great expense of GP1–GP3.

1.5 Overview

Chapter 2 begins by providing background on the applications of physical computing and the core technologies required to build a physical computing device. Chapter 3 explores questions derived from our extensive discussion in Chapter 2 across the areas of programming, hardware composition, and wireless networking. Our findings in Chapter 3 motivate the contributions of this thesis, which we discuss in Chapters 4, 5, and 6. Finally, in Chapter 7 we summarise the contributions of this thesis and finish with some concluding remarks.

Chapter 2

Enabling technologies for citizen developers

This chapter provides a foundational understanding of the enabling technologies for citizen developers. Throughout each section we extract common trends and emergent requirements and needs of the citizen developer. Content is written to be accessible to a broad audience and experienced researchers may want to skip to the end of this chapter to learn the key findings.

We start by discussing the existing application domains of physical computing in Section 2.1, followed by an in-depth discussion of the embedded development boards used for physical computing (Section 2.2). We then broadly discuss programming languages (Section 2.3), hardware composition (Section 2.4), and wireless networking (Section 2.5), concluding with a summary of this chapter in Section 2.6.

2.1 Application domains

Physical computing is being applied everywhere. Hobbiest makers are building physical computing devices to learn, create, and innovate. Technologists are building interactive devices for the Internet of Things (IoT) to automate and optimise their daily routines. Educators are even using physical computing to create more engaging lessons for their students. This section discusses the specifics of how physical computing is applied to the maker movement (Section 2.1.1), the IoT (Section 2.1.2), and to Education (Section 2.1.3).

Fig. 2.1 A prototype of the Pebble smart watch created using physical computing [76].

2.1.1 The maker movement

The drive to make and create is innate, whether it be through profession or through hobby. The maker movement [145] is the embodiment of this idea, allowing people to create by tinkering, learning and having fun in the process. Though people can make and create using a variety of tools and materials, the maker movement has its roots in technology and seeks to make physical computing a fundamental skill intuitive by all.

"Making" is synonymous with physical computing, and refers to the process of creating a physical computing device. As illustrated previously, there are four technologies fundamental to this process: (1) an *embedded development board* with a reprogrammable microcontroller and exposed General Purpose Input/Output (GPIO); (2) the *programming languages and environments* used to create microcontroller applications; (3) *hardware composition* via wired protocols that enable interactivity between applications and sensors; and (4) wireless protocols that may be optionally used to network devices together. The combination of these technologies produces a physical computing device—an interactive device that responds to real world stimuli.

Making as a process has permeated across many different domains and now forms the basis of product innovation, the backbone of scientific research projects, and even the foundation of interactive fashion garments.

Making to innovate

The democratisation of electronics and microcontroller programming brought about by the maker movement has lowered the barrier to entry to product innovation [206]. Startups and incubators now borrow technology and processes from the maker movement to construct physical computing devices that solve real-world problems. Convincing prototypes yield investment from venture capitalists spawning new businesses that provide an income to creators.

Crowdfunding platforms, like Kickstarter [23] and Indiegogo [12], enable innovation to occur at a far greater scale and granularity than can be achieved through venture capitalist funding. Crowdfunding takes place on websites where individuals can choose to 'back' projects which provide solutions for small-scale problems that would otherwise not be of interest to venture capitalists. Most electronics-based projects use technologies from the maker movement to create initial prototypes and every prototype undergoes iterative and progressive revisions. The Pebble smart watch is one notable example [17, 34] (Figure 2.1). But despite more intuitive technology, only those with technical expertise can spot an opportunity for innovation and subsequently prototype a solution.

Making for science

Scientists and researchers often need to use hardware to advance their findings. Low cost modular peripherals and intuitive microcontroller programming are allowing researchers and scientists to re-produce hardware at a lower cost, democratising access to usually expensive hardware [236]. Making can also be used to create new research devices, providing new perspectives on existing problems [151, 183, 269, 124].

Not all researchers and scientists however have a background in technology and despite more intuitive and lower cost prototyping tools, physical computing still requires input from technologists. Input from technologists can either be a positive force for change, or a negative force that constrains science to existing technologies. The naïvity and inexperience of technical novices can often yield new approaches to problems that may not have even been considered by technologists.

Making for artistic creativity

The Arduino Uno was created to enable artists to incorporate microcontroller programming and electronics into their creations [108]. As a testament to their creativity, some artists are creating *interactive* art installations through the application of physical computing. Figure 2.2

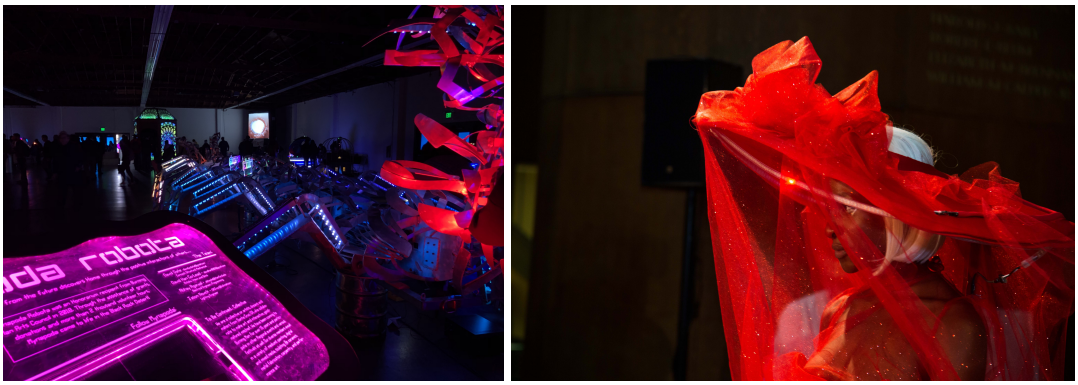

Fig. 2.2 Examples of applying physical computing to Art. An animatronic scorpion (left); and an interactive fashion garment (right).

(left) is one such example and shows an interactive animatronic scorpion composed of artistically hand-crafted metal, microcontrollers, sensors, and electronics. Lights are used to enhance the form of the scorpion in the dark and proximity to the scorpion triggers its animatronic stinger.

Fashion is another area which is seeing the application of physical computing to enhance creative processes. The *avant-garde runway* [9] allows fashion designers to express creativity, individuality, and innovation. Now fashion designers are attempting to incorporate physical computing into garments to bring interactivity to the runway. The end result are visually impressive, interactive garments that incorporate lights, sensors and actuators (Figure 2.2, right).

Fashion designers often have little knowledge of electronics or programming and they alone cannot produce interactive garments. Teams of technologists are required to help build garments and ground the ambitions of fashion designers. Special tools are also used to bridge the terminology gap between designers and technologists [257]. Once again, the presence of technologists in the creative process can either help or hinder.

Making to learn

Makers partake in making for the thrill of the creative process and the ongoing learning that occurs throughout [145]. To a maker no domain is out of reach and it is for this very reason that it is hard to predict the future applications of physical computing.

For example, Figure 2.3 (left) shows a school bus that has been converted into a fully functional recreational vehicle (RV) [27]. The conversion process involved the use of many skills including metalwork, woodwork, and mechanical engineering. Its creator did not stop

Fig. 2.3 Examples of maker projects created for learning: A DIY RV, converted from an old school bus (left); and a cardboard aeroplane (right).

there. Inside, a network of physical computing devices are used to dynamically control the interior cabin lighting.

Figure 2.3 (right) shows another unpredictable application of physical computing. Here, a delta wing cardboard-aeroplane has been constructed from household materials: lollipop sticks, sellotape, and glue. A physical computing device is integrated into the body of the craft to control motor speed and angle of the flaps. Another physical computing device—wirelessly networked to the one embedded in the aeroplane—remotely controls the position of the flaps and the speed of the motor.

These examples shows the true breadth of makers. Both applications of physical computing demonstrates the level of versatility needed from physical computing devices and supporting technologies. Both examples also show the lengths makers are willing to go to learn new skills. The creator of the RV was by no means an expert in woodworking or metalwork at the beginning of the project, but by the end they had honed the craft.

Analysis

The maker movement has made physical computing 'another tool in the toolbox' for citizens across many domains. However, physical computing technologies continue to be a source of complexity for some.

Citizen developers require intuitive technologies Designers, scientists, researchers, and many more not captured by this section, require assistance from technologists in order to engage with physical computing. Technologists are not always available to participate

and their sometimes fixed mindsets can stifle the creative process causing opportunities for innovation to be missed.

Citizen developers require flexibility The physical computing creations shown in this section also evidences a degree of unpredictability when citizens apply physical computing, and this unpredictability demands flexible operation. To support the creation of networked light switches and remote control aeroplanes for example, wireless networking protocols need to flexibly support both high and low data rate applications. And to better support the creation of interactive garments and iteratively prototyped hardware, wired protocols need to support flexible and dynamic physical composition.

Citizen developers require concurrent operation The physical computing creations shown are often performing more than one operation at a time. In the cardboard aeroplane example, a microcontroller controls servo angle and motor speed whilst receiving further commands from another device and in the animatronic scorpion example, a microcontroller controls lighting effects whilst performing proximity detection.

Citizen developers require energy efficiency The physical computing creations shown across this section are mostly embedded into materials and powered by battery: interactive garments, product prototypes, and cardboard aeroplanes. The use of batteries means that great care must be taken to efficiently consume energy to improve device longevity, giving users more 'time for awesome'.

2.1.2 The Internet of Things (IoT)

For most people, there is no need to think about how devices connect to the Internet in their daily lives. It is transparent—enter the correct credentials for a WiFi network and immediately a device can become part of the Internet. Over the past few decades however, the shape and utility of Internet connected devices has changed. No longer do devices take the form of traditional Personal Computers (PC), laptops, and smart phones. Instead, Internet connected devices are becoming *things* within our environment. In this phenomena—broadly referred to as the *Internet of Things (IoT)*—Internet connected devices are usurping the roles of traditional items (e.g. televisions, fridges, light bulbs), providing insightful behaviour-changing sensor data, and automating aspects of our daily lives. The IoT therefore presents a huge opportunity for innovation, and as a result, the number of Internet connected devices is expected to grow from 16 million to 156 million by 2024 [229].

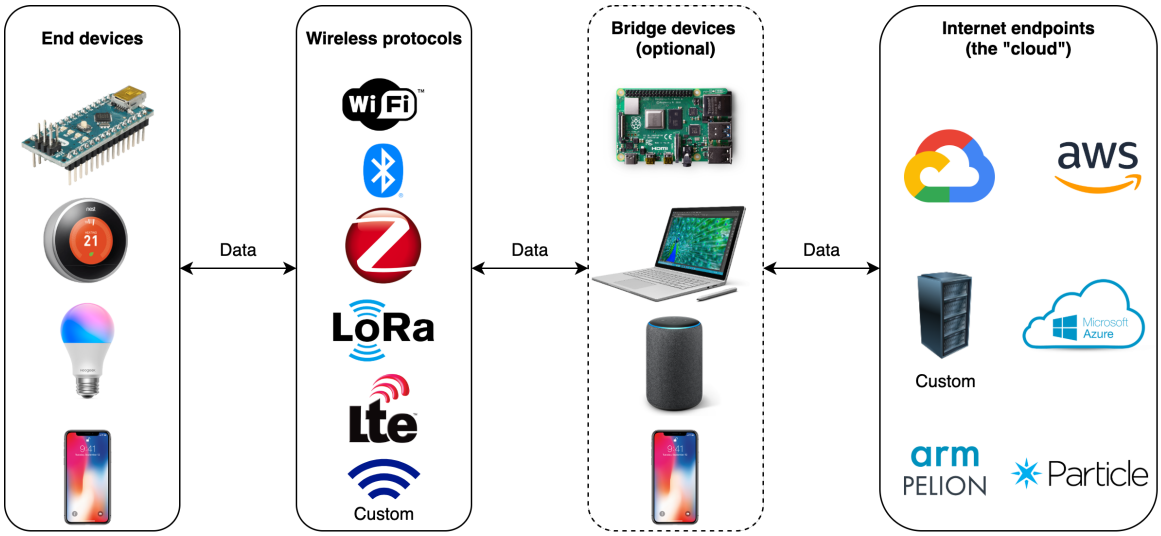

Fig. 2.4 Generalised architecture of the IoT.

Physical computing devices are becoming part of the IoT in the form of *end devices*. End devices are interactive devices that control, sense, or actuate environments and other devices. Using *wireless protocols*, end devices communicate with each other and to the Internet. Not every protocol can communicate with the Internet directly however, and an optional intermediate device called a *bridge* may be used to proxy Internet requests. *Internet endpoints* are the end destination of requests and can be hosted locally or remotely as part of the Internet. This general architecture is captured in Figure 2.4 and more detail is provided on each in the following subsections.

End devices

End devices are the physical interfaces between the real world and the virtual. They are Internet connected interactive devices that are placed in environments and are used to sense, control, or actuate environments and devices. In the IoT end devices usually take on one or more of the following roles:

- *Control*: End devices that change the behaviour of other smart devices take on the role of controllers. Smart phones and voice controlled Smart Assistants [4, 19, 38, 64] are examples of control end devices. Some controllers are reliably powered from mains sockets.
- *Report*: End devices that detect changes in an environment *report* the results to other end devices and Internet endpoints. Examples include smart switches, thermometers,

and presence detectors. Sensing data is usually processed by an external controller, which in turn activates an actuator. Sensors are usually embedded in environments and isolated from reliable power sources. As a result, sensors derive their power from batteries, the replacement rate of which depends on device energy consumption.

- *Actuate*: End devices that change something in the environment take on the role of actuators. Examples include Internet connected bulbs, power sockets, and thermostats. Actuation is usually caused by a controller based upon information from a sensor. Actuators are usually powered from a reliable mains source due to high energy consumption when actuating.

Driven by the maker movement, technologists are now building and prototyping their own Internet capable physical computing devices. Using such devices, technologists can build custom end devices that are a composition of the above roles. The technologist, however, has the complex job of creating the physical computing device, managing infrastructure, connectivity, and the data flow between devices.

Wireless protocols

Wireless protocols standardise communication between disparate devices. Each protocol is designed with specific constraints and application domains in mind, and it is the job the developer to navigate the complex design space and select the right protocol for the job.

All protocols need to balance responsiveness with power efficiency, especially so with battery powered end devices. Generally, the more responsive a protocol is, the greater the energy consumption. Increased radio activity is the source of increased energy consumption, and each protocol is designed with different trade offs between responsiveness and energy efficiency. For physical computing devices, responsive, soft real time, energy efficient networking is key to distributed interactivity.

Protocol range also impacts the applicability of a protocol. Protocols that work over short ranges, like Bluetooth [113], are not suitable for applications that need to work across distances greater than 30 metres. Long range protocols like LTE [256] or LoRA [117] are suitable for long range applications that span many kilometers. Ad-hoc networking protocols like Zigbee [187] or Z-Wave [159] offer some middle ground, supporting ranges up to hundreds of metres through extensible networking. A further of discussion of these protocols is provided later in this chapter.

Each protocol also has an associated monetary cost. Specialised hardware peripherals and software stacks to control them are required to enable wireless networking. Software stacks

demand more Random Access Memory (RAM) and flash memory and additional hardware peripherals require more silicon. Both factors add monetary cost to the manufacture of end devices, and consequently the price for the consumer.

Bridge devices

The IoT is a homogenous blur of wireless protocols due to the different requirements of end devices and applications. Regardless of wireless protocol, the ultimate goal is to communicate with the Internet and Internet connected devices communicate using the Internet Protocol (IP), a higher-level protocol that sits atop different wireless and physical transports. However, the design goals of IP are not well aligned to low power, low data rate wireless protocols and memory constrained microcontrollers. Consequently, many wireless protocols define high level, non-IP based protocols that are better suited to protocol and device constraints.

Bridge devices proxy packets received from non-IP to IP-based networks and receive packets from one wireless protocol and transmit them using another. In some cases, wireless protocols may be directly translated to wired protocols like Ethernet [218]. Even though many bridge devices are standalone devices, it is possible to turn most devices that can operate two or more wireless protocols into a bridge device. An example of this model in action is the modern smart phone and smart watch ecosystem where a smart phone 'bridges' data requests from a smart watch to the Internet.

Internet endpoints

Internet endpoints act as a centralised point of communication for IoT end devices. They can exist remotely as part of the Internet or Local Area Networks (LANs). Internet endpoints are primarily used by end devices to store and retrieve data but they also provide a centralised means for users to control and actuate end devices using smart phone and web applications.

A well thought out Internet endpoint presents REpresentational State Transfer (REST) [150] APIs. REST-ful interfaces support five well-defined and stateless operations: POST, GET, PUT, DELETE, PATCH. These five operations encapsulate every possible machine-to-machine interaction and they form the basis of nearly every Internet request. Their usage is especially prevalent in the IoT and developers use REST to neatly abstract data storage and retrieval interfaces.

Well-defined Internet endpoints can be easily combined with data flow management tools like If This Then That (IFTTT) [21] (Figure 2.5) and Microsoft Flow [45]. Data flow management tools let users define automated control flows for Internet endpoints, and more

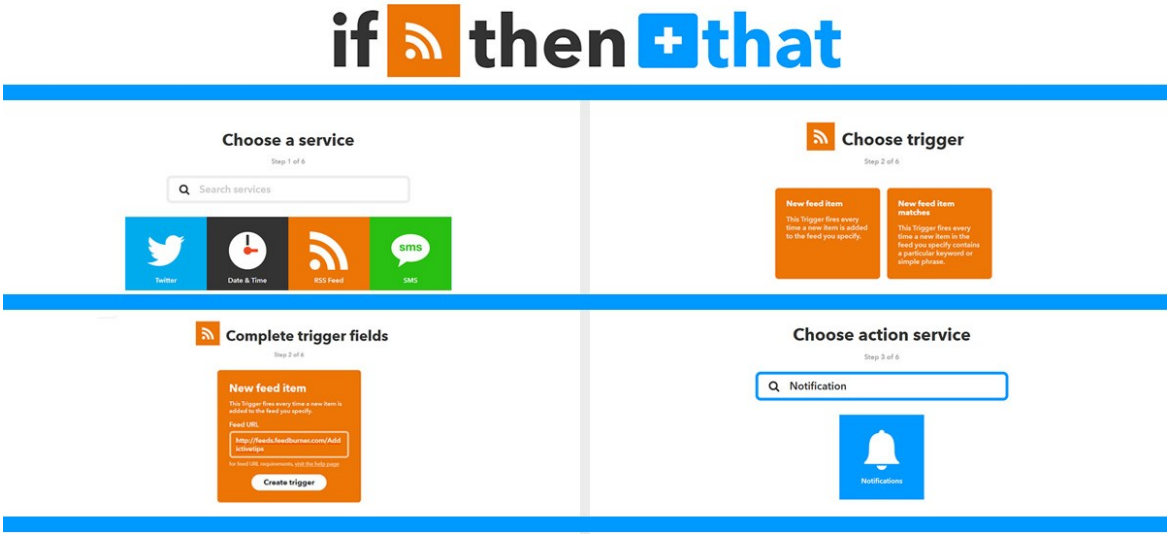

Fig. 2.5 The If This Then That (IFTTT) recipe builder. Users can connect events from one Internet endpoint to control another Internet endpoint [65].

recently, for IoT devices. Social media messages and arbitrary endpoint events can now be used to control and actuate end devices, allowing users to control a light bulb from a tweet.

IoT for consumers

Consumer IoT seeks to automate the mundane aspects of every day life, like switching on a light bulb, turning on the heating, or keeping track of items in refrigerators. There are now companies that specialise in producing pre-packaged IoT solutions that work out of the box with minimal configuration—an important factor for consumers that may have little technical expertise.

Consumer IoT solutions generally follow the architecture described earlier. Most solutions are distributed with a dedicated bridge device that wirelessly connects end devices to Internet endpoints. The advent of smart assistants however, is increasing the utility of bridge devices. Now many bridge devices double as a point for voice controlled interactivity.

The Amazon Alexa [4] ecosystem (Figure 2.6) is perhaps the most widely adopted consumer IoT platform to date. The centre piece of the ecosystem is the Alexa smart assistant, a physical interactive device that can respond to voice commands and interact with Alexa enabled devices. A plethora of peripherals can be connected to Alexa, including Philips Hue light bulbs [77], allowing users to control lighting using their voice. Alexa hubs double as bridges for connecting end devices to the Internet, and their low cost makes extending IoT infrastructure intuitive.

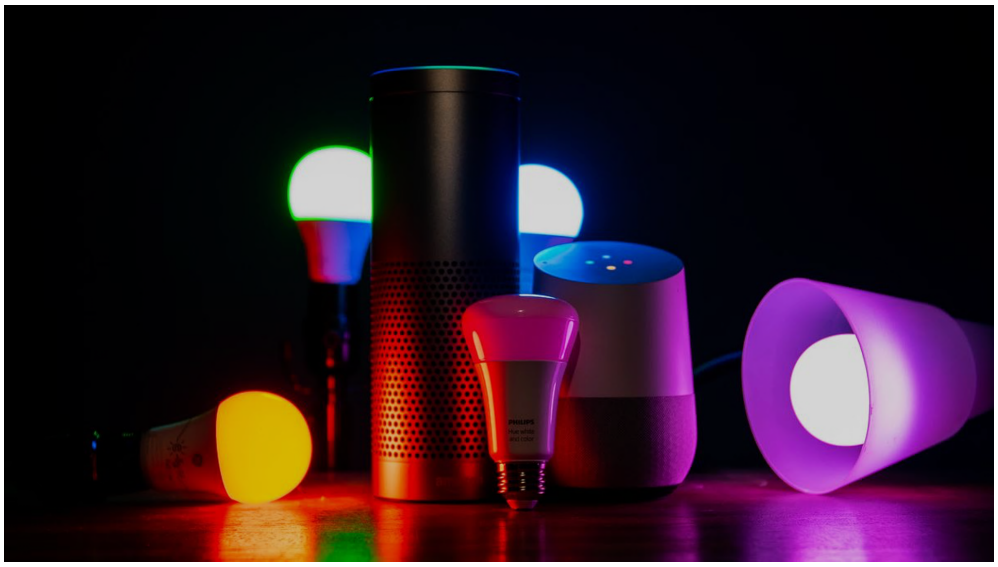

Fig. 2.6 A selection of consumer oriented IoT devices including Amazon Alexa and Philips Hue light bulbs [66].

Apple HomeKit [64] is an ecosystem of IoT end devices that are heavily integrated with the iOS ecosystem (e.g. iPhone and iPad). A static central hub is not required if an iOS device is connected to the same network as a HomeKit device—an iPhone or iPad dynamically takes on the role of a bridge device. An iOS app installed by default on every iPhone and iPad is used to control end devices. Communication with HomeKit devices happens locally and is not processed by an external server and Apples' voice assistant, Siri, can also be used to control and actuate end devices without additional configuration. Remote control over HomeKit devices requires an iOS device (e.g. an Apple TV, iPad, or HomePod) is connected to the network and signed into the same Apple ID.

Many consumer IoT solutions also provide simple programming environments to automate interactions between devices. The Alexa ecosystem allows users to define routines allowing users to associate actions with custom phrases and events. Routines are defined using a condition-based paradigm to define IoT interactions, similar to that of IFTTT.

Apple HomeKit users can compose automations in the Shortcuts application (Figure 2.7). The deep integration between iOS and HomeKit allows users to create deeper more personal interactions between devices. For example, the Shortcut shown in Figure 2.7 sends a text message and turns the heating on when its creator leaves work.

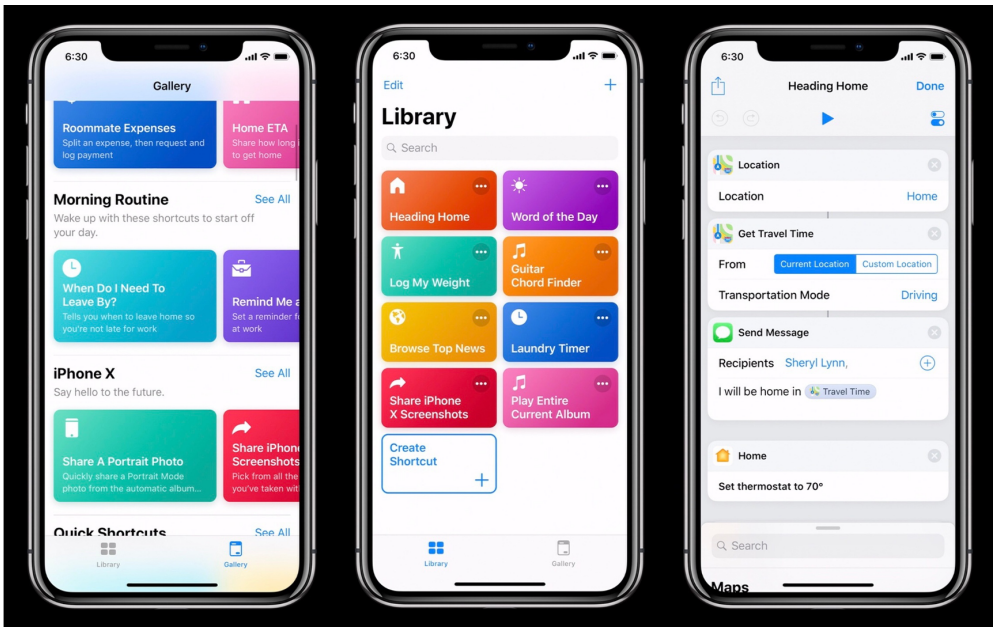

Fig. 2.7 The Apple Shortcuts application that can be used to automate interactions between iOS and HomeKit devices [48].

IoT for technologists

An emerging area of the IoT is *Do It Yourself (DIY) IoT* [261]. DIY IoT encourages technologists to build their own IoT solutions, empowering the technically capably to use IoT to solve real world problems. Here, technologists build and prototype physical computing devices, connect them to bridge devices (if required), and onto the Internet.

Figure 2.8 (left) shows an Internet connected plant monitoring system [35]. An embedded development board is wired to forks which are placed in soil to detect the moisture level. A separate light sensor keeps track of how much light the plant is getting. Both statistics are reported to an Internet endpoint using built in WiFi support giving horticulturists an approximation of when a plant requires more water or more light. Though the form factor of the device is large at present, the presence of a USB battery pack suggests that eventually the device will decrease in size for easy deployment. Technologists can build more than one to monitor the health of many plants.

Figure 2.8 (right) shows a Twitter controlled lighting system. Inside, an embedded development board is connected to three neopixels and is wirelessly networked to an Internet endpoint using open source CheerLights firmware [55]. The colour of the lights is automatically changed through tweets following the form '#cheerlights <colour>'. As can be seen

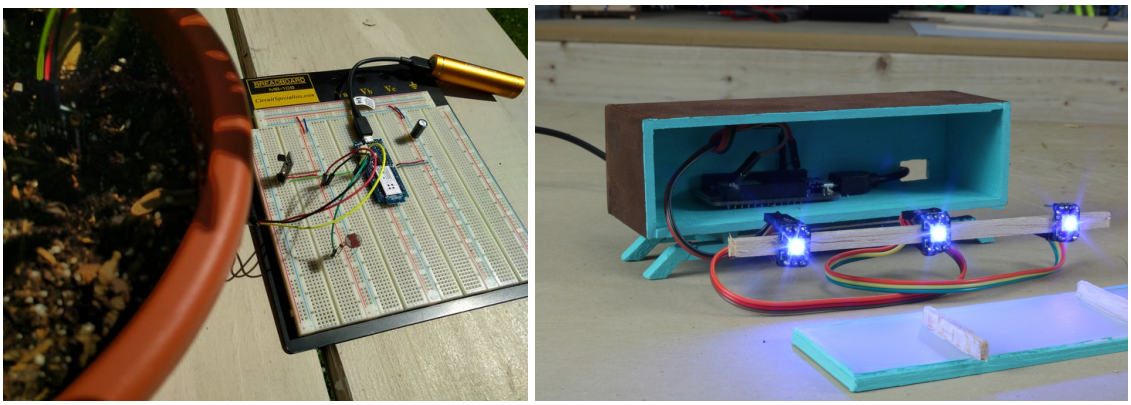

Fig. 2.8 DIY IoT end devices. A remote plant monitoring system [79] (left); and a Twitter controlled lighting system [55] (right).

from the image, prototypes can near the quality of finished products and technologists may place more than one around their home for decoration.

More technical tools for customising the operation and interactions between IoT devices are available to technologists. Node-RED [230] enables the easy piping of data using a graph-based programming model. Each node within a graph takes data input and has a resulting output (Figure 2.9, left). Connecting nodes together forms a data flow whose output can be injected into Internet endpoints or end devices. Node-RED requires NodeJS [273] to operate and is primarily designed to run on the Raspberry Pi. Because of its JavaScript underpinnings however, applications can also be deployed to commercial infrastructure services like Amazon Web Services (AWS) and Microsoft Azure. Common nodes for interacting with popular web services (like Twitter) and databases allow users to build usually complex applications with ease.

Mozilla WebThings [223] provides bridge firmware for routers and Raspberry Pi devices. Once configured, users can control connected end devices around the home via the bridge. End devices appear in the WebThings portal (hosted by the bridge) and present themselves as a specific type of device (e.g. a smart bulb). End devices also present properties that can be tweaked from the portal interface (Figure 2.9, right). The WebThings platform is also compatible with many consumer accessories and goes some way towards simplifying the complex infrastructure and connectivity problems of the IoT.

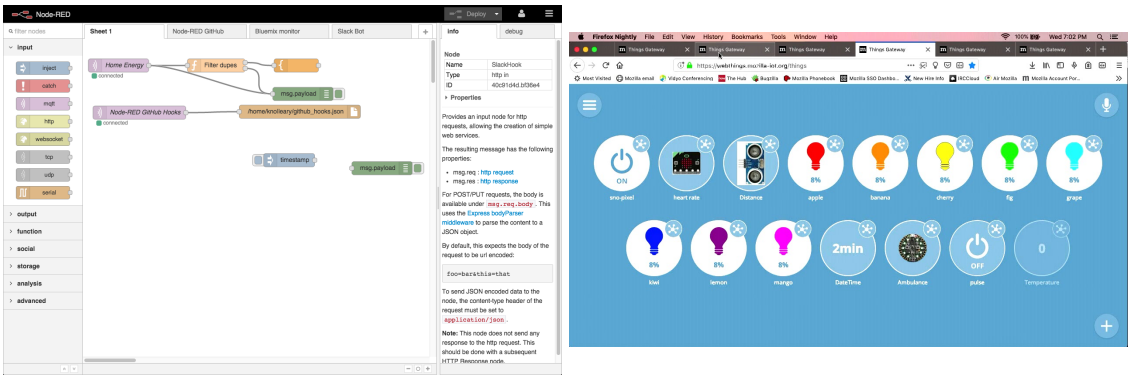

Fig. 2.9 The Node-RED flow creator [74] (left); and the Mozilla Web Things dashboard [85] (right).

IoT for research

The IoT is also a proven vessel for science and research. Bates et al. used the IoT to monitor the energy consumption of an entire university campus [110]. Obtaining fine-grained energy data allowed facility staff to reduce the base energy consumption of buildings.

Fine-grained data and distributed sensing has also been used to measure soil quality. Insights gleaned the use of IoT revealed how soil quality is changing over time, allowing farmers to better optimise their crop placement [277].

The use of IoT as a tool for research currently involves technologists. But it is important that more people are given the opportunity to independently create and innovate using the IoT. Democratisation of its use will reveal new applications and uses that have not yet been realised.

IoT for businesses

The IoT has proven itself to be a valuable tool for businesses and is used for enhanced logistics and tracking, better work force utilisation, and improved analysis of consumer behaviour. There are now a number of companies aiming to give businesses the tools and infrastructure to build their own IoT solutions: IBM with IBM Watson IoT [178], Amazon with AWS IoT [2], and Microsoft with Azure IoT [219]. But the focus of these platforms is to ease the creation of Internet infrastructure, rather than end devices.

There are companies that seek to provide a complete IoT solution—including tools to help with the creation of end devices. Particle [32] caters to the generic applications of the IoT, but also provides tools to specialise and create custom end devices. Particle includes a suite

Fig. 2.10 An Internet connected smart prosthetic supported by Particle [80].

of embedded development boards, a custom online programming environment, and a secure infrastructure to support applications. Devices can be managed from a centralised portal which even supports dynamic firmware updates from the only programming environment. Because creating end devices is challenging, Particle also provides access to IoT experts that can take an end device from prototype to market.

Using the Particle ecosystem, many companies have built custom end devices for the IoT. Unlimited Tomorrow is one such start up that seeks to create smart prosthetics for users, controlled by electrical waves emitted by the nervous system (Figure 2.10). Each prosthetic is fitted with a microcontroller that actuates prosthetics when certain electrical wave patterns are recognised. Tuning electrical waves to each user is a process that requires continual customisation. Microcontrollers are therefore also networked to the Internet where electrical waves are used as inputs to machine learning models to tune microcontroller operation. This application expresses the true potential of the IoT and further democratisation of physical computing will expose many more opportunities for its application.

Analysis

In the many examples above, there is no evidence that those with little technical expertise can create end devices for the IoT. In fact, some businesses, like Particle, even profit off of the fact that end device creation is steeped in technical complexity. Creating an end device for the IoT should be as easy as consumer-oriented IoT makes home IoT, but the process of

building and connecting a physical computing device to the Internet remains a challenging task.

Citizen developers require simple ad-hoc networking When considering the examples above, there is a clear need for dynamic networking. Technologists tend to build end devices using intuitive protocols like WiFi, constraining projects to within range of the nearest access point. Consumer IoT solutions already use ad-hoc networking protocols to make adding new devices to ecosystems easy, dynamic, and scalable. For technologists, however, the use of ad-hoc networking protocols comes with additional complexity and even particle—who specialise in creating commercial IoT solutions—recently discontinued their ad-hoc networking solution citing its complexity [33].

Citizen developers require minimal configuration Many of the examples above seek to simplify the configuration of IoT applications. Consumer applications especially try to make installing IoT devices in the home as easy as turning on the device itself. Even technologists seek to minimise configuration, choosing to use ubiquitous, but more power consuming protocols like WiFi, over more power efficient and suitable protocols like Zigbee.

Citizen developers require minimal infrastructure Across the examples above there is also trend of simplifying or minimising infrastructure. Ecosystems aimed at consumers, like Alexa and Apple HomeKit, make the addition of infrastructure simple, dynamic, and compelling, as hubs double as both points for interaction and connectivity. Technologists also choose to use a ubiquitous protocol, like WiFi, that are immediately compatible with their home network over other protocols that require additional infrastructure, like Zigbee.

Citizen developers require simple abstractions In some cases—especially in consumer-oriented solutions—users are able to create simple applications to personalise the functionality of their IoT devices. Consumers can use tools like Shortcuts and Alexa routines to customise the operation of their end devices. Application creation is facilitated using simple abstractions that require no programming experience. Technologists also use environments, like Node-RED and WebThings, to customise and abstract away many of the complexities of the IoT. These environments support greater customisability than consumer-oriented abstractions through the incorporation of programming.

Citizen developers require energy efficiency Many of the examples discussed in this section operate on battery power and energy efficiency is therefore important when considering physical computing technologies for building devices for the IoT.

2.1.3 Education

With the modern job market increasingly demanding *digital skills* [155, 140], imbuing future generations with technical proficiency is more important than ever. Recognising this, the UK government in 2014 made computer science a core subject in the national education curriculum [164] and educators are now expected to teach the subject to students. Educators, however, typically lack the technical expertise and confidence to teach computer science [116, 115, 114]. They therefore require intuitive computer science education tools that have a low barrier to entry.

Physical computing offers the most promising approach for an engaging and intuitive educational experience. It was first applied to education by Papert [233, 232, 189] who proposed the concept of constructionist learning—using computers as a building tool to realise ideas, learning in the process. Papert exemplified the constructionist concept using Logo to move real-world programmable robotic turtles [232, 216].

Physical computing technologies have evolved since the seminal work of Papert in 1980, and they now allow students to build custom input devices for personal computers, construct physical computing devices from pre-existing and easy-to-connect modules, and even build entirely new physical computing devices from reusable materials. The upcoming sections categorise the different usages of physical computing across education broadly following the work of Hodges et al. [177].

Peripheral devices for personal computers

Some physical computing technologies blend the real-world with the digital by acting as inputs to personal computers. Makey Makey [134] distributes an embedded board that exposes a number of capacitive GPIO (Figure 2.11, left). The embedded board is pre-programmed to act as a keyboard peripheral, and key presses are triggered based on the state of the GPIO. Users can build custom physical user interfaces from materials like fruit and plasticine to trigger key presses.

Scratch [244]—a visual programming environment for 2D game creation—has recently been developed to extend beyond the screen, moving into the physical world using programmable physical computing devices (Figure 2.11, right). In a similar way to Makey

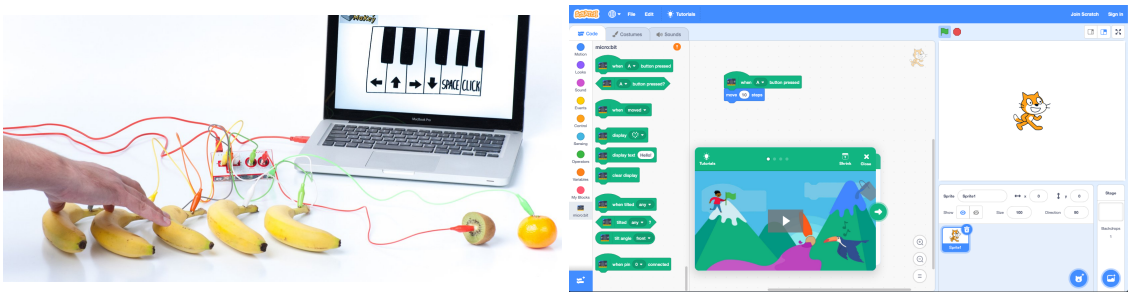

Fig. 2.11 Physical computing solutions that act as peripherals for personal computers. Makey Makey [71] (left); Scratch augmented with the micro:bit extension (right).

Makey, pre-compiled binaries for physical computing devices cause them to appear as USB or BLE devices and act as inputs to Scratch programs. Scratch connects to physical computing devices in tethered mode communicating using wired protocols like Firmata [267] or wireless protocols like Bluetooth. As programs execute, scratch issues commands and listens for events issued by the physical computing device. Device events can trigger in-game events, unifying the real world with the virtual.

Kodu [208] gives users a 3D world to explore. Using a custom visual programming language, users can program their hero to target enemies and move through the 3D environment. Like Scratch, Kodu also lets users connect physical computing devices in a tethered capacity. Interactive devices can act as inputs to the 3D world, translating real world interactions into the virtual world.

Tangible programming kits

Tangible programming solutions enable users to build custom physical computing devices using pre-fabricated, easy-to-compose hardware modules. For example, littleBits [111] (Figure 2.12, left), allows hardware modules to be easily connected using polarised magnets. Modules have specific functions, including: input modules, such as buttons, switches, sensors; output modules, such as lights, displays, buzzers; branch modules, for extending signals; and power modules for providing electricity to the circuit. Modules are combined by the user to create simple circuits that interact with the environment.

Cubelets [28] also uses magnets to connect cube-shaped modules together. Cubelets can be stacked on top of one another to build simple interactive systems. Like littleBits modules, each cube has a specific function. Cube types include motors, light sensors, power cubes, and potentiometers. Correct connection orientation is guaranteed by polarised magnets. Cubes

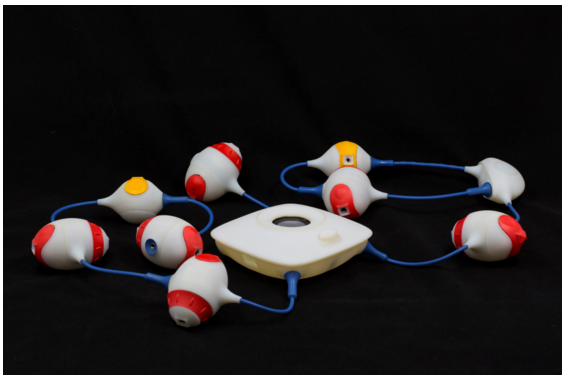

Fig. 2.12 Examples of tangible programming technologies: littleBits [70] (left); Torino [221] (right).

can be combined in an assortment of ways to create devices that respond to a variety of sensor inputs.

Tangible programming is also ideal for widening participation. Torino [221] is designed for the visually impaired (Figure 2.12, right). Torino hardware modules each have a specific texture to convey meaning, and each texture maps to a programming concept, like loops, logic, and conditionals. Hardware modules are combined using a simple 3.5mm audio jack, but the end result is not an interactive system, it is a program that creates sound on a personal computer. As Torino modules are connected, the sound produced by the program reflects the physical composition of the modules, giving partially sighted users an engaging and inclusive learning experience.

Programmable 'turtles'

Since Papert introduced the concept, many have emulated the idea of programmatically moving a real-world turtle. The Sphero Mini [265] (Figure 2.13, left) is one of the most well-known modern robotic turtles. Users program its movements via a smart phone application and can be drawn, or programmed in Scratch-style blocks, and JavaScript. When completed, movements are transferred to the Sphero Mini via Bluetooth for execution. Smarter versions of the device augment the turtle with additional sensors, giving the turtle—and its users—the power to respond to real world stimuli.

Cubetto [13] (Figure 2.13, right) takes a slightly different approach. It provides a physical programming palette in which users plug in different jigsaw pieces (instructions) to cause

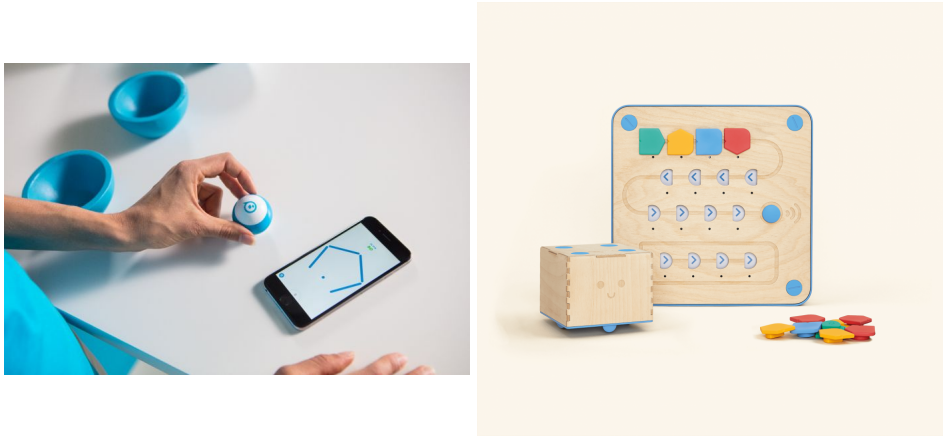

Fig. 2.13 Examples of programmable turtles: Sphero [84] (left); Cubetto [57] (right).

a small robot (Cubetto) to navigate the environment. Jigsaw pieces are plugged into the programming palette and commands include forward, back, left, and right. Commands follow a sequential execution model and are transferred to Cubetto for execution.

Cue [291] is a more advanced robotic turtle. It has a variety of in-built sensors and can produce and recognise speech. Applications for Cue are developed using a visual programming language and are once again transferred to the device via Bluetooth for execution.

Programmable construction sets

Programmable construction sets combine programming, hardware modules, and reusable construction materials to build physical computing devices. One of the most well-known examples of this style of physical computing tool is Lego Mindstorms. Lego Mindstorms is derived from Paperts' famous Mindstorms paper and integrates the familiar plastic building blocks with programming and electronics. The set consists of a reprogrammable main "brick" that runs an application created in a visual programming environment. Completed programs written in the visual program environment are transferred to the main brick via infrared or USB cable. Brick-based modules, featuring sensors and actuators, can be connected to the main brick and used by applications to create a physical computing device. The main brick communicates with the sensors via electrical contacts that take the place of the usual plastic connectors. The Lego Mindstorms programming environment was the first to offer a visual programming paradigm and Scratch is heavily influenced by the fundamental concepts introduced by Lego Mindstorms.

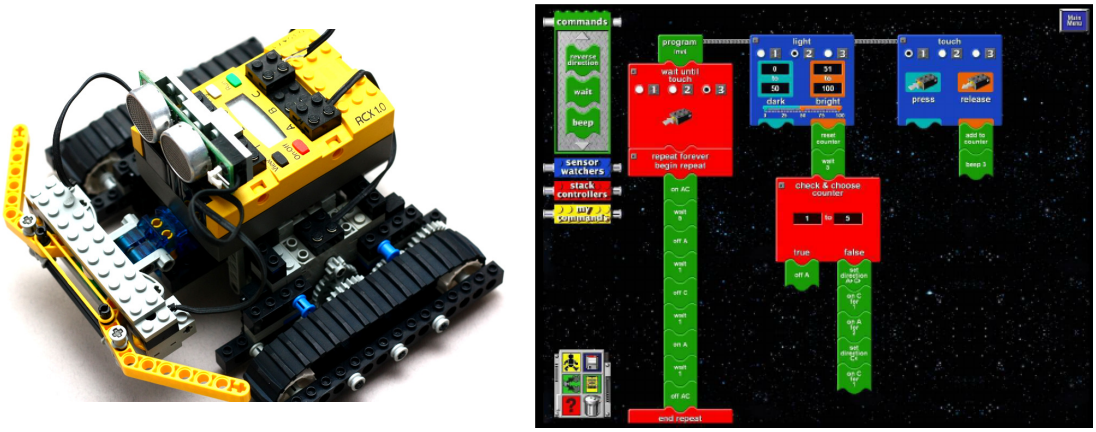

Fig. 2.14 The Lego Mindstorms RCX "brick" [68] (left) and visual programming environment [67] (right).

Another programmable construction kit, the VEX V5 [247], takes a similar approach. The V5 ecosystem features a main brain and a number of peripheral modules that can be used to build devices. Instead of lego bricks, V5 uses grated metal and nuts and bolts to build more rugged robots. The V5 ecosystem allows for text-based programming using C++ and visual programming using Scratch. Programs are transferred via cable to the brain.

Analysis

The physical computing technologies discussed in this section provide an engaging way for students to learn computer science concepts. Perhaps, more importantly, they give educators a low-barrier, friction-free way to teach computer science concepts. Both teachers and students typically lack technical expertise and the adoption of physical computing in education is evidence that more intuitive design increases participation from those with little technical expertise.

Citizen developers require intuitive technologies There is a distinct absence of unconstrained hardware prototyping in the physical computing technologies discussed above. Their absence implies that in their current form, prototyping technologies for freely building physical computing devices are outside the reach of students and teachers.

Citizen developers require simple abstractions Across the physical computing technologies discussed in this section, abstractions are a reoccurring theme. Tangible programming kits abstract hardware as distinct building blocks, programmable turtles abstract movement

commands as flow-based sequences, and programmable construction kits use reusable materials and visual programming languages to scaffold the construction of physical computing devices. These abstractions are simple enough for use by technically inexperienced students and educators.

Citizen developers require simple composition Whenever hardware composition is involved, the physical computing solutions in this section seek to simplify composition. Personal computing peripherals use crocodile clips, tangible programming kits use polarised magnets and 3.5 mm jacks, and programmable construction sets use easy to compose materials like lego to make hardware composition simpler. These simpler composition mechanisms allow technically inexperienced teachers and students to participate in physical computing.

2.2 Embedded development boards

Embedded development boards are central to enabling citizens to build physical computing devices. They are typically consist of PCB, a single microcontroller, and electrical contacts that expose microcontroller GPIO for prototyping.

There are many embedded development boards, but they broadly fit into three different categories. *Prototyping* boards (Section 2.2.1) are designed to make it easy to connect to external circuits and peripherals through dedicated GPIO connectors; *networking* boards (Section 2.2.2) are prototyping boards with built-in wireless networking capabilities; and *integrated* boards (Section 2.2.3) are prototyping boards that have on-board sensors and optionally built-in wireless networking. Integrated boards are able to function as physical computing devices without any further prototyping. The upcoming sections discuss each type of board in the context of making, the IoT, and education.

2.2.1 Prototyping boards

Though all embedded development boards can be conceivably used to build physical computing devices, *prototyping boards* offer an intuitive and reasonably low cost starting point. Prototyping boards typically have no on-board sensors and instead provide great customisability by using wires to interface with external sensors and electronics. This flexibility allows users to more easily build custom physical computing devices.

The Arduino Uno (Figure 2.15) is one of the most iconic prototyping boards, seeing heavy adoption by the maker community [107]. The Uno is relatively low cost (around £30

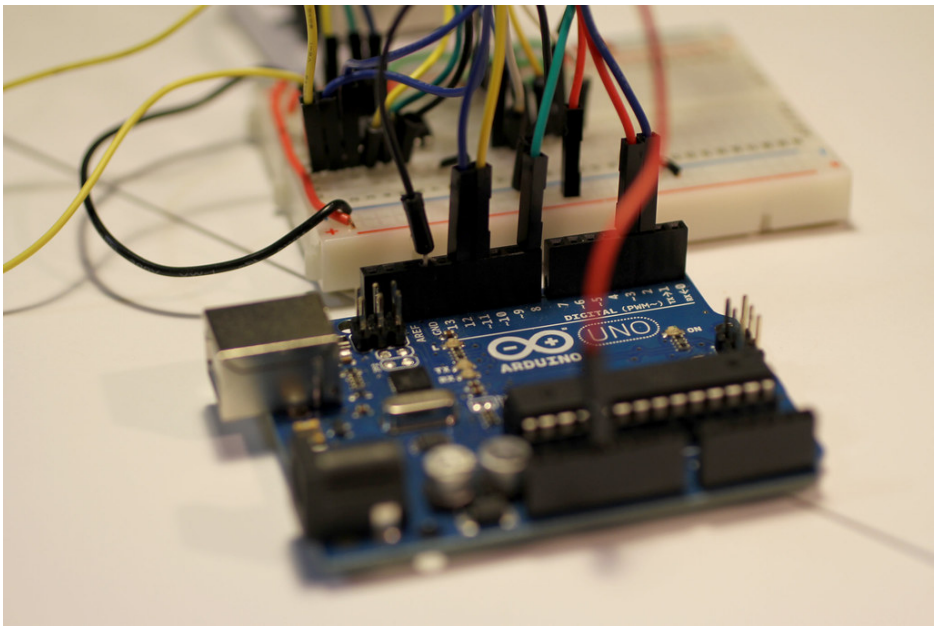

Fig. 2.15 Prototyping a physical computing device using an Arduino Uno [50].

at release) and features an 8-bit Atmel ATMega328p reprogrammable microcontroller and standardised PCB mounted GPIO headers. GPIO headers expose electrical microcontroller contacts for flexible hardware composition, letting users interface with external circuits and sensors programmatically. The microcontroller itself has just 2 kB of RAM and 32 kB of flash, requiring applications to be written in low level programming languages like C/C++ for efficiency. The accompanying Arduino IDE seeks to simplify C/C++ programming through a common set of simple APIs (originally designed for artists [108]). The IDE also builds programs and transfers binaries to Arduino devices using serial-over-USB.

Since its launch in 2003, there are now many Arduino boards in existence. Over time, Arduino devices have moved from less capable 8-bit microcontrollers to more capable 32-bit cortex M0 microcontrollers. This new class of microcontroller brings more memory, integrated peripherals (such as wireless connectivity), and faster processor speeds making the Uno obsolete. Throughout this revolution however, the Arduino has maintained its form factor, standardised pin out, and low cost.

Feather boards, produced by Adafruit [18], offer a smaller form factor and lower price point (around £20) than Arduino boards (Figure 2.16, left). Their smaller form factor gives users greater versatility when building physical computing devices. Feather boards can be programmed using the Arduino IDE and wired directly to peripheral sensors. A number of compatible Wings can also be mounted on top of Feather boards to add additional sensing capabilities and wireless connectivity. Feathers and Wings share a different standardised pin

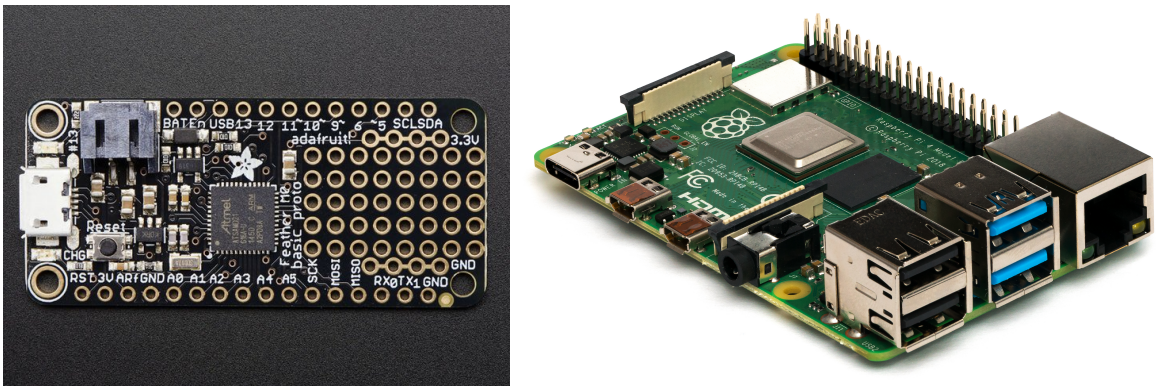

Fig. 2.16 Other embedded development boards for prototyping. A Feather form factor board (left) and a Raspberry Pi (right)

out to Arduino. Through their open source and collaborative approach to learning materials, software, and hardware, Adafruit has become a pillar of the maker community.

Another iconic device of the maker movement is the Raspberry Pi [279] (Figure 2.16, right). Launched in 2012, the Raspberry Pi was intended as a device for education, but it instead gained utility as a low cost standalone computing device. The Raspberry Pi is in a different class from Arduino and Feather boards and runs a full linux-based operating system with a Graphical User Interface (GUI). Wireless connectivity was added to later revisions of the board making it easy to connect the Pi to the Internet.

The Pi is also an ideal board for prototyping physical computing devices. Via GPIO headers incorporated into the PCB, users can prototype in a similar way to the Uno. Headers allow communication with sensors and the direct manipulation of both digital and analogue signals from intuitive programming languages like Python [281]. GPIO headers are also standardised but observe a different standard to the Arduino and Feather ecosystems.

2.2.2 Networking boards

Thanks to Moore's law [251], many development boards now come with wireless networking built in. Wireless capability is either supported by the on-board microcontroller directly or by a separate PCB-mounted co-processor. Because of their direct support for popular wireless networking protocols, networking boards are used by technologists and businesses to prototype and build devices for the Internet of Things (IoT).

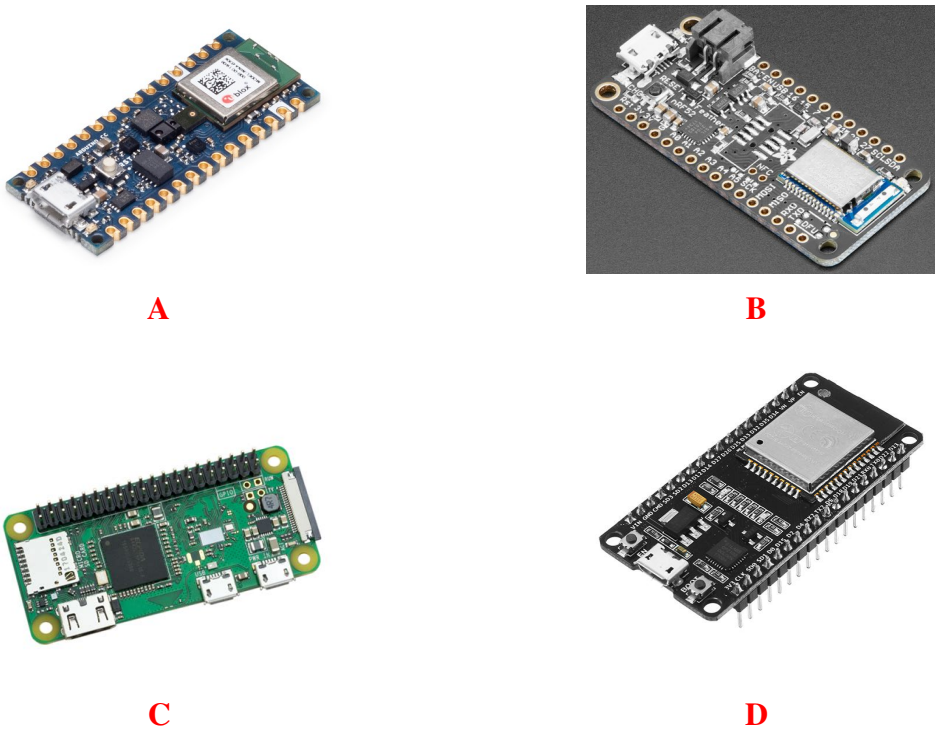

Fig. 2.17 Arduino Nano BLE Sense [49] (A); Adafruit Bluefruit [46] (B); Raspberry Pi Zero WiFi [81] (C); ESP32 development board [59] (D).

For technologists

Many networking boards are simply prototyping boards with added support for wireless networking. Before networking boards, wireless networking peripherals were connected to prototyping boards in very much the same way as sensors. Integrated networking has significantly reduced the complexity of building an IoT device.

The need for integrated networking emerged across many ecosystems simultaneously and there are now many networking boards to choose from. The Arduino ecosystem now offers smaller form factor Arduino Nano boards, like the Arduino BLE sense (Figure 2.17A). Adafruit now produce Feather boards with integrated wireless networking like the BlueFruit (Figure 2.17B). Standard Raspberry Pi models now also include built in support for BLE and WiFi, and the Raspberry Pi foundation recently produced a smaller, low power version of the Raspberry Pi, the Raspberry Pi Zero W (Figure 2.17C), for simpler IoT prototyping. These boards cost between 20 and 30 pounds.

The need for integrated networking has also prompted the creation of new development boards. The ESP32 development board (Figure 2.17D) is one of the most popular networking boards used to build IoT devices. The ESP32 has built-in support for both BLE and WiFi, and

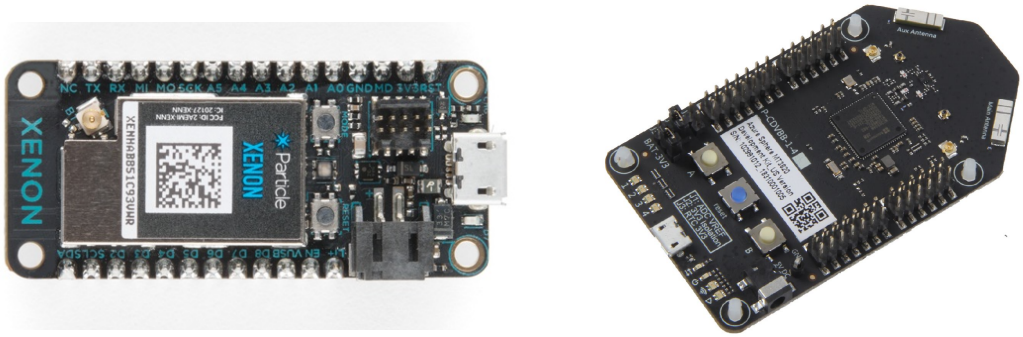

Fig. 2.18 Particle Xenon [75] (left), Azure Sphere [54] (right).

a powerful processor that can support a variety of higher level programming languages with ease. These features, in addition to its low cost of around £15, make the ESP32 a compelling choice for many IoT projects.

For businesses

The IoT presents businesses with many opportunities for innovation. Building an IoT solution however, is steeped in technical complexity, especially when part of the solution involves the creation of a custom IoT device. Added business concerns around security make the process even more challenging. There are now companies that specialise in providing IoT solutions that scaffold businesses in building IoT devices and connecting them to the Internet.

Particle [32] provides infrastructure and four networking boards to create custom IoT devices. All four boards observe the feather form factor and pin out specification [18] and each board contains two microcontrollers, one for application execution and another for wireless connectivity. The Photon board supports WiFi; the Electron board incorporates cellular technology for longer range applications; the Argon board features WiFi and BLE for connecting Bluetooth devices to the Internet; and the Boron board supports LTE for long range IoT applications. Particle boards can be connected together to adhere to different application requirements. Interestingly, Particle recently discontinued its embedded development board designed for ad-hoc wireless networking, Xenon (Figure 2.18, left), citing configuration complexity [33].

Microsoft recently announced Azure Sphere [268], a security centred solution for IoT devices. The Azure Sphere development board (Figure 2.18, right) features three processors, one with heavily secured processor with built in tamper detection, and two less secure

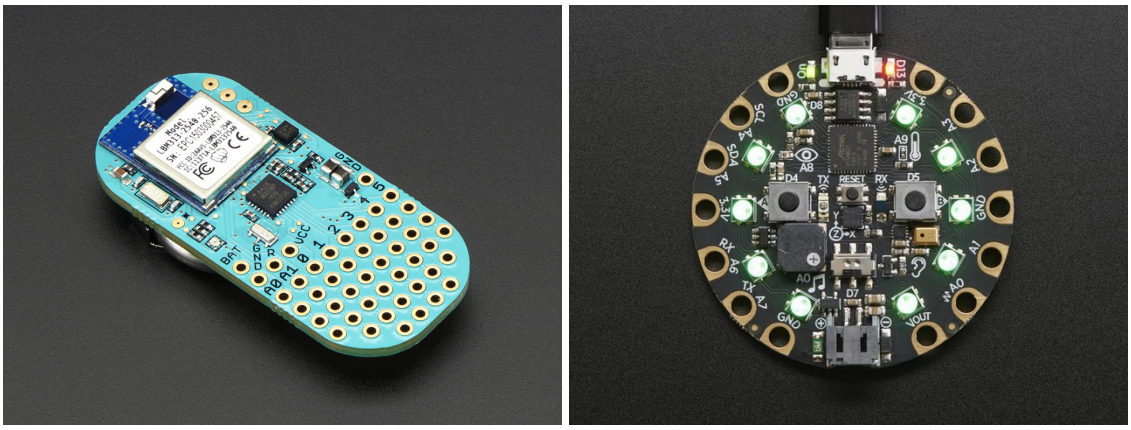

Fig. 2.19 The Light Blue Bean [69] (left) and Circuit Playground Express [56] (right).

processors for non-critical co-processing. Sphere directly connects to Microsoft Azure infrastructure, providing businesses with a completely secure IoT solution. Notably Starbucks is now using Sphere to remotely monitor coffee machines [39].

2.2.3 Integrated boards

Integrated development boards come with both sensors and wireless networking built in. Many integrated boards can therefore act as physical computing devices without any further prototyping. This has proven to provide a more intuitive physical computing experience and many integrated boards are now used as tools for education [116].

The Light Blue Bean (Figure 2.19, left) is an Arduino compatible integrated board that costs around £30. It uses the same processor as the Arduino Uno (the ATMega328p) but adds a Bluetooth co-processor, an accelerometer, temperature sensor and a single RGB LED. A small prototyping area allows users to expand the capabilities of the bean by wiring it to external circuits and peripherals. Like any other Arduino device, the Bean is programmed using the Arduino IDE. It can also be connected to mobile applications running on iOS and Android devices via Bluetooth for the tethered execution of user commands.

The Circuit Playground Express (CPX) is another Arduino compatible integrated development board. It follows the circular form factor of the extremely popular LilyPad Arduino—a prototyping board for creating interactive fashion garments—where the back of the board contains no components and GPIO are brought right to the edge of the circular PCB. This makes it easy to combine the board with garments using conductive thread or crocodile clips. The CPX offers a huge amount functionality for the low price of £20 and features

many on-board components including 10 RGB LEDs, a microphone, a speaker, an infrared transmitter and receiver, an accelerometer, a temperature sensor, and 3 buttons (Figure 2.19, left). Programs can be written in the Arduino IDE, in Microsoft MakeCode [142], or using a custom code editor and CircuitPython [11]. Language diversity provides great flexibility for educators and makers alike. Program binaries are transferred to the device using the Arduino IDE or alternatively, using a simple USB mass storage abstraction. When connected to a personal computer, the CPX appears as a flash drive and program binaries can be transferred using a simple file copy operation.

The BBC micro:bit (Figure 2.20) is another example of an integrated development board. The micro:bit is the product of the British Broadcasting Coorporation (BBC) Make It Digital Initiative, a concerted effort by the BBC and 29 project partners [102] to encourage a new era of creativity in the young using programming and digital technology as its medium. Simultaneously, the initiative would also support the UK's mandate to teach computer science concepts at all grade levels [237]. The microbit was deployed to approximately 800,000 UK Year 7 (11/12 year old) school children in 2015-2016.

The BBC and its partners developed the micro:bit as an engaging, inexpensive (£14), powerful, and easy-to-use learning tool. As such, the micro:bit can be programmed from any web browser via Microsoft MakeCode and MicroPython [161]. The device is also playful and engaging, the size of a credit card, can be powered from battery, and has built in support for BLE, a 5x5 LED matrix display, an accelerometer, and a magnetometer. The edge connector—a slot-based GPIO connection interface along the bottom of the board—extends the micro:bits functionality, allowing users to connect to electrical circuits and sensors using banana plugs and wires. Program binaries are transferred to this microcontroller using a simple file copy operation to a USB flash drive (similar to the CPX). The micro:bit supports all these features using a microcontroller with only 16 kB of RAM and 256 kB flash. Its easy programability, embedability and feature set enables a plethora of lessons without requiring the use of wires and breadboards, making physical computing more intuitive.

2.2.4 Analysis

This section has discussed the three types of embedded development board used across the domains of making, the IoT, and education. From prior discussion we can extract some emergent trends.

Microcontrollers are becoming more capable Over the past decade embedded development boards have moved away from 8-bit microcontrollers to more powerful 32-bit

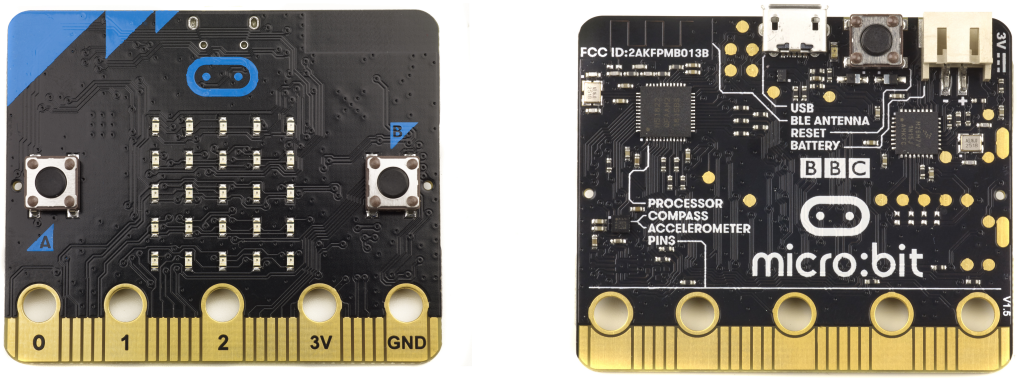

Fig. 2.20 Front of the BBC micro:bit (left) and back (right).

microcontrollers. These microcontrollers are not only more powerful but they are more capable too, with many of them coming with built in peripherals for digital signal processing and even wireless networking. Increased capability has not led to an increase in the cost of embedded development boards however, and now users are getting more versatility at less monetary expense. But whilst microcontrollers are more capable and powerful, they still have significantly less RAM and flash than traditional computers (between 2 and 512 kB). These constraints limit potential applications.

Integrated development boards are more intuitive Integrated boards with built in sensing capabilities can act as physical computing devices without any further prototyping. This has resonated well with technically inexperienced educators and students, leading to the wide adoption of integrated boards as tools for education. The relationship between integration and accessibility is further evidenced through the emergence of dedicated networking boards that directly incorporate wireless networking.

Ad-hoc networking protocols are not widely used Despite the trend towards integrated wireless networking, there are few embedded development boards that support ad-hoc networking protocols like Zigbee. One might expect the Arduino ecosystem to have a board that supports ad-hoc networking, but to date, no board has such support. Particle—a company that specialises in building bespoke IoT solutions—did offer an ad-hoc networking board, but discontinued it recently citing the complexities of ad-hoc wireless networking.

2.3 Programming languages and environments

Programming languages are used to create applications for computers—*including the microcontrollers on embedded development boards*. Programming languages are therefore another fundamental technology required to build a physical computing device.

Programming languages allow humans to express their intentions to machines as instructions, and through the process of interpretation or compilation, text-based instructions are converted into machine executable code. Interpreted languages convert program text or pre-generated byte code to machine code at runtime, whereas compilation performs that process ahead of time. In either case, computers execute the resulting code.

The ease of programming varies between programming languages. Low-level, compiled programming languages like C/C++ are known to be highly efficient but hard to program in. They generally have no abstraction, require manual memory management, and have a complex syntax. Higher level, interpreted languages, however, have the inverse relationship, delivering an easier programming experience at the expense of efficiency. Higher level languages are flexible, have an easy to understand syntax, and manage memory automatically.

This section provides a broad overview of programming languages and environments beginning with a discussion on those used generally for education (Section 2.3.1). We then move onto specific languages and environments used to create applications for microcontrollers when prototyping (Section 2.3.2) and when building products (Section 2.3.3).

2.3.1 Used for education

Programming is a hard skill to learn but teaches many of the concepts fundamental to computer science. It is unsurprising then that many tools for computer science education incorporate some form of programming as part of their experience. Traditional low level programming languages however are hard for beginners to use, and many of these tools seek to provide a simpler programming experience better suited to technically inexperienced students and teachers.

There are now many different programming languages designed to make programming simpler and more intuitive. This section covers these languages in the context of education, beginning with text-based programming languages and progressing to visual programming languages.

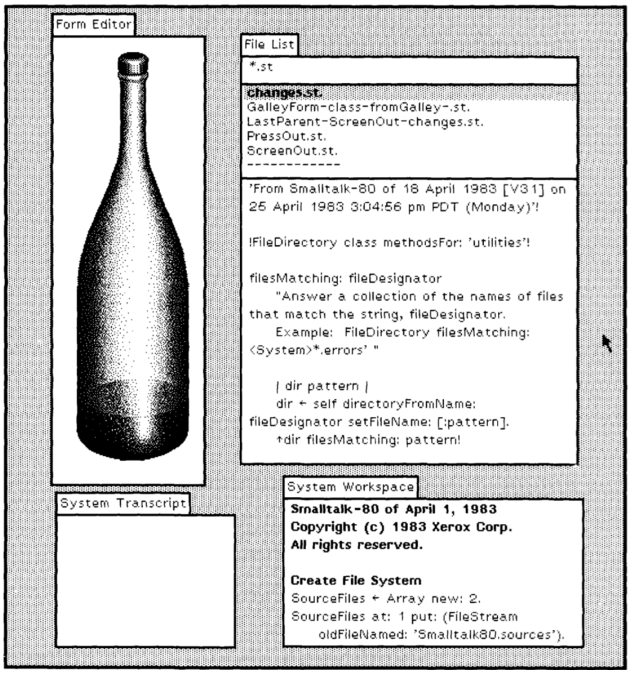

Fig. 2.21 The Smalltalk IDE [162] (left); and Smalltalk code sample (right)

Text-based programming languages

Text-based programming languages are the most traditional starting point for learning to program, but many low-level text-based programming languages (like C/C++) are hard for beginners to use. Over the past few decades there has been much effort to make text-based programming more intuitive. This has resulted in the creation of higher-level text-based languages that abstract away programming complexity and seek to make it possible for users to eventually transition to low-level programming languages. A specially designed IDE accompanies many higher level languages which hides underlying language processes (like compilation/interpretation) and in some cases provides a goal-based simulator to create a compelling learning experience. Because of their ease of use, many of the upcoming languages are used (or have been used) by educators in the classroom.

Logo [232] is one of the earliest higher level programming languages. It removes the complex syntactical elements of C/C++ and focuses on a refined vocabulary of keywords to create a friendly experience for beginners. Logo was specially designed for education and came with a self-contained Integrated Development Environment (IDE) containing a simple text editor and a turtle. Students could control the turtle using programs written in the text editor, creating a compelling and engaging learning experience.

```
1 def say_hello():
2     print("Hello World!")
3
4 say_hello()
```

```
1 sayHello = ()=> {
2     alert("Hello World!");
3 }
4
5 document.onEvent('ready', ()=>{
6     sayHello();
7 })
```

Fig. 2.22 A Python code sample (left); and functionally equivalent JavaScript code sample (right)

Smalltalk [162] (Figure 2.21) is considered one of the first higher-level languages to observe an object-oriented programming paradigm. It came with an IDE that contained a text editor and an environment for designing graphical user interfaces (GUI). Each GUI component was modelled as a discrete programming object, empowering users to more easily construct their own desktop applications programmatically. Though having a slightly more complex syntax than Logo by using full stops to denote the end of program statements, Smalltalk saw huge adoption.

Python [281, 248] (Figure 2.22, left) is a modern general purpose programming language. It is object oriented, has a small number of reserved keywords, and uses whitespace to structure code. Using whitespace to structure code is theorised to make the scope of code more obvious. A self-contained Python IDE is available but it is not particularly designed to facilitate education. Many educators however choose to use Python because of its real world applicability. Professional developers use Python to manipulate file systems, interface with the internet, create web servers, and even train machine-learning models with just a 'few lines of code'.

JavaScript [152] (Figure 2.22, right) is another popular general purpose text-based programming language. It is not especially designed for beginners but its universal adoption in the web allows beginners to learn and create from any device with a web browser. The language includes more advanced syntax like brackets, semi-colons, and braces, and is primarily used to bring additional functionality, animations, and client-side logic to HTML web pages. The use of JavaScript has also grown beyond the browser to include the desktop and mobile devices. Now professionals can write JavaScript applications to manipulate local file systems, interact with physical IO devices, develop scalable servers for websites, and create first-class applications for mobile devices. For educators, JavaScript provides an ideal stepping stone between languages without syntax (e.g. Python) and low-level languages with syntax (e.g. C/C++).

```
if cat.DistanceTo(Fish) < 1.0:
    doInOrder(
        cat.Move(Up, 0.5),
        cat.Move(Down, 0.5))
```

```
live_loop :flibble do
  sample :bd_haus, rate: 1
  sleep 0.5
end
```

Fig. 2.23 An Alice code sample (left); and Sonic Pi code sample (right)

There are also languages that are designed specifically for cross-curriculur learning. Alice [136] combines 3D animation with learning to program. Using Alice, students and teachers can manipulate and move objects in a 3D space to create animations. Though more recent versions of Alice support visual programming, the original IDE launched with a text-based programming language built on top of Python. The Alice programming language provides APIs that hide complex 3D animation algorithms, and concepts like for and while loops (Figure 2.23, left).

Sonic Pi [90] is a text-based programming environment that combines music and learning to program. Sonic Pi makes use of the flexible syntax of Ruby [153] to create a programmatic way of describing pitches, rhythms, and audio effects. Some effort goes towards hiding the complexity of for and while loops by encompassing every audio sequence in a do loop. The Sonic Pi IDE provides auto completion and real time error detection to ease the programming experience for students and educators (Figure 2.23, right).

Visual programming languages

Visual programming languages abstract away the complexities of text-based programming, providing users with a graphical means of building applications. Generally in these languages, segments of pre-defined code are connected together to build standalone applications or applications that work towards an objective. Visual programming languages are popular with educators because they allow students to focus on the structure and functionality of programs rather than syntax. But whilst visual programming languages allow students to learn fundamental (and translatable) computer science concepts, applications are constrained to what is permitted by each language.

RAPTOR [126] gives users a flow chart based visual programming language to build applications. Given a start and end point, users add nodes to the flow chart to build programs. The connections between nodes define the order of execution. RAPTOR provides many built in nodes, including ones for user input, for creating GUIs, and for primitive graphics (as seen in Figure 2.24). RAPTOR can also generate programs in many text-based languages, including Java or C#, allowing students and educators to build applications that run on

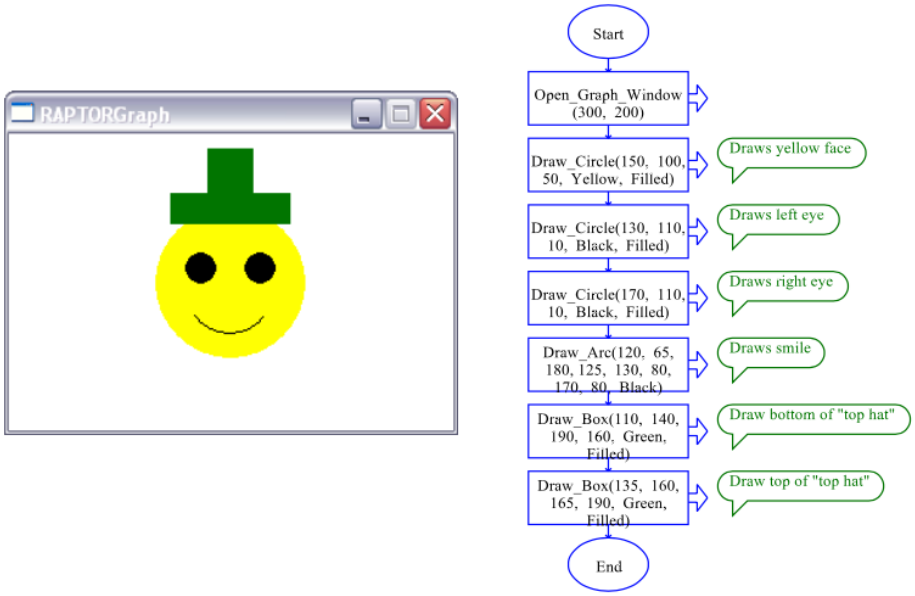

Fig. 2.24 The RAPTOR visual programming environment [126]. Students use a flow chart abstraction to create programs.

millions of devices. The flow chart model has proven valuable for educators, simplifying the explanation of loops and conditionals by making the flow of execution clear.

Scratch [245, 212] (Figure 2.25, left) is the most popular visual programming environment used for education. Scratch programs are composed of pre-defined blocks that can be pieced together (like Lego bricks). Blocks are coloured based up on their functionality and have different connector types and shapes to prevent incompatible blocks from being connected together. Scratch makes use of event-based programming to group code. For example, a piece of code may be invoked when two particular sprites collide or when application execution begins ("on start"). Using Scratch, users can create physics-based 2D games consisting of customisable animated sprites and graphics. Scratch has been shown to be successful in teaching novices translatable programming concepts in an engaging way [290] and is attributed with defining modern block-based programming as we know it.

With advancements in web technologies, block-based visual programming languages are moving to the web. Blockly [156] provides a framework for building custom web-based visual programming languages. The framework contains the basic elements of a programming editor and a set of visual blocks that can be customised to any application domain. A web-based programming experience means that educators do not have to get IT technicians to

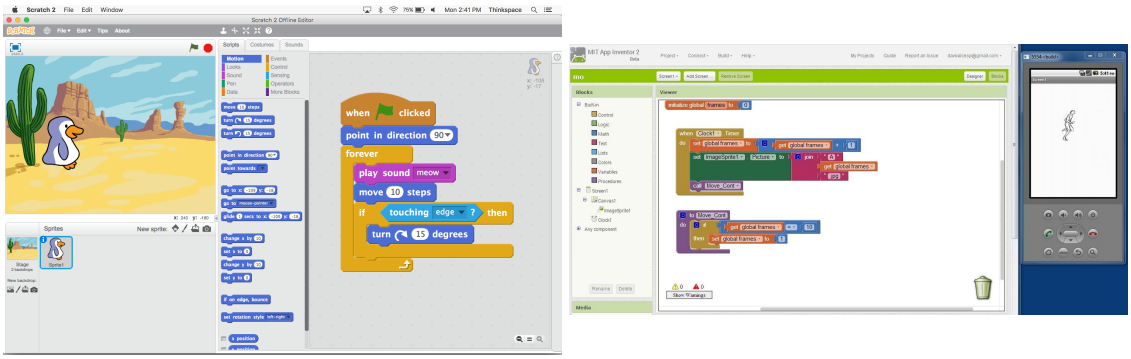

Fig. 2.25 Visual programming languages that use a block-based abstraction to create programs: Scratch [83] (left) and MIT App Inventor [47] (right).

install applications on computers—a well recognised organisational barrier in schools [138]. Scratch, in version 3.0, recently moved to the web using Blockly.

MIT App Inventor [252] (Figure 2.25, right) applies Blockly to mobile application development. A web-based interface builder lets users drag and drop GUI components onto a mobile application canvas. Components can be associated with code segments written using a custom Blockly dialect that makes heavy use of event-based programming. During program composition, App Inventor constantly reloads and deploys code "live". Completed applications can be deployed to a store for use on mobile devices by other App Inventors.

2.3.2 Used for prototyping

There are many languages and environments aimed at making programming microcontrollers, and by extension hardware prototyping, more intuitive. Hobbiest makers use the Arduino IDE (Figure 2.26) to program Arduino devices. The IDE presents a simple text editor and two default functions called 'setup' and 'loop'. The setup function is used to configure hardware before the loop function is repeatedly invoked—like the inside of a 'while' loop. Simple C/C++ APIs that give great control over the hardware are placed into each respective function to build applications. These APIs have their origins in Wiring [108], where APIs were designed to help artists incorporate electronics into their art pieces. The Arduino IDE compiles completed C/C++ applications and transfers them to Arduino devices using USB serial. The tightly integrated experience of Arduino has significantly lowered the barrier to hardware prototyping, but its language still poses barriers for novices.

Since the Raspberry has more memory and processing power than many Arduino boards it can directly support higher level languages like Python, JavaScript and Scratch. Users can interact with external circuitry using these higher level languages via the GPIO headers

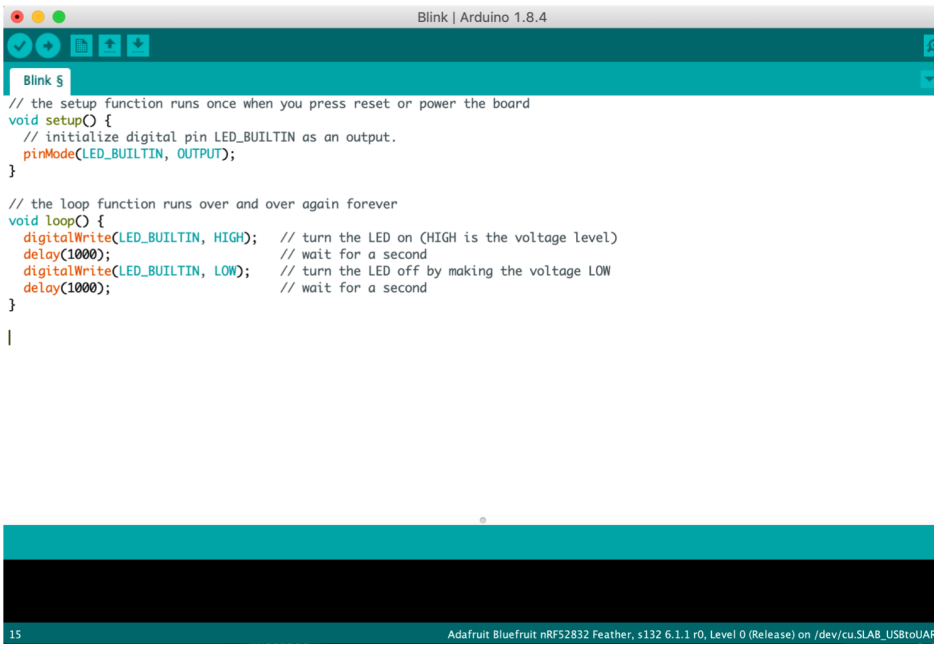

Fig. 2.26 The Arduino Integrated Development Environment (IDE)

mounted on the device. Whilst Scratch APIs offer limited control over the hardware, Python and JavaScript offer a huge amount of control. The extensive package ecosystem of both JavaScript and Python also provides many drivers for complex sensors. The small, portable form factor of the Raspberry Pi combined with these higher level languages creates a compelling prototyping platform. The device however cannot be powered by battery, limiting its use.

Many embedded development boards now support solutions like MicroPython [161], and variations like CircuitPython [11], that bring Python programming to resource constrained microcontrollers. These solutions provide many of the higher level primitives seen in standard Python and add additional modules to control the hardware it is running on. Python programs are transferred over USB serial to the microcontroller where programs are interpreted. Both MicroPython and CircuitPython vastly simplify the usually complex experience of programming microcontrollers and eases the transition from writing programs for computers, to writing programs for microcontrollers.

Espruino [289], DukTape [14], and JerryScript [160] are versions of JavaScript written for resource constrained microcontrollers. Each solution varies in the number of JavaScript language features that are supported. Like Python solutions for microcontrollers, program text is transferred over USB using a serial terminal. Some solutions perform additional

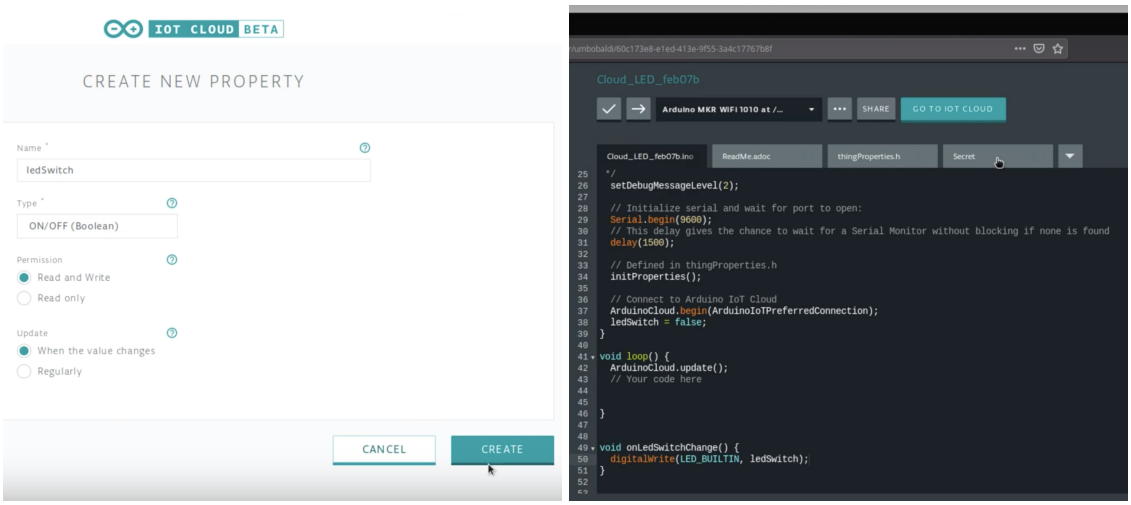

Fig. 2.27 The browser-based Arduino cloud environment Property creation (left) and integrated development environment (right)

optimisations like preserving previously interpreted byte code for more optimal execution. Once again, the use of higher level languages simplifies the hardware prototyping process.

Programming environments aimed at prototyping are also moving to the web. Mbed [97] is an entirely online embedded development platform where hobbyists and technologists alike can create C/C++ applications in a web-based IDE. Mbed provides a common Hardware Abstraction Layer (HAL) for many ARM cores, and gives users the option between a highly concurrent Real Time Operating System (RTOS) or a simple driver-oriented development experience. By offering these options, mbed is suitable for both prototyping and product.

The domains of making and the IoT are not mutually exclusive and now many technologists are prototyping hardware for the IoT. The Arduino IoT cloud [7] is a platform that integrates the familiar Arduino development experience with the web (Figure 2.27). Integration creates a cohesive end-to-end experience that enables application development, network configuration, and device management to happen all from within a platform agnostic web application. Before creating an application, a developer defines a number of properties (Figure 2.27, left). These properties are later injected into code for use by applications (Figure 2.27, right). The Arduino loop convention is used to poll Internet endpoints to detect changes to properties. Of course, for the application to work an Internet capable Arduino development board is required.

2.3.3 Used for product

During the creation of a product, companies seek to minimise cost to maximise profit. One area that is ripe for optimisation is the electronic aspect of a product. Here, any cost reduction can have a significant and positive impact on profit margins. It is therefore the job of professional electrical engineers to optimise for cost wherever possible. Since microcontroller memory and processing power translates to cost, embedded software developers have the challenging job of writing highly functional applications with limited microcontroller resources.

Real Time Operating Systems (RTOSs) strike an appropriate balance between functionality and efficiency for embedded software developers. RTOSs provide an environment suitable for hard real time applications and are highly concurrent in order to manage the competing attention of different hardware peripherals. They also provide the appropriate primitives to manage competition over hardware and software resources. Though there are many open source RTOSs available like Zephyr [89] and mbed OS [97], many development teams choose to create their own [40]. This allows them to better match RTOS behaviour to their application.

The advent of the IoT presents many new business opportunities, including those that better enable other companies to build products for the IoT. Pelion [8], by ARM, gives businesses an entire IoT stack to connect, manage and program devices. The Pelion programming environment is web-based and compatible with any ARM-based embedded device. It is primarily intended for use with mbed OS [97] but has libraries to support Arduino and many other real-time operating systems. Applications that make use of the Pelion communication stack have a secure connection to the cloud and their firmware can be remotely updated.

Particle devices can also be connected, managed, and programmed from the web. Particle customers create Arduino applications that use particle libraries to expose local variables to the cloud and vice versa. Applications poll internet endpoint to detect changes to variables in a similar way to Arduino Cloud. Arduino APIs are mapped to an underlying common device runtime (i.e. an RTOS) that adds support for concurrent applications. This runtime also abstracts hardware, transparently mapping networking calls to whatever wireless protocol is supported by the development board. Particle can automatically patch security vulnerabilities in the device runtime, and businesses can dynamically deploy firmware updates. The use of Arduino hides the underlying complexity of the device runtime, lowering the barrier to entry to IoT product creation.

Azure Sphere devices run a custom modified linux distribution called Azure Sphere OS. Designed for security, user applications are split into high-level applications and real-time

applications. High-level applications run inside a container and can only interact with the operating system and selected libraries. Real-time applications however are allowed to directly interact with the hardware and either run 'bare metal' on the main processor or on any of the two co-processors. High-level applications can communicate with real-time applications and the Internet, but real-time applications cannot. Applications for Sphere can be written using any code editor but Sphere devices are factory secured and require special modification for direct programming. For live products, firmware can only be updated using the secure Azure device management portal.

2.3.4 Analysis

This section has explored different programming languages and environments and their use for education, for prototyping, and for product creation. From the discussion above, we can extract a few key observations.

Programming is moving to the web. The web browser is becoming the new development environment. Educators and students can now access tools like Scratch from any device with a web browser. Hobbiest makers can use Arduino Cloud to create applications for Arduino devices without installing any software. Even professional embedded developers can use write low-level C/C++ firmware without installing a single compilation toolchain.

Programming languages are becoming more intuitive. There is clear trend towards making programming more intuitive to those with little technical expertise. Educators use higher-level languages like Scratch and Python to teach students fundamental computer science concepts. The same trend is even evident in the resource constrained world of the microcontroller where Hobbiests now use MicroPython and Espruino as a convenient stepping stone to hardware prototyping. Professionals however continue to use C/C++.

For microcontrollers, efficiency is paramount Though more intuitive languages are available, a large proportion of the aforementioned programming languages for microcontrollers make use of C/C++. This is because microcontrollers have significantly less RAM and flash than personal computers, requiring the careful use resources, currently only intuitive through low-level programming languages. Professional embedded software developers and even hobbyist microcontroller programming environments, like Arduino, choose to use highly efficient languages like C/C++.

Concurrency is king Prior discussion shows that programming languages that support concurrency offer many benefits. In education, languages like Scratch and App Inventor make programming more intuitive to technical novices through the heavy use of asynchronous event-driven programming. For professional embedded developers, concurrency is key to managing the many demands of real-time and interactive devices.

2.4 Hardware composition

Everyday citizens compose hardware. It happens when we plug our headphones into audio systems, our keyboards and mice into computers, and our smart devices into mains chargers. The purpose of connecting devices together is to allow them to communicate with one another over an electrically conductive physical medium like a cable.

A physical computing device is incomplete without the sensors that enable its interactivity and hardware composition is therefore also part of building a physical computing device. Here, sensing peripherals are connected to microcontrollers using an electrically conductive medium. Typically, however, conductive mediums are exposed electrical contacts on PCBs rather than cables. Electrical signalling is used to communicate between microcontrollers and peripherals, and wired protocols standardise electrical signalling for complex peripherals.

This section covers the different ways electrical contact is made between microcontrollers and peripherals (Section 2.4.1), the fundamentals of electrical signalling (Section 2.4.2), and the wired protocols that use electrical signalling to communicate (Section 2.4.3). The wired protocols discussed in this section specifically relate to those used for prototyping. A more detailed treatment is provided later on.

2.4.1 Making electrical contact

When prototyping there are broadly three ways of connecting microcontrollers and peripherals. The first, wiring, is the traditional way to compose a physical computing device. The use of wires typically requires electrical expertise however, and the second and third ways seek to simplify hardware composition. One allows peripherals to be stacked on top of embedded development boards, and the other allows embedded development boards to slot into peripherals.

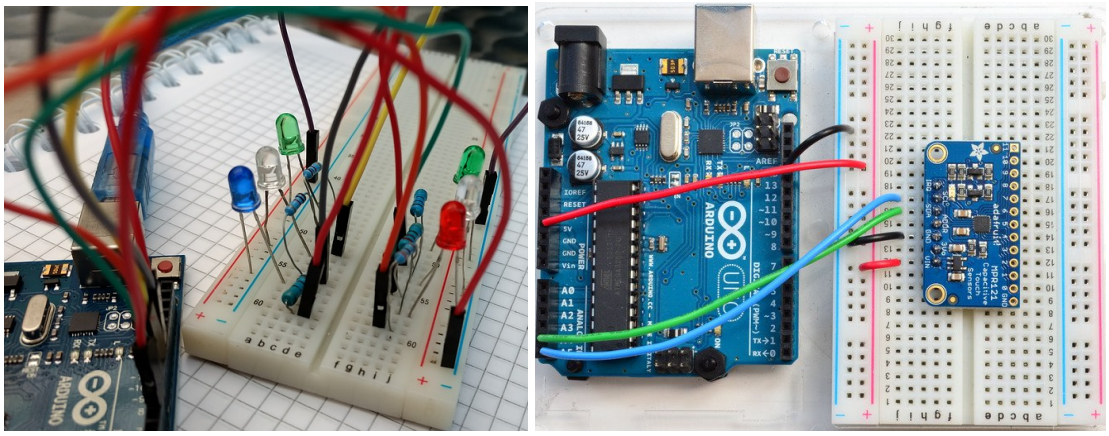

Fig. 2.28 Using prototyping wires to connect an Arduino to an external breadboard [51] (left); and connecting a peripheral module to an Arduino [52] (right).

Wiring

Wires are the most common way to compose a physical computing device and give users the freedom to connect peripherals and electronic circuits to microcontrollers using just two malleable contacts. Wires can be made from a solid core or split core—the latter is easier to manipulate. When composing a physical computing device, wires need to be trimmed to length, stripped of their non-conductive outer shell to create two electrical contact points. Each contact is affixed to components using solder to form an electrical connection. A hot iron is used to melt conductive solder onto electrical contacts to form a solid joint.

Specially designed *prototyping wires* let users more easily compose physical computing devices. Prototyping wires are pre-cut and have reusable connectors attached to either ends of the wire. Connectors are either a conductive plug or a receptacle designed to receive a plug. Breadboards allow users to mount through-hole components and connect prototyping wires between components to create external circuits. Breadboards can also be connected directly to an embedded development board using prototyping wires (Figure 2.28, left). Wires are usually connected to receptacles mounted on the PCB itself, and PCB receptacles are in turn connected to microcontroller GPIOs.

External peripheral modules can also be connected to embedded development boards. Peripherals typically use wired protocols, like I2C or SPI, to communicate with the microcontroller on the embedded development board. A minimum of four lines must be routed from the microcontroller to peripheral in order to communicate. Embedded development boards generally support prototyping with both external modules and breadboards (Figure 2.28, right).

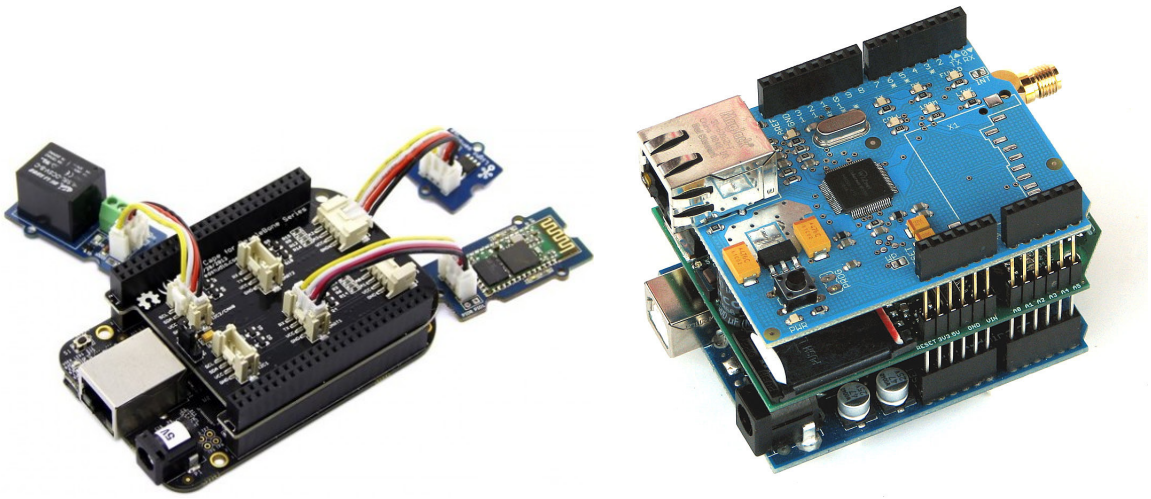

Fig. 2.29 Connecting Grove modules to a Grove expansion board [62] (left); and 'stacking' shields onto an Arduino [53] (right).

The prior connection mechanisms require a basic knowledge of electronics, incompatible with technical novices. Electronics manufacturers have therefore sought to create new and simpler ways to connect peripherals to microcontrollers using wires. The Seeed Grove ecosystem of hardware [20] consists of main development boards and external peripheral modules that use a custom cable and connectors to join peripherals to microcontrollers (Figure 2.29, left). The cable consists of four internal wires, two are used for data communication and two are used for power and ground. The grove cable therefore supports simple digital/analogue IO, and a subset of wired protocols (I2C, and UART). The 4-pin plug connectors at either end of the grove cable only allow users to mate peripherals with PCB mounted receptacles in one orientation, guaranteeing electronic compatibility. Grove users must plug peripherals into the PCB receptacle that match the communication protocol of the peripheral.

Unbeknownst to many, we compose hardware all the time in our day-to-day lives. When we charge our phones or connect a keyboard or mouse to a personal computer, we are composing hardware. USB connectors and cables supports the examples above, and many more beyond that. The careful design of USB cables and connectors has led to their wide adoption. Devices with USB connectors typically communicate via the USB protocol.

Stacking

Other communities seek to make composing physical computing devices easier by entirely removing wiring from the process. Instead, inflexible, but compositional, peripheral PCB

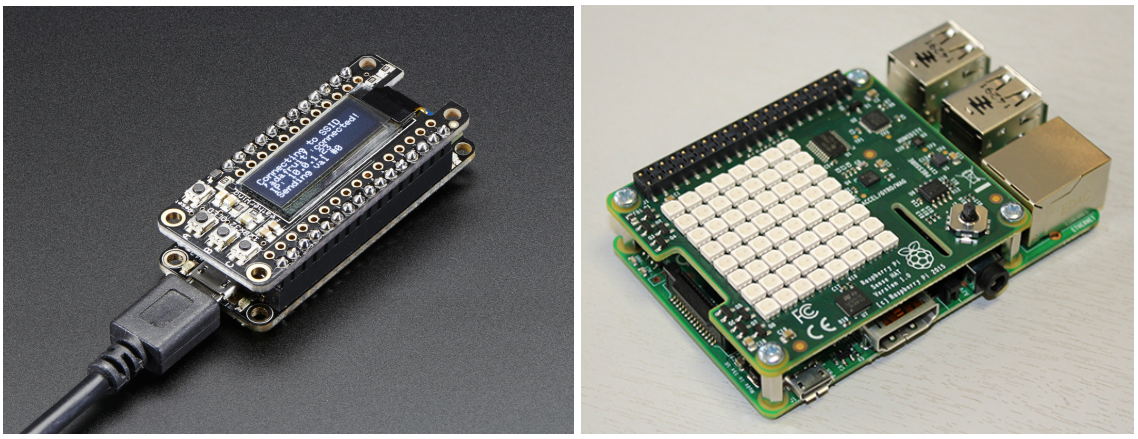

Fig. 2.30 Adding a wing to an Adafruit feather board [60] (left) and placing a SenseHAT on a Raspberry Pi [82] (right).

boards can be 'stacked' on top of embedded development boards. Stackable PCBs originate from the PC/104 consortium [30] and their use for hardware composition was made popular by Arduino ecosystem. Here, stackable PCBs are called *shields* and shields expose plugs that directly mate with the PCB receptacles of the Arduino (Figure 2.29, right). They also support the stacking of additional shields by mapping unused GPIO through to the Arduino. Shields are only compatible with boards that follow the original form factor and pin out specification of the Arduino Uno.

The Adafruit ecosystem of peripherals adopt the Feather form factor, created due to the size of Arduino Shields [18]. The Feather specification defines the size, spacing, GPIO layout, and naming conventions for all Feather boards. Feather boards typically contain a reprogrammable microcontroller and Wings can be added to Feathers to add new peripherals and communication interfaces. Like Arduino Shields, feathers are stacked on top of one another using plug connectors (Figure 2.30, left).

The Raspberry Pi ecosystem uses Hardware Attached on Top (HATs) [22] to augment Raspberry Pi devices with sensors and co-processors. All HATs follow a common specification that defines compatible pin layouts for each type of Raspberry Pi. HATs are mated to the exposed GPIO of the Raspberry Pi and can be affixed to the Pi using screws for more rigidity and security (Figure 2.30, right).

Slotting

Prior means of hardware composition are less than ideal. Wiring is a well recognised barrier for technical novices [116], and the exposed pins of stackable shields can snap over time,

requiring great expense to replace. The classroom is an area where both of these concerns are prevalent. Educators and students need an easy means for composition with low monetary expense [115].

The BBC micro:bit is an integrated development board designed for the classroom. Though it has many built-in capabilities, its creators did not want to limit the potential of its application. The bottom of the micro:bit, known as the *edge connector* [15], exposes a number of electrical contacts that are routed to microcontroller GPIO. The edge connector can be therefore be controlled by software and used to interface directly with external circuits.

There are multiple ways to connect the edge connector to external circuits. Solder can be used to attach wires directly to electrical contacts, banana plugs can mate with the widely bored holes at its top, or, more ruggedly, a specially designed receptacle can mate with the entire edge connector. The receptacle provides an easy 'slot-based' way to connect the micro:bit to peripherals, creating an experience that is similar to the way credit cards are 'slotted' into terminals.

Accessories make use of the edge connector in different ways. Some accessories simply provide a 'break out' board for the micro:bit that converts the edge connector into prototyping wire compatible pins (Figure 2.31, left). Other accessories, like the Macqueen remote control car accessory (Figure 2.31, right), directly integrate the micro:bit into the chassis of the car itself.

As only one accessory can be connected at a time, the edge connector is inherently non-compositional. Its success with those with little technical expertise however means that many other embedded boards manufacturers are following suit. The MxChip IoT Development board [5], the Brainpad Arcade [10], and the Meowbit Arcade [24] all connect to peripherals using the same edge connector specification as the micro:bit. This means that peripherals designed for the micro:bit are also now compatible with other development boards.

2.4.2 Electrical signalling

Microcontrollers communicate using analogue or digital electrical signalling over physical mediums (i.e. wires) that conduct electricity. Both types of electrical signalling express data by modifying the line voltage between the minimum and maximum supported voltage. Line voltage typically ranges between 0 and 3.3 volts but this is often changes depending on application.

In digital signalling, line voltage represents a set of discrete values. Typically for microcontrollers there are two discrete values: zero (0 volts), and one (3.3 volts) (Figure 2.32,

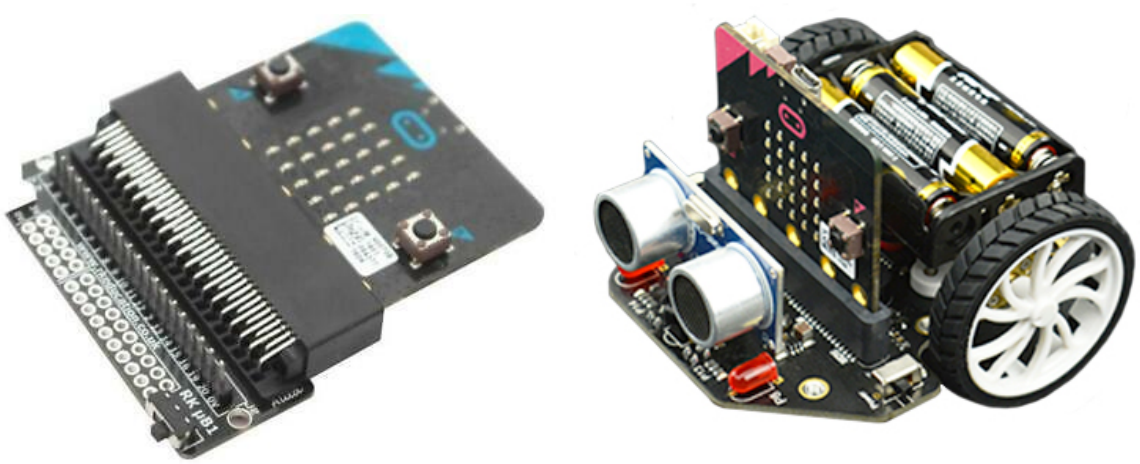

Fig. 2.31 Edge connector accessories for the BBC micro:bit. A break out board [58] (left); and a remote controlled car called macqueen [73] (right).

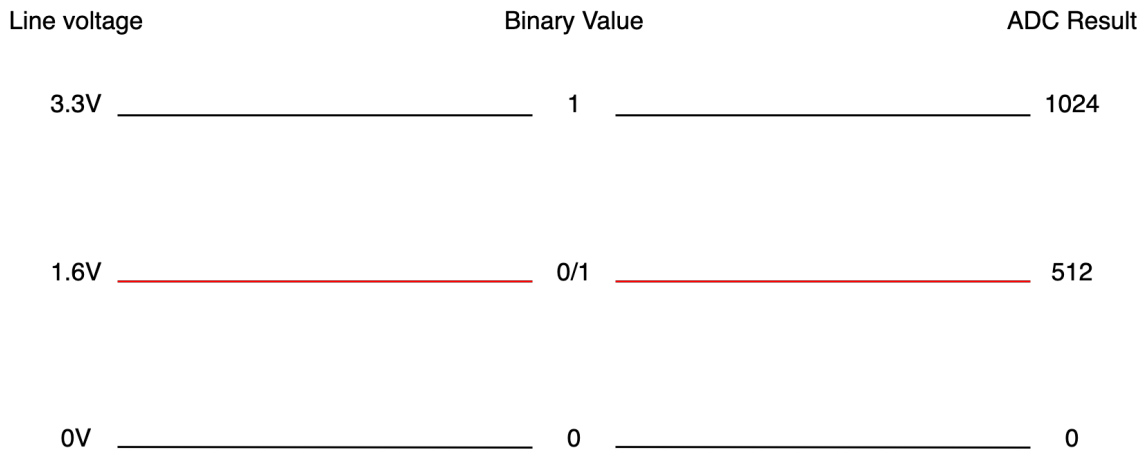

Fig. 2.32 Line voltage (left), binary value (middle), and corresponding 10-bit ADC value (right).

middle). Digital signalling is universal to all microcontrollers and applications can easily detect changes in voltage by polling the state of microcontroller GPIO. Many microcontrollers now also come with dedicated peripherals for asynchronously detecting immediate changes in voltage. Simple digital sensing is used by sensors like buttons, where button state is encoded into binary digital values.

In analogue signalling, line voltage is used to represent a continuous range of values. An Analog-to-Digital Converter (ADC) is required to approximate line voltage to a number where 0 volts represents the minimum value, and 3.3 volts represents the maximum. The range of possible values is confined to voltage range and ADC resolution—an ADC with a 10-bit resolution only allows for 1024 different values (Figure 2.32, right). Analogue signalling is especially useful for simple sensors that change line voltage proportionally to what they are sensing (i.e. a thermister or light dependent resistor).

2.4.3 Wired protocols

Simple electrical signal quickly become limiting when more complex sensors are required. Take an accelerometer for example. Accelerometers can report highly accurate acceleration in the x, y, and z axis. But representing each axis as an analogue range or digital binary values does not yield enough accuracy to be useful. Moreover, implementing sensing with individual lines would quickly consume microcontroller GPIO. Long ago however, the embedded development community settled on a number of wired protocols that standardise electrical signalling for more expressive, multi-peripheral, abstract communication between microcontrollers and peripherals. These protocols build on basic analogue and digital signalling.

RS232 [254], otherwise known as UART is the simplest of these more expressive protocols. It is a bi-directional point-to-point protocol that supports the transmission of arbitrary data. UART is therefore generally used for peripherals that have a variable stream of data like GPS sensors. Reducing communication to a byte stream gives software developers great flexibility, allowing them to define their own packet structures for communication.

RS232 is known as an asynchronous protocol and minimally requires four wires to operate: two data lines for bi-directional communication, and two lines for power and ground. Asynchronous protocols require both receiver and transmitter devices to have an accurate clock in order to decode data signals. Signals are decoded according to a pre-defined clock rate that must be known ahead of time by receiver and transmitter. Clock pulses define the rate at which the data line voltage is sampled, ultimately defining how data is received.

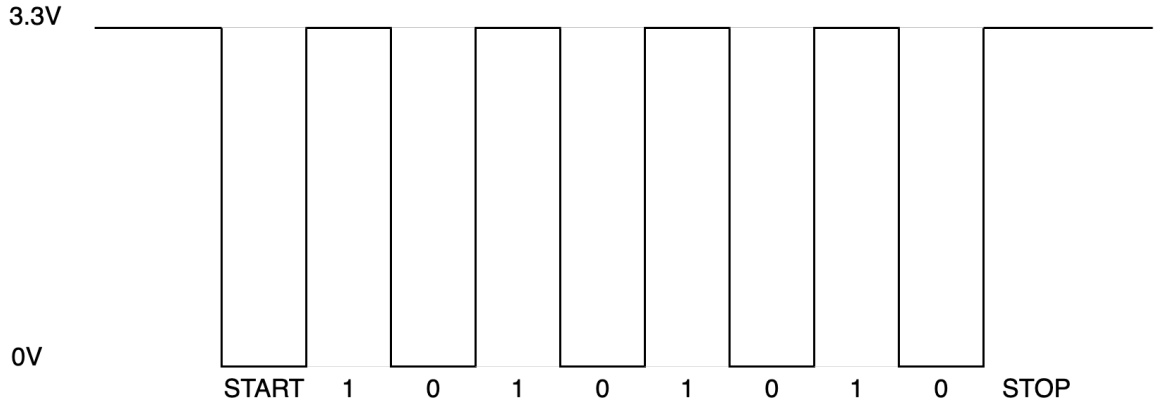

Fig. 2.33 The UART data signal.

Figure 2.33 shows the electrical signalling of a single UART byte. As seen in the figure, UART operates digital electrical signalling and represents binary data by switching line voltage between 0 and 3.3 volts. Default UART implementations leave the line at 3.3 volts when not communicating, and a transmission is signalled by a single start and stop bit. Transmission polarity and bit packing are configurable and it is possible to support additional bits for basic error correction. For detail on the electrical signalling of other protocols discussed in this section, please visit their individual specifications.

I2C is a bus-based wired protocol is designed to efficiently connect microcontrollers to peripherals on a PCB. A single central device (the microcontroller) controls operation of the bus, interrogating peripherals to retrieve data. Each I2C peripheral has an address and a corresponding register map defined in a data sheet and it is the job of an embedded developer to turn a data sheet into a software driver for use by applications. Software drivers map low-level I2C address and register combinations to easy to use APIs. For instance, there may be a single API to access the x, y, and z acceleration data of an accelerometer.

I2C is known as a synchronous protocol and requires four lines to operate: a data line, a clock line, power, and ground. The clock line is used to define the sample rate for the data line. This means that only the central device—which drives all communication—needs to have an accurate crystal. Because I2C was designed for interconnecting components on a PCB, it does not permit the connection of multiple central devices, the multiple connection of the same peripheral, nor the dynamic connection of peripherals.

The Serial Peripheral Interface (SPI) [199] is another bus-based protocol that interconnects a single central device with one or more peripherals. Instead of using addresses, SPI routes individual GPIO lines to each peripheral. The individual GPIO lines are used to signal

that a peripheral should respond to a query issued by the central device and peripherals still observe a register/command map to access device specific functionality. SPI is a synchronous protocol and requires minimally four lines to operate: data, clock, power, and ground. Of course, additional lines may be required if more than one peripheral is connected to the bus.

USB [264] is a bus-based protocol that is used to interconnect peripherals, like keyboards and mice, to personal computers. USB supports significantly faster data rates than the protocols discussed above and it is also suitable for high throughput devices like cameras. One USB host directs communication with one or more peripheral devices. Peripherals present interfaces to the host and these interfaces are used to load software drivers for peripherals dynamically. A subset of commonly supported drivers can also be leveraged by USB peripherals, allowing devices to universally support common devices without any additional installation. Whilst many microcontrollers support protocols like I2C, SPI, and UART, few low cost microcontrollers support USB. This is because more complex protocols require more silicon, leading to an increased microcontroller cost. The level of dynamism and ease of use however, has made USB the most widely used wired protocol in the world.

2.4.4 Analysis

Connecting devices to peripherals is central to building a physical computing device. This section has looked at the various ways microcontrollers and peripherals are physically connected, and the protocols that enable communication. From these discussions we can draw the following conclusions.

Citizen developers need dynamic composition The ability to dynamically connect peripherals to personal computers using USB has made high-level hardware composition intuitive to the masses. Many protocols for low-level hardware composition however have no such capability and assume that microcontrollers and peripherals are placed statically on a PCB.

Citizen developers need hardware abstraction Similarly, USB also evidences the importance of hardware abstraction. Instead of requiring specialised drivers for each peripheral (like with I2C or SPI), USB abstracts common functionality into general purpose drivers. This means that users can connect any keyboard to a personal computer, and it will work without any additional installation.

Dynamic protocols are not widely available Prior sections also illustrate that protocol dynamism translates to an increase in protocol complexity. Protocol complexity translates to an increase in cost, and now microcontrollers widely support simple protocols like SPI and UART, but do not support highly dynamic—but more complex—protocols like USB. As evidenced in Section 2.1 however, many technologists are now dynamically and iteratively prototyping with hardware. Instead of dynamic protocols, they use highly static and efficient protocols that were not designed with dynamic connectivity in mind.

Citizen developers need intuitive connectors Simpler and more robust connectors are required for those with little expertise. This is clearly evident in prior discussion where technologists use freely connectable wires, and technically inexperienced educators and students use rugged, but easy-to-use, slot-based connectors. In the world of mainstream hardware composition, USB provides easy to connect cables and connectors.

2.5 Wireless networking

Everyday citizens use wireless protocols to connect to the Internet—connecting to the Internet is often just a case of entering a single WiFi password. Wireless networking is so pervasive now that we do not even have to think about how our devices are connected together and it is easy to forget how complex the world of wireless networking has become.

Take for example a modern home Local Area Network (LAN) where there are a perplexing number of wireless protocols in use (Figure 2.34). Here, laptops stream audio downloaded from the Internet using WiFi to Bluetooth headphones. Smart phones simultaneously communicate with other cell phones using LTE and with LAN devices using WiFi. And smart assistants receive commands from the Internet using WiFi to control smart motion sensors via Zigbee (top right).

There is good reason for this homogenous collection of wireless networking protocols. Different application requirements demand the use of different protocols. Some devices, like smart motion sensors, require low throughput low data rate protocols for increased battery lifetime. Other devices, like laptops and smartphones, require high throughput high data rate protocols to quickly deliver media rich content. Regardless of the protocol however, wireless signals ultimately need to be translated into electrical signals and sent on to the Internet via a physical cable.

Communication between Internet connected devices is standardised by the Internet Protocol (IP). The protocol standardises maximum packet sizes, addressing schemes, and

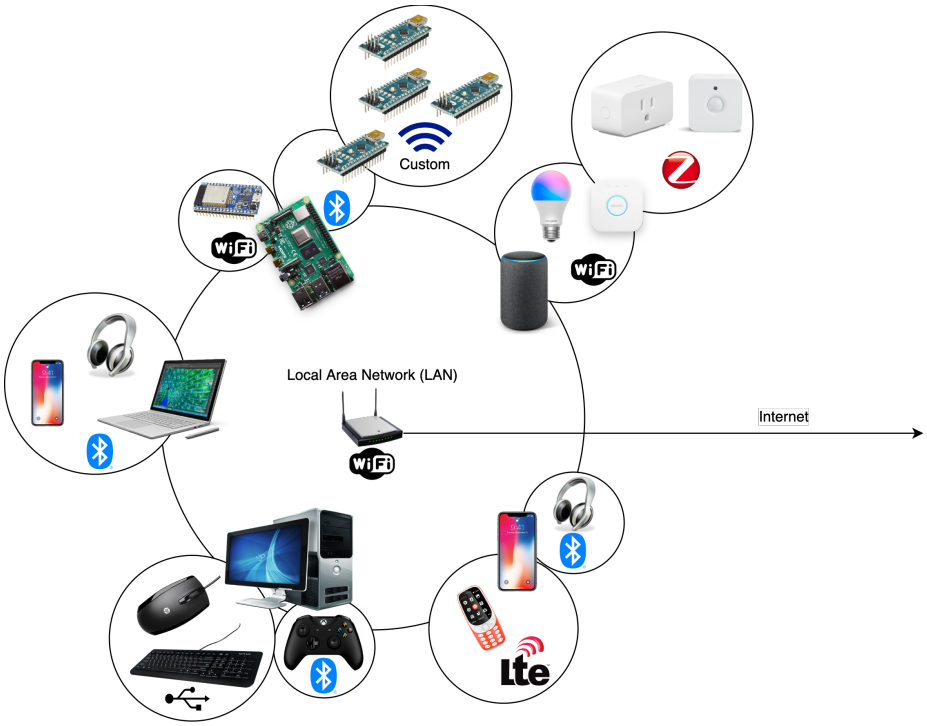

Fig. 2.34 Networks and subnetworks within a modern home LAN.

transmission frequency for communication via the Internet. It is designed to work transparently across any communication medium, including wireless protocols. IP was created long before the emergence of the Internet of Things (IoT), and is incompatible with the resource constrained world of IoT devices. As a result, many wireless protocols do not support IP.

Disparity in support for IP makes wirelessly networking devices especially challenging. Highly energy efficient protocols, like Zigbee, require a significant amount of protocol translation to convert Zigbee packets into an IP compatible format. The process is so complex that protocol translation requires the use of a dedicated bridge device. The use of Zigbee is similarly complex and requires infrastructure to make its energy efficiency gains. These factors make the use of any protocol other than WiFi a challenging proposition when citizen developers are involved.

The remainder of this section discusses wireless signalling (Section 2.5.1), and the protocols that standardise wireless signals to enable universal communication between devices (Section 2.5.2).

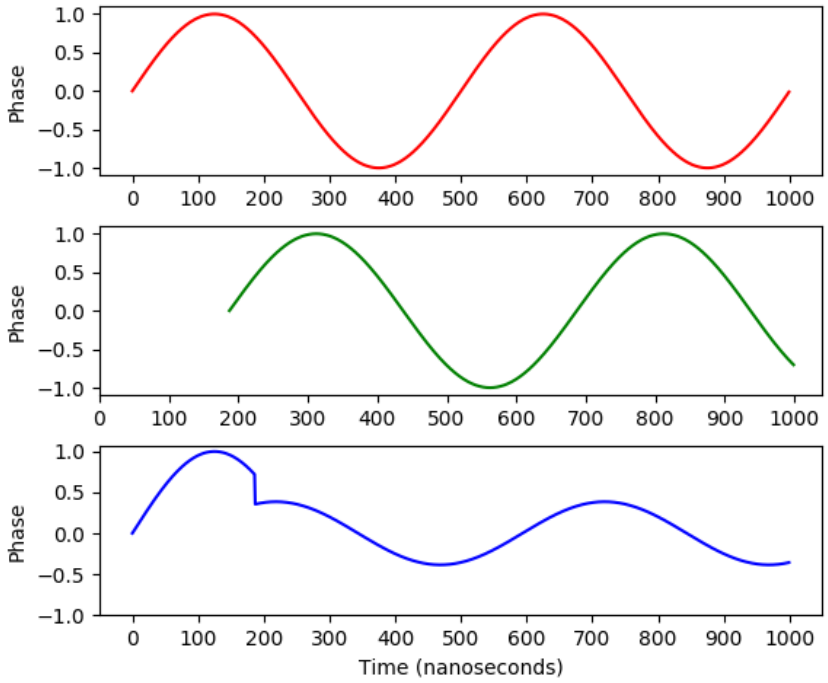

Fig. 2.35 The resulting weaker waveform (bottom) from summing the top and middle waveforms.

2.5.1 Wireless signalling

Wireless protocols use radio waves to communicate. Radio waves have well observed properties and are dimensioned by phase, frequency, and amplitude. *Phase* is the orientation of the wave as it moves through time and *frequency* is the rate at which the phase changes. *Amplitude* can be considered the energy of the wave and is the difference between the highest point of the wave and zero. When identical waveforms are produced, the resulting amplitude is the sum of all waveforms (Figure 2.35). Summation can either strengthen or weaken waveforms.

In wireless communications there are two types of signal, the *data* signal and the *carrier* signal. The data signal represents binary data by modifying the amplitude, phase and frequency of waves. Waveforms are defined to represent binary symbols, and the data rate is how many symbols can be represented per second. A 1 megabit data rate means that 1,000,000 symbols can be represented every second at a 1 microsecond granularity. The data signal is modulated by a carrier signal to improve transmission reliability. The carrier signal modulates by a fixed frequency (like 2.4 GHz) affecting signal permeation, and ultimately reception distance.

The success of wireless communications is dependent on the environment and the location of transmitters. We summarise the properties that affect signal propagation below:

Multipath Different environments affect the propagation of radio waves. Surfaces like glass can reflect waves and cause multiple paths of propagation. Different propagation paths cause the same signal to be received at different times. This can strengthen, or more likely, weaken signals (as can be seen in Figure 2.35).

Environmental interference Different materials also affect the propagation of radio waves. Thin materials allow radio waves to pass through mostly unaffected, whereas thick materials like cement, concrete, and metal, almost entirely prevent the propagation of signals.

Signal interference Both data and carrier signals are subject to interference. Interference comes from other devices transmitting data out of phase with the desired signal. Even if devices transmit the same data, offsets in the time domain can lead to signal cancellation (Figure 2.35).

Capture effect [200] Radio receivers track signals that are stronger and ignore competing weaker signals. Signal strength is affected by transmission power and the factors above. Therefore, as devices move through an environment and signal strength changes, receivers are subject to lock onto different radio waves.

2.5.2 Protocols

Wireless protocols standardise the meaning of symbols and provide additional layers of abstraction to make it easier to communicate between devices. These layers of abstraction are categorised by a conceptual model known as the OSI model [120].

The OSI model normally has 7-layers, but we use five for conciseness. The *physical layer* handles the transmission and reception of raw wireless symbols over a shared medium (See Section 2.5.1). The *link layer* manages physical addresses, mediates access to the physical layer, and decodes symbols into bits (and vice versa). The *network layer* defines packet structures, conceptual network addresses, and data routes between devices. The *transport* layer enables multi-packet transmission between devices, where streams of data are decomposed and recomposed into multiple packets. And the *application* layer defines communication APIs for use by software developers.

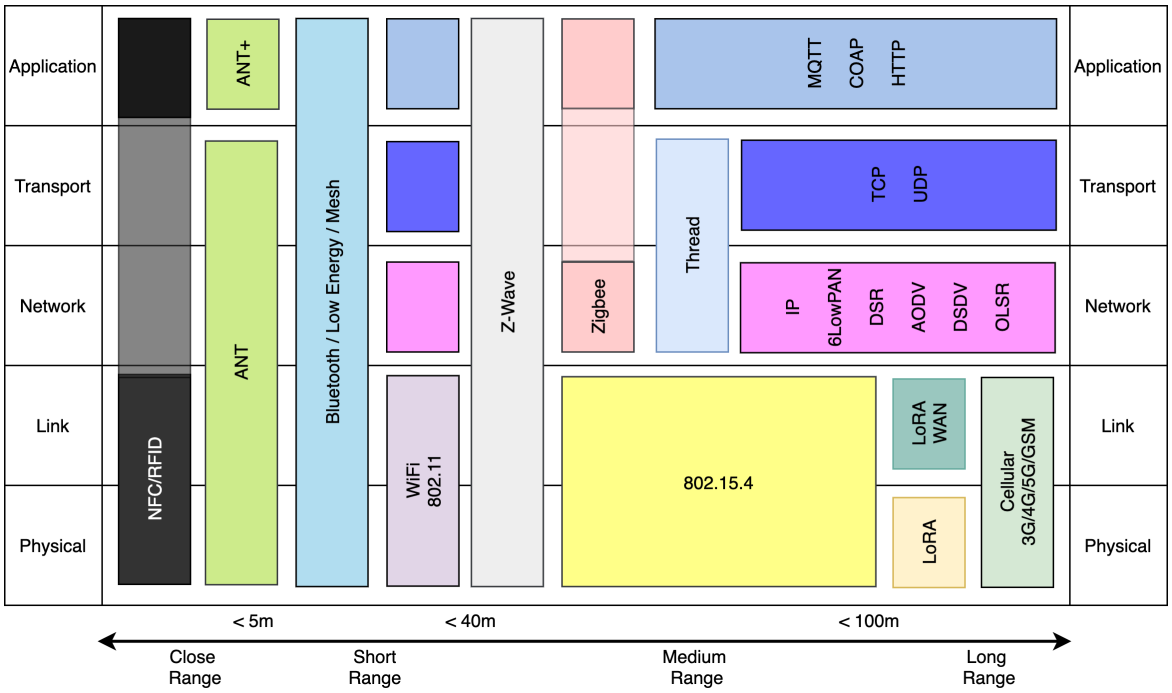

Fig. 2.36 Layers of the Open Systems Interconnect (OSI) model defined by protocols in this section. Protocols are ordered by first hop transmission distance.

Every wireless protocol defines part of the *Open Systems Interconnect (OSI) model* and Figure 2.36 summarises the layers defined by popular wireless protocols ordered by first hop transmission distance. Some protocols skip layers of the stack for efficiency and this is shown in the figure by the use of faded colours. We include higher level protocols for context (i.e. IP, TCP, MQTT) that build on popular physical standards like WiFi.

Upcoming discussion of wireless protocols is also ordered by first hop transmission distance. We discuss each protocol in relation to their: data rate, throughput, energy consumption, and application.

Close range

Close range protocols operate over really small distances (< 5 metres) and are used for the purposes of personal identification and tracking. Their small transmission distance leads to greater power efficiency.

Near Field Communication (NFC)/Radio Frequency Identification (RFID) [99, 288] both enable communication between a passive tag and a reader. The passive tag harvests energy from the reader, powers on, and transmits a small amount of data to the reader (usually less

than 2 kB). However, whilst the structure and format of RFID transfers are fixed, NFC allows the secure exchange of arbitrary data streams. NFC has therefore seen significant adoption in proximity-based transaction systems, whilst RFID is typically used to secure products in shops. Due to their short range and low data rate, both NFC/RFID are very power efficient.

Adaptive Network Topology (ANT) [184] is used to connect low-power fitness tracking devices, like heart monitors, to Internet enabled exercise machines for more accurate calorie tracking. Application specific profiles are loaded onto ANT devices, and a dedicated driver is required for every different profile. *ANT+* adds profiles that abstract common functionality so that any application can freely communicate with an ANT device. Once again, due to low throughput and short transmission distances, ANT is very power efficient.

Short range

Short range protocols typical operate over distances smaller than 40 metres and are used for the purpose of personal networking. *Bluetooth Low Energy (BLE)* [163] is a power efficient wireless networking standard that emerged alongside the IoT. Efficient power consumption is achieved through two device roles: *central* and *peripheral*. Central devices control the operation of one or more peripheral devices, accessing data when required by user applications. Host driven communication means that for the majority of the time, BLE peripherals can leave receivers disabled. Improved energy efficiency leads to greater longevity using smaller batteries, and BLE is used in high data rate, low throughput devices like heart rate monitors, personal device trackers, and smart beacons for indoor localisation.

Bluetooth [113] offers faster data rates than BLE at the cost of increased energy consumption. The paradigm offered by Bluetooth is also central-peripheral and common applications include wirelessly interconnecting human interface devices to personal computing devices. Connecting Bluetooth headphones, where a higher data rate is required, is one example where Bluetooth is used over BLE. Both Bluetooth and BLE allow one central device to communicate with up to 7 peripheral devices. For each connection, devices must be paired together using error-prone pairing methods [193].

Whether convening in a public space, at work, or in transit to another location, *WiFi* provides a universal and familiar way of wirelessly networking devices. WiFi networks require just a password and network name to connect and are scalable to tens of devices. For good connectivity and fast data speeds, close proximity and minimal obstructions are required. Due to high data rates, power consumption for end devices is generally quite high in comparison to other networking protocols. It is therefore intended for use by devices with easily replenishible or constant power supplies.

Medium range

Medium range protocols are generally more dynamic and collaborative than short or close range protocols. Medium range protocols partially gain greater distance through different wireless signalling, but mostly make further gains through ad-hoc networking. With ad-hoc networking, devices can network with one another to extend communication range beyond a single hop. Ad-hoc networks are best suited to low power, low throughput devices.

Z-Wave [159] is an ad-hoc networking protocol designed for use in home automation and industrial control applications. Z-Wave devices can form a network on-the-fly using a pairing mechanism similar to Bluetooth or BLE. Packets from devices are propagated through the network in waves and can travel up to four network hops; a single network hop can represent a distance of 75 metres. For improved energy efficiency, Z-Wave can only communicate data at 100 kbps.

Zigbee [187] is designed for low power ad-hoc networking. It builds upon the 802.15.4 wireless standard which is designed to better permeate materials that prevent signal propagation, like thick walls and metallic objects. Zigbee has two types of device. Reliably powered devices act as the backbone of the network and are actively used to maintain routes across the network. Less reliably powered devices only wake to communicate data. This division of labour results in great energy efficiency. Its low energy consumption for low data rate peripherals has made Zigbee the wireless communication protocol of choice for many IoT solutions.

Thread [278] also builds upon the 802.15.4 physical standard but is designed to simplify the translation of packets to IP compatible packets. To do this, Thread builds upon 6LowPAN, a compressed form of IPv6 designed for low power wireless protocols. Devices must maintain individual routing tables to efficiently route packets across the network. As Thread is built on 802.15.4, there is a division of labour between devices similar to Zigbee.

Long range

Long range protocols are used to network devices across huge distances, often greater than a kilometer. This makes them ideal for large scale IoT deployments. Greater transmission distance typically demands more power consumption however.

LoraWAN [92] is designed to operate over huge distances whilst efficiently consuming power. Instead of increasing the transmission power to reach longer distances, LoRA devices *transmit for longer*, scaling the data rate with transmission range—the shorter the distance, the faster the transmission. LoRA has two device roles. Gateway devices provide routing

information to standard devices so that data correctly reaches its destination. Standard devices are simply participants in the network and are divided into three classes. Class A devices receive only after transmitting data; Class B devices schedule receive windows using beacons from LoRA gateways; and Class C devices always receive unless transmitting. LoRA leaves device class selection to the developer, and device power consumption changes depending on the chosen class. Its low power and huge range make LoRA an ideal protocol for large scale sensor deployments.

Long-term Evolution (LTE) [294]—also known as *cellular*—enables long-range communication between mobile devices, commonly mobile phones. In an LTE network, mobile devices periodically communicate with nearby cell towers. Cell towers track which devices have recently checked-in and data communications from IP-based networks (and vice versa). LTE transmissions happen concurrently using techniques like Code Division Multiple Access (CDMA) [241] and transmitter power scales inversely with proximity to a cell tower—the closer a transmitter is to a cell tower, the less power it has to use to transmit a message. Energy consumption increases if a device is moving quickly between cell towers as requests may need to be retried. To provide additional signal resilience in this case, devices often maintain multiple connections to towers on different generations of cellular (2G, 3G, 4G). Though LTE is scalable to hundreds of nodes, its power consumption reduces its practicality for sensor deployments.

2.5.3 Analysis

This section has discussed popular wireless networking protocols, how they are used, and how they fundamentally work. From prior discussion we can make the following observations:

Battery powered devices require energy efficient protocols. Prior discussion presents a clear relationship between battery power application and protocol selection. Devices that are embedded and distributed across environments on battery, use low data rate energy efficient protocols like Zigbee. Mobile devices like smart phones also require careful energy consumption, but they typically cannot afford low data rates due to media rich content.

Citizen developers require intuitive experiences. The simple user experience of WiFi means that it is used by technical novices to connect devices to the Internet. Only a wireless router is required to connect devices to the Internet, and it directly supports higher level protocols like IP.

Ad-hoc networking protocols give great flexibility. As well as offering greater energy efficiency, ad-hoc networking protocols offer more flexibility than other types of protocol. Networks can be dynamically extended, and devices dynamically added. This is good for environments with high levels of interference like the home.

2.6 Summary

This section summarise observations made across this chapter to motivate areas for further study. We break discussion into three areas: programming (Section 2.6.1), hardware composition (Section 2.6.2), and wireless networking (Section 2.6.3). We conclude the section by extracting the needs of the citizen developer and deriving the four Guiding Principles (GP) applied throughout this thesis (Section 2.6.4).

2.6.1 Programming

Due to their complex syntax and manual memory management, low-level text-based programming languages like C/C++ are hard to use. Ease of use is traded for unparalleled memory and processor efficiency and it is therefore no surprise that these languages reign supreme in the resource constrained world of the microcontroller. Abstraction layers like Arduino seek to make C/C++-based microcontroller programming more accessible, but they still prove inaccessible to those with little technical expertise.

Technical novices have demonstrated a preference towards higher level languages. Higher level languages aim to make programming easier through adding abstractions, automatically managing memory, and simplifying syntax. Event-based visual programming languages, like Scratch, have been shown to be particularly more intuitive for those with little technical expertise. The simple connection of pre-defined blocks allows users to focus on the structure of programs rather than the syntax. Installation free programming environments for these languages has shown to make programming even more accessible.

Though the abstractions introduced by higher-level languages greatly simplify programming, they come at the cost of processor and memory efficiency. It is for this reason that they are not widely supported by resource constrained microcontrollers. Without these abstractions however, physical computing remains inaccessible to many citizen developers.

Are there any programming languages/environments for microcontrollers that support installation-free (P3), event-based (P2), visual programming (P1)? Do any such

environments support these features without compromising memory and processor efficiency?

2.6.2 Hardware composition

USB makes hardware composition intuitive for all. It is designed for dynamic connectivity and detects devices as they are connected. It also abstracts hardware as a set of standard software interfaces allowing USB devices of the same type (i.e. a keyboard) to take the place of one another without any installation. These properties, combined with its low infrastructure (single cable) approach to wiring, makes hardware composition intuitive to those with little technical expertise.

The same cannot be said of prototyping physical computing devices however. Here, protocols like I2C and SPI are used to connect microcontrollers to peripherals. Unlike USB, these protocols do not allow for dynamic connectivity and connections are made using multiple wires, requiring a basic knowledge of electronics. Hardware abstraction is also not supported, and specialised drivers are required for each peripheral even if peripherals have the same high level functionality (i.e. an accelerometer).

As shown in the world of consumer devices, these properties are important for technically inexperienced citizens. They are even more important for physical computing, where hardware is increasingly being composed iteratively and dynamically. Intuitive composition does come at a monetary cost, and more dynamic protocols like USB are not widely supported on low cost microcontrollers. Without these features however, hardware composition remains inaccessible to many citizen developers.

Are there any low-infrastructure wired protocols for hardware composition that support dynamic connectivity (HC1), device discovery (HC2), and hardware abstraction (HC3)? Are any such protocols as widely supported by microcontrollers as I2C or SPI?

2.6.3 Wireless networking

WiFi makes wireless networking intuitive to all. It is designed to support high throughput connections between interactive consumer devices and all that is required to create and connect to a WiFi network is a router, a network name, and a password. This low configuration, low infrastructure approach makes wireless networking intuitive to those with little technical expertise.

Because of its ease of use, WiFi is similarly used to network physical computing devices. Physical computing devices however are typically powered by battery and widely deployed across environments. Here, energy efficiency and scalability quickly become a more important concern than throughput. The use of WiFi therefore places unnecessary constraints on applications through its proximal operation and its preference for throughput over energy efficiency.

Ad-hoc networking protocols offer a scalable and energy efficient alternative to WiFi. However, these protocols are not widely used for networking physical computing devices because they require configuration and infrastructure to operate. This complexity caused Particle—a company that specialises in the IoT—to discontinue their ad-hoc networking offering. Without intuitive and energy efficient ad-hoc networking however, many citizen developers cannot flexibly network physical computing devices.

Are there any wireless ad-hoc networking protocols that require no configuration (WN1) and no infrastructure (WN2) to operate? Are any such protocols able to support interactive applications (WN3) without sacrificing energy efficiency?

2.6.4 Needs of the citizen developer

From the analysis sections throughout this chapter we derive the following Guiding Principles (GP) that should be observed during the creation of any new technologies:

GP1 *Intuitive*: Visual programming languages that vastly simplify programming, wired protocols that make hardware composition dynamic, and wireless protocols that make networking devices simpler are all used in preference of technologies that may offer greater efficiency. New technologies must be easy to use for citizen developers with little experience of hardware and software development.

GP2 *General*: Fashion designers use physical computing to build interactive garments for the runway, teachers use physical computing to educate future generations, and makers use physical computing to build new and unusual inventions. Citizen developers also use a variety of different embedded development boards, each with different microcontrollers and varying capabilities. New technologies must be suitable for different application domains and the spectrum of embedded hardware.

GP3 *Extensible*: Visual programming languages scaffolds those with little expertise, but this quickly becomes a constraint when citizen developers reach a certain level of

competency. Equally though, technologies that allow for too much extensibility (i.e. wired protocols) present too much of a barrier to those with little expertise. New technologies must support the easy addition of new functionality and interoperate with existing tools and standards.

GP4 *Efficient*: Microcontrollers demand programming languages efficiently use memory and processor cycles, and battery powered devices demand that wireless protocols appropriately manage their energy efficiency. New technologies must be highly efficient, but not at the great expense of GP1–GP3.

Chapter 3

Related Work

In Chapter 2, we identified three areas that currently prevent the democratisation of physical computing: *programming*, *hardware composition*, and *wireless networking*. As a result of extensive discussion, we identified attributes of existing technologies in each area that may result in more intuitive technologies if applied to physical computing. Based upon these attributes, we also posed three further investigatory questions (IQs):

IQ1 Are there any programming languages/environments for microcontrollers that support installation-free (P3), event-based (P2), visual programming (P1)? Do any such environments support these features without compromising memory and processor efficiency?

IQ2 Are there any low-infrastructure wired protocols for hardware composition that support dynamic connectivity (HC1), device discovery (HC2), and hardware abstraction (HC3)? Are any such protocols as widely supported by microcontrollers as I2C or SPI?

IQ3 Are there any wireless ad-hoc networking protocols that require no configuration (WN1) and no infrastructure (WN2) to operate? Are any such protocols able to support interactive applications (WN3) without sacrificing energy efficiency?

This chapter seeks to provide answers to IQ1–IQ3 and begins by discussing programming (Section 3.1, IQ1), then progresses onto hardware composition (Section 3.2, IQ2), and finishes by discussing wireless networking (Section 3.3, IQ3). We summarise our findings for each investigatory question in Section 3.4

3.1 Programming

This section explores programming languages and environments specifically designed for microcontrollers. We cover compiled programming languages (Section 3.1.1), interpreted programming languages (Section 3.1.2), and visual programming languages (Section 3.1.3). In Section 3.1.4, we provide an analysis of the space, and categorise languages and environments according to their *efficiency*, and by the number of design requirements (P1—P3) from IQ1 they enable.

3.1.1 Compiled programming languages

For many decades, compiled programming languages like C and C++ have reigned supreme in the resource constrained world of physical computing devices. Due to their memory and power efficiency, they remain the staple of embedded systems development to this day. Compiled programming languages are efficient for two reasons. The first is the compilation of programs to *processor specific binary instructions*. The use of binary instructions lead to more efficient use of program memory (flash) and processor cycles. The latter has implications for energy use and real-time behaviour: increased processor cycles lead to greater energy consumption and application latency. The second efficiency gain comes during application development. Manual Random Access Memory (RAM) management causes developers to optimise applications to use a little memory as possible. Complex, but unambiguous, syntax allows compilers to reduce the number of binary instructions. Efficiency therefore comes at the cost of simple application development.

There are many existing languages and environments that seek to make it easier to develop applications in compiled programming languages. We group work into the following categories: C/C++ runtime environments, C/C++ Hardware Abstraction Layers (HALs), compiled high-level languages, and interoperable binary standards.

C/C++ Real Time Operating Systems (RTOSs)

Applications for embedded devices require real-time behaviour and the careful management of competing concurrent tasks. Tasks usually fit into two categories. System-level tasks that manage networking protocols like WiFi and USB, and application-level tasks that respond to data received from system level tasks. System-level tasks are usually high priority tasks and preempt application-level tasks.

	Tiny OS	Contiki	mbed OS	Mynewt	FreeRTOS	RIOT	Zephyr
RAM consumption (kB)	1	2	8	3	3	1.5	8
Flash consumption (kB)	4	30	16	8	10	5	14
Scheduling model	Non-preemptive	Non-preemptive	Pre-emptive	Pre-emptive	Pre-emptive	Pre-emptive	Multiple
Real-time							
Modular							
Language Support	NesC	C	C/C++	C/C++	C/C++	C/C++	C/C++
Purpose	WSNs	WSNs	IoT	IoT	IoT	IoT	IoT

Fully supported | Partially supported | Not supported

Table 3.1 Comparison of different operating systems for low resource physical computing devices. Operating systems referenced in order of appearance [204, 147, 97, 88, 109, 104, 89, 130]

Real Time Operating Systems (RTOSs) offer real-time behaviour and provide a framework for efficient task management. RTOSs are written in compiled programming languages and are designed for memory efficiency. Existing RTOSs for low resource physical computing devices are encapsulated in Table 3.1. Common properties of each are listed for comparison. These properties must be carefully considered by application developers.

One of the primary considerations is the RAM and flash consumption of an RTOS. As RAM is often more scarce than flash memory, all of the operating systems in the table consume more flash than they do RAM.

The table also shows that each RTOS is designed for a specific purpose. End purpose informs the design of RTOSs. Tiny OS [204] and Contiki [147], are designed for resource constrained wireless sensors and as a result consume a small amount of RAM. mbed OS [97] and Zephyr [89] are designed for resource-rich and highly capable cortex-m processors where RAM consumption is not as much of a concern. The figures in the table are extracted from highly tuned builds, it is likely that the default configurations for each RTOS will consume more flash and RAM.

Developers also consider the scheduling model of an RTOS. The scheduling model has implications on the ease of writing applications, handling concurrent access to hardware resources, and the amount of memory consumed by individual threads. As can be seen in Table 3.1, for constrained sensing devices, operating systems opt for non-preemptive scheduling to keep memory consumption to a minimum. Non-preemption, in the case of

```
#include "mbed.h"
#include "platform/mbed_thread.h"

int main()
{
    DigitalOut led(LED1);
    while (true) {
        led = !led;
        thread_sleep_for(BLINKING_
                RATE_MS);
    }
}
```

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN,HIGH);
  delay(500);
  digitalWrite(LED_BUILTIN, LOW);
  delay(500);
}
```

Fig. 3.1 Code to blink an led in mbed os "bare metal" (left) and the Arduino abstraction layer (right)

Contiki, reduces memory consumption by ensuring tasks are executed to completion before the next task is scheduled. This simplifies scheduling by maintaining just a single, shallow stack.

For operating systems designed for the IoT, a preemptive scheduling model is more common. This is due to the need to perform critical system/networking operations as well as concurrently execute user applications. The use of preemptive scheduling not only comes with greater memory overhead, but also comes with more development complexity. Mutexes, semaphores, and locks are required to handle shared access to software and hardware resources.

The realtime-ness of an RTOS is also an important consideration for developers. Lack of real-time capability renders an RTOS useless for many applications. Similarly, the modularity of an operating system is important for flash-constrained devices. Modularity allows unneeded software features to be omitted from the final binary. The language an RTOS is built on also bears some consideration. RTOSs built on familiar languages are likely to be used in preference of others.

C/C++ Hardware Abstraction Layers (HALs)

The multi-tasking and scheduling primitives of RTOSs create barriers for citizen developers. Simpler *Hardware Abstraction Layers (HALs)* lower the barrier to entry to compiled languages like C/C++. mbedos [97] currently contains one such simpler abstraction layer. Using the 'bare metal' profile, RTOS features are removed. Instead users are given a single thread to operate in and access to simpler standalone drivers like I2C and DigitalOut. Figure 3.1, left, shows a simple program that uses the DigitalOut class to toggle an LED.

Embracing a similar approach, the Arduino [107] abstraction layer gives users a subset of simple C functions to create programs for physical computing devices . An equivalent code sample for Arduino is shown in Figure 3.1, right. As can be seen, rather than use object-oriented classes, Arduino uses function calls and constants to control embedded devices. Code is also split into setup and loop calls, separating configuration code that is called once at the beginning of a program, from looping code, that is repeatedly called throughout the lifetime of an application. These design decision are born from a desire for simplicity whilst retaining efficiency. Citizen developers do not need to understand object orientation or looping constructs to build applications with Arduino.

With hardware abstraction layers, embedded devices are presented with uniformity. For the majority of applications, such uniformity is beneficial. However, not all embedded devices are created equal. Generalised abstractions do not support unique hardware peripherals and register configurations, preventing certain applications from being built by citizen developers.

Compiled higher-level languages

As an alternative to compiling low-level programming languages like C/C++, researchers have looked at compiling higher-level languages like Java to native machine code instructions [282, 127, 100, 112, 253, 105, 194]. These approaches generally map higher-level languages onto runtime environments written in C/C++. Runtime environments are either custom built, or built atop an RTOS. Applications written in higher level languages are compiled by traditional embedded toolchains to produce a binary. Compared to fully interpreted languages, compilation of higher level languages leads to quicker execution and lower RAM consumption. However, using fairly static low-level languages to support highly dynamic higher-level languages, means that not all language features can be implemented.

New higher-level languages, like Rust [215], are designed to provide higher level language features at the speed and efficiency of more traditional compiled languages. Rust gives embedded developers guarantees of memory safety, safe resource scheduling, extensibility, and object-oriented reflection, easing the process of embedded development. To support these features, Rust makes use of a modern compiler technology called Lower-level Virtual Machine (LLVM) [196]. LLVM virtualises the compilation process by turning language constructs into an Intermediate Representation (IR). By connecting a processor backend, IR is then transformed into efficient binary instructions. Rust has been shown to produce highly efficient machine code that nears the efficiency of native C/C++ in benchmarking tests [37].

Binary standards

Many of the runtime environments and programming languages above support dynamic code loading. Dynamic code loading allows developers to add, modify and change program execution without reprogramming the entire application. Code is transferred via various wired and wireless protocols. Once again, for reasons of efficiency, code is represented as binary instructions.

Application Binary Interfaces (ABIs) standardise the binary representation of code to improve interoperability between different runtime environments and languages. ABIs embed Application Programming Interface (API) metadata and binary offsets for code execution into each binary file. Languages/runtimes can then process received binary files, infer function calls, and subsequently execute them using simplex C function pointers. Many ABI standards exist, the most popular are the Component Object Model (COM) [165, 42], OpenCOM [132], and FRACTAL [121].

3.1.2 Interpreted programming languages

Higher-level interpreted languages aim to make software development more intuitive, familiar, and less error prone. Programs written in an interpreted programming language either interpreted from program text, or compiled to more efficient byte code. Dedicated programs, known as *interpreters*, interpret byte code. Interpreters are written in compiled languages like C/C++ and convert byte code into binary instructions for processor execution. Programs written in higher level languages can therefore run on any computing device with the appropriate interpreter installed.

An object model usually underpins higher-level languages. Object models allow developers to model key application components as well-defined classes. Classes define methods or functions, typically called APIs, that govern interactions with objects. Interaction typically only affect the object itself and objects are typed so that languages know the API calls that can be made. Object inheritance allows a class to extend and override the interfaces of another.

Enforcement of object typing varies from language to language. Strongly typed languages check API usage before invocation, whereas weakly typed languages invoke APIs without checking compatibility. Statically typed languages use compiler technologies to guarantee API calls are compatible before execution. Dynamically typed languages perform no such step, leaving users to encounter runtime errors.

Most higher level languages support reflection and polymorphism. These features allow applications to perform runtime introspection to *dynamically* infer object types. Applications written in higher level languages are therefore more supportive of dynamic applications.

With the more dynamic offerings of higher-level languages, languages originally designed for personal computers are now being ported to embedded devices. The remainder of this section summarises higher level languages for embedded devices. Languages are grouped by whether they support dynamic or static typing.

Dynamically typed

Lua [179] is a lightweight multi-paradigm higher level language primarily designed for use with embedded systems. The language itself has very few built in features, supporting just traditional control flows, functions, and tables. As a result Lua is portable and fast.

Object orientation is supported through prototypes. Prototypes are functions that accept a context variable with parameters that change or modify context. Context is a table, usually referred to as *self*. Prototype functions are prefixed with a name but there are no language constructs to group prototypes into classes.

Lua is highly extensible, and as a testament to its extensibility, true object oriented programming can be added to lua via an extension. The extension makes use of *fallbacks* which are deep system calls that are invoked when a runtime error condition occurs. Extensions can override fallbacks to provide alternative behaviours. In this case, object orientation.

Lua programs are converted into byte code for more efficient execution. Lua makes further efficiency gains from its interpreter implementation. Lua's interpreter is register-based (as opposed to stack-based) and aligns with the architecture of many embedded systems. Interpreting byte code is therefore a simple translation operation.

Python [281, 248] is a highly popular general purpose programming language designed to make programming more intuitive. Instead of complex semicolons, braces, and brackets, Python uses whitespace to structure code. Highly specialised packages provide simple APIs to hide complex algorithms beyond the capability of novices. Python also supports object orientation, allowing more advanced applications to be created. This combination of features makes Python an ideal way to learn computer science concepts.

Though it was written for use on personal computers, because of its popularity and ease of use, Python has been co-opted as a language for embedded systems development. MicroPython [161] and CircuitPython [11] (Figure 3.2, left) are widely used Python implementations designed for use with low-resource embedded devices. Because of device constraints, these implementations only support a subset of the full Python language.

```
import time
import board
from digitalio import
    DigitalInOut, Direction, Pull

led = DigitalInOut(board.D13)
led.direction = Direction.OUTPUT

while True:
  led.value = !led.value
  time.sleep(0.5)
```

```
var on = false;
function toggle() {
  on = !on;
  digitalWrite(LED1, on);
}
setInterval(toggle, 500);
```

Fig. 3.2 Code to blink an LED in CircuitPython (left) and Espruino (right)

MicroPython and CircuitPython only interpret program text. Program text is interpreted in two ways. The first is through an interactive *Read Eval Print Loop (REPL)*, an interactive interpreter that reads program text, evaluates and executes corresponding commands, prints the result, and loops to read more program text. Developers connect to interactive REPLs via a serial terminal and transfer program text via the terminal for the embedded device to execute. This method of interpretation is great for API experimentation.

The second way program text is interpreted is through operating the REPL *without interactivity*. Program text can be transferred once again through a serial terminal, through dedicated applications, or more universally through a simple file write operation over USB. CircuitPython uses USB (via the embedded device) to present a flash drive to developers. Developers can then update program text stored on the embedded device by editing and saving 'code.py' to the flash drive. The REPL subsequently interprets program text from flash memory.

Solely using a REPL for program execution is extremely costly in terms of RAM consumption and processor efficiency. As a result some real-time applications cannot be created in MicroPython or CircuitPython. For example, MicroPython's flash and RAM consumption on the BBC micro:bit prevents the creation of Bluetooth applications.

JavaScript [152] is a popular language primarily used for web development. Recently the use of JavaScript has moved beyond the browser, and it is now used for desktop and mobile application development as well. Developers can now write JavaScript applications to manipulate local file systems, interact with USB devices, develop scalable servers for websites, and create first-class applications for mobile devices. JavaScript uses brackets, semi-colons, and braces to structure code. Despite its more advanced syntax, JavaScript is a popular beginner programming language.

Again, due to its popularity and ease of use, JavaScript has been co-opted as a language for programming embedded devices. A number of interpreters now support the execution of JavaScript programs on embedded devices. Espruino [289] (Figure 3.2, right) is one such interpreter. Espruino interprets program text and supports a subset of the full JavaScript language. Once again, program text is interpreted from an interactive REPL or from text stored in program memory. The Espruino web Integrated Development Environment (IDE) can transfer text to devices via serial or BLE. The use of BLE allows Espruino devices to be programmed from a variety of devices, including smartphones and tablets. Because interpreting program text is slow, the web IDE can compile JavaScript functions into binary code. Functions to be compiled are marked with the compile keyword and these functions are sent to an external compile service for compilation to binary. Due to incompatibilities between the JavaScript interpreter and raw binary code, not all JavaScript functions can be compiled.

Duktape [14] is a slightly different interpreter that uses a foreign function interface to efficiently bind JavaScript functions to C/C++ functions. Mapping JavaScript functions to C/C++ functions leads to far more efficient, real-time applications. Duktape is designed to ease the process of porting JavaScript to embedded devices than for use by novices. Using the Duktape C/C++ APIs, Developers can write C/C++ applications that take JavaScript program text and interprets it via the Duktape virtual machine. Despite its claims of efficiency, Duktape consumes 160 kB of flash and 64 kB of RAM with its default configuration.

JerryScript [160] is another JavaScript interpreter. Instead of interpreting program text directly, JerryScript converts JavaScript to byte code for more efficient execution. Program text is converted into byte code once and stored in RAM for later execution if required. Like Duktape, JerryScript gives developers a C/C++ API to execute JavaScript programs. Using these APIs developers can build standalone applications or interactive REPLs. JerryScript applications support the inclusion of JavaScript modules. Modularity lets developers better organise their code. JerryScript reports that it can execute on embedded devices with 200 kB of flash and 64 kB of RAM.

Statically typed

Java [98, 205] is a statically-typed general purpose programming language intended to run anywhere. Java programs are compiled into byte code before execution. The byte code compiler guarantees variable and interface compatibility reducing the likelihood of runtime errors. Byte code can be executed universally by any Java Virtual Machine (JVM) that follows the language specification. Though Java was originally designed to run on embedded

devices, its expanding feature set has imposed memory demands that have outgrown many embedded devices. As a result, interpreters for embedded Java execution implement a subset of the JVM specification.

Squawk [260] applies a number of optimisations to Java byte code to make them suitable for embedded devices. The first reduces the size of class files. Java class files contain the byte code to be executed by the JVM. Squawk translates class files into a condensed representation suitable for resource constrained devices. The second reduces complexity of code for easier garbage collection. Garbage collectors automatically free memory after it is deemed no longer in use. Determining when memory is no longer in use is a challenging problem and requires large algorithms for complex code. Squawk pre-processes byte code and produces semantically equivalent, less complex to collect byte code. The third optimisation resolves class references before execution. Class references resolution takes a negligible amount of time on personal computers, but on slower microcontrollers, this process takes far longer. Squawk resolves class references before the programs are transferred to the embedded device speeding up execution time. Despite these RAM, flash, and processor optimisations, powerful embedded devices are required to execute Squawk. It is reported that Squawk consumes 270 kB of flash and 80 kB of RAM [259].

The .NET Framework is a collection of APIs for the Windows operating system. The framework supports many different programming languages, including statically types languages such as C# [172] and visual basic. Applications built using these languages are compiled to an intermediate assembly language for execution by the Common Language Runtime (CLR). The CLR takes intermediate assembly and converts it into native machine code using a Just In Time (JIT) compiler. Converted code is saved for faster future execution. The .NET Framework therefore allows code to be written in many different languages and environments, but run universally on any Windows device. Because the framework provides access to the entire Windows operating system, it is not well suited to resource constrained embedded devices.

The .NET Micro Framework (.NET MF) [217, 271, 272] reduces the size of the .NET Framework to make it compatible with resource constrained embedded devices. The .NET MF contains device drivers, GUI components, and network stacks as well as the CLR from the full framework. Underpinning the CLR is a hardware abstraction layer. The HAL implements a small number of functions and is called by the CLR when required. The CLR then interprets intermediate assembler and calls into the HAL to execute applications—that is to say programs are always interpreted rather than JIT compiled. According to Thompson, the .NET MF requires 512 kB of flash and 300 kB of RAM, excluding application code [272].

powerful processors are required to run the .NET MF. In .NET Gadgeteer [175], the .NET MF was used to help citizen developers build physical computing devices.

OCaml [202] is an object oriented statically typed language, influenced by Caml. OCaml was originally designed for automated theorem proving and primarily supports interpreting byte code (though OCaml programs can be compiled to native binaries). OCaml programs are verified before execution. The verification process prevents common runtime errors like dereferencing null pointers. Its size means it has not seen wide adoption as a programming language for embedded devices.

Projects like OCaPIC [283] bring OCaml to resource constrained microcontrollers. OCaPIC converts OCaml byte code to a condensed representation, reducing file size by up to 75%. OCaPIC also parses files and removes and redundant or complex mathematical operations. OCaPIC enables the execution of OCaml on a PIC18f microcontroller with 128 kB flash and 4 kB RAM.

3.1.3 Visual programming languages

Visual programming languages allow users to focus on the structure and functionality of code, rather than the syntax. Compared to text-based programming languages, visual programming languages lower the barrier to entry to programming.

Existing visual programming languages can be grouped into two broad categories, flow-based programming languages, and block-based programming languages [214]. *Flow-based* [222] programming languages inform let users compose programs by connecting pre-defined segments of code in a sequential order. The output of the previous segment feeds into the next. *Block-based* programming languages on the other hand let users plug pre-defined blocks of code together like Lego bricks. Blocks are abstracted as a sequence of disjointed operations. The following sections group existing work by these two categories.

Flow-based

FLOWOL (Figure 3.3, left) is a popular flow-based programming language used in schools. It uses interactive scenes, called mimics, to motivate students to solve problems by programming. Mimics usually represent scenarios that require programmatic control, like traffic lights or draw bridges. By creating applications, students can cause mimics to change i.e. by changing the colour sequence of traffic lights. Programs are structured using pre-defined flow chart segments. Each segment feeds into the next, and if statements are modelled as yes / no queries. More importantly, FLOWOL can also be used to program embedded

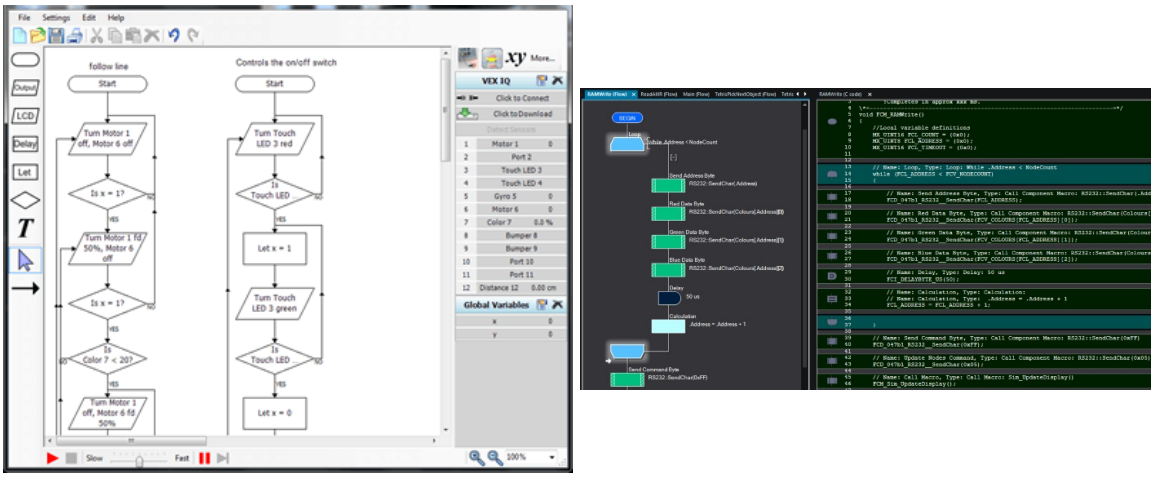

Fig. 3.3 The FLOWOL 4 programming environment (left) and the Arduino abstraction layer (right)

devices like the Arduino Uno, VEX IQ, VEX Cortex, and PICAXE microcontrollers. This gives citizen developers an easy introduction to microcontroller programming. FLOWOL appears to be resource efficient as it is capable of executing on resource constrained PICAXE microcontrollers.

Flowcode (Figure 3.3, right) is another flow-based programming language. It can be used to program ARM, AVR, and PIC microcontrollers; all Arduino devices are also supported. As in FLOWOL, Flowcode lets users combine pre-defined segments to execute code. Segments map onto C/C++ and using a dedicated IDE, users can peer behind the flow chart abstraction to view the underlying C/C++ code. For Arduino targets, Flowcode segments map directly to the Arduino HAL. Here, peering behind the abstraction allows developers to learn Arduino APIs. A compiler converts Flowcode applications to binary files for execution on embedded devices. Flowcode also supports pseudocode and block-based programming to align with other novice programming environments.

NodeRED [230] is a flow-based programming environment but not in the traditional sense—NodeRED acts as a user application for the Raspberry Pi operating system. Like prior languages, NodeRED lets users combine pre-defined segments of code together to create applications. However, instead of low-level instructions, users connect high-level data sources and sinks to automate IoT data flows. Data sources can be any web-based or local resource and can be combined with an arbitrary number of processing nodes. Sink nodes absorb data flows, usually as databases or as presentation layers like HTML web pages. Using the flow-based programming model, NodeRED removes the complexity of connecting and piping data between disparate web resources and interfaces. However, not all complexity

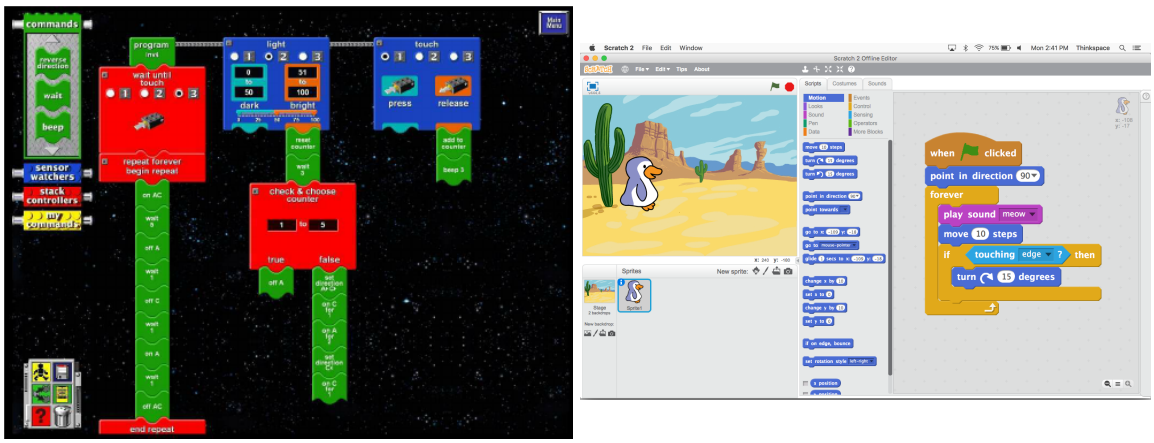

Fig. 3.4 Code samples from Lego Mindstorms (left) and Scratch (right)

can be abstracted away—users still have to write text-based code for data processing and presentation nodes. Despite this, NodeRED vastly simplifies processing and automating IoT data flows.

Block-based

Lego Mindstorms [189] (Figure 3.4, left) defined block-based programming as we know it today. Lego Mindstorms was a kit that featured the first block-based software programming environment, a re-programmable command "brick", and a number of easy-to-connect sensors. By combining the command brick with sensors and block-based programming, citizen developers could easily build devices that react and respond to the real world (see Section 2.1.3). The block-based programming environment let users compose programs by plugging together segments of code like physical lego bricks. By adopting Lego as its construction material, Mindstorms enabled the animation of custom lego creations. Lego Mindstorms EV3 [25] is a modernised versions of the original Lego Mindstorms kit with updated hardware. Whilst Mindstorms is aimed at key stage 3 students, alternative products like Spike Prime [16] and WeDo [43] are aimed at key stage 2 and 1 students respectively.

Scratch [212, 245] is an evolution of the Lego Mindstorms visual programming language. Scratch programs are composed of blocks that are pieced together like Lego bricks. Blocks are coloured based up on their functionality and have different connector types and shapes to prevent incompatible blocks from being connected together. Using the Scratch programming environment, users can create physics-based games consisting of customisable animated sprites and graphics. Events are a key concept in Scratch and are used to trigger blocks of code asynchronously. An example of event-based Scratch programming is shown in Figure 3.4,

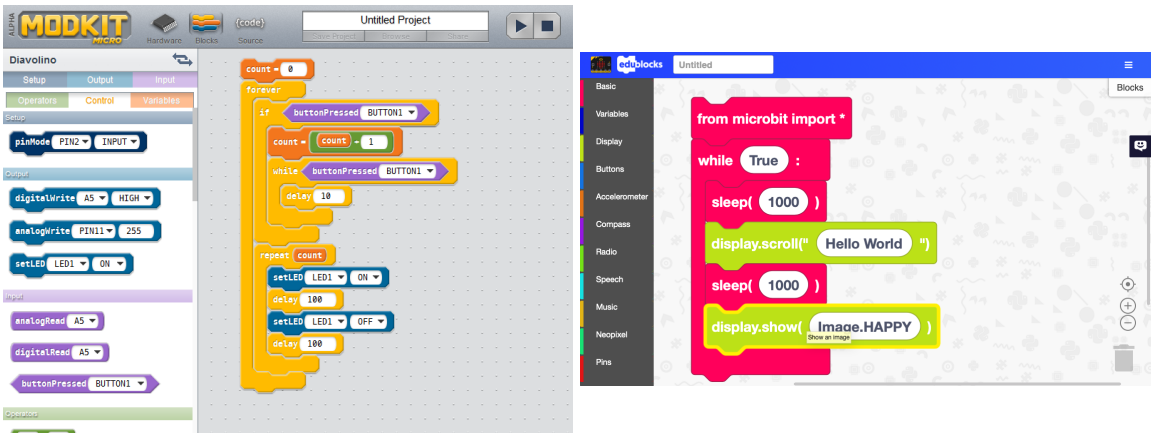

Fig. 3.5 Code samples from Modkit (left) and EduBlocks (right)

right, where code is executed when the green flag is clicked. Scratch, however, cannot be used to directly program physical computing devices. Instead, physical computing devices act as inputs to Scratch applications. Interactive devices tether to Scratch via USB or BLE and emit events when devices detect interactions. Scratch extensions add the custom blocks required to communicate with physical computing devices. Scratch—and especially event-based programming—has been shown to be successful in teaching novices core, translatable programming concepts in an engaging way [290, 276].

Kodu [208] is an educational 3D games world where users can write block-based programs to cause a hero to move about the world. Kodu, like Scratch, can also tether to physical computing devices. Interactive devices can then act as inputs to the 3D game world, allowing the real world to interact with the virtual.

Modkit [220] (Figure 3.5, left), Splish [181], and ArduBlock [87] are visual programming environments for Arduino devices. These environments wrap Arduino HAL APIs with colourful blocks. The primary benefit to this approach is that users do not have to learn low-level programming languages to create applications with the same efficiency as C/C++. A secondary benefit is that by sharing a common device vocabulary (led, pin) and programming language vocabulary (while, if), it is possible for users to eventually transition to C/C++. Simply wrapping C/C++ APIs is not enough for citizen developers. Citizen developers require simpler event-driven and asynchronous programming [275, 213].

EduBlocks [207] (Figure 3.5, right) uses blocks to present MicroPython APIs in a more friendly way. Though Python was designed as a novice friendly language, programming constructs and use of whitespace to dictate program structure can be daunting to users. EduBlocks removes both whitespace and program constructs from programs creating a simpler experience for users. Wrapping Python code in blocks also makes future transitions

from block-based to text-based programming easier. Built-in editor transitions from blocks to program text makes this transition even easier. Programs written in EduBlocks generate program text for interpretation by a MicroPython interpreter running on a physical computing device. EduBlocks currently supports all CircuitPython devices, the Raspberry Pi, and the BBC micro:bit.

Microblocks [125] uses an interpreter to bring asynchronous block-based programming to physical computing devices. Devices are connected over USB to the Microblocks desktop programming environment. Code running on the device is updated as users compose programs, leading to an extremely responsive development experience. Microblocks applications can also run independently of the IDE, allowing devices to be embedded in custom creations. As learned from prior sections, interpreted languages consume significantly more energy on battery than other modes of program execution. To execute on microcontrollers, microblocks requires 50 kB of flash and 12 kB of RAM.

With advances in web technologies, code editors are moving to the web. Block-based code editors are moving to the web too. Blockly [157, 156] brings block-based programming to web. Blockly provides a reusable framework that lets developers build domain specific block-based languages. A number of programming environments for physical computing devices have been created using Blockly. Scratch 3.0, EduBlocks, Ozoblockly [231, 154], Open Roberta [246, 180], and CodeBug [133] and many more. Combined, block-based programming and web technologies significantly lower the barrier to entry for programming physical computing devices. They require no installation and programs can be created from any device with a web browser.

3.1.4 Analysis

Programming languages are central to building a physical computing device. Citizen developers use programming languages create applications that define possible interactions between users and devices. Applications are converted into executable instructions and are subsequently executed by microcontrollers.

Compiled languages directly compile text-based applications to processor and spatially efficient *machine code*. Dedicated compiler tool chains are required to handle the complexity of producing machine code and because machine code is processor specific, applications need to be re-compiled for each different processor architecture.

Compiled languages are often referred to as low-level languages as they generally offer few built-in features. This makes compiled programming languages highly memory and

processor efficient but substantially harder to use. Though there are new compiled languages that offer the features higher-level languages at the expense of some efficiency, much of the existing work seeks to abstract existing low-level programming languages by creating specialised environments. These environments vary from complex real-time operating systems, to simple Hardware Abstraction Layers (HALs).

Interpreted languages use raw program text or byte code to represent programs. A program executing on the microcontroller, known as an interpreter, converts raw *program text* or pre-compiled *byte code* to machine code. Representing code in this way brings great portability to applications by driving implementation complexity to the interpreter. Converting byte code or program text to machine code, however, results in more *processing overhead* per line of code. Greater processing overhead leads to more energy consumption and less real-time performance. Both are important in embedded and interactive applications.

Interpreted languages are generally feature-rich. Automatic memory management lets developers create and dispose of objects without care. Powerful built-in data types and class systems let developers program with ease. Reflection and polymorphism let developers inspect objects at run time for super dynamic applications. Any language feature increases interpreter complexity, consuming more *flash* and *RAM*.

Interpreters are not the only source of increased flash and RAM consumption for interpreted languages. Type systems, used to give developers guarantees of API compatibility, also increase consumption. *Statically typed* languages perform type checking during a pre-compilation step and store API metadata in byte code. Flash consumption is therefore generally higher for statically typed languages. *Dynamically typed* languages, on the other hand, support a small number of types and will try to coerce variables to the expected type where possible. Type checking is performed at runtime whenever an object or API is used and a run time error is generated when this is not possible. A smaller number of types means that dynamically typed languages are more flash efficient.

Citizen developers are not typically professional software developers and *visual programming languages* have been shown to be easier to use than text-based programming languages [275, 213]. By using graphics, rather than text, visual programming languages let citizen developers focus on code structure rather than syntax. Visual programming languages that also support *event-based programming* have also been shown to be more intuitive for novices. Event-based programming models require less usage of complex looping structures simplifying program creation. Web-based visual programming environments further democratise access to programming. Visual programming languages for microcontrollers typically use high or low level languages as their foundation and therefore represent programs

Table 3.2 A sample of the programming languages and environments discussed in this section. Languages and environments are ordered by the design requirements (P1—P3) they enable. MakeCode and CODAL, one of the contributions of this thesis, is provided for comparison.

	Primary Domain	Supported Languages	Program Representation	Processor Efficiency	Flash Efficiency	RAM Efficiency	Visual programming (P1)	Event-based programming (P2)	Installation-free (P3)
Scratch	Education	Scratch	N/A	N/A	N/A	N/A	Yes	Yes	Yes
Microblocks	Education	Blockly	Byte code	Moderate	High	Moderate	Yes	Yes	No
OzoBlockly	Education	Blockly	Byte code	Moderate	–	–	Yes	No	Yes
EduBlocks	Education	Blockly, Python	Program text	Low	Moderate	Moderate	Yes	No	No
FHighcode	Education	Custom Blocks, Psuedocode, FHighs, C/C++	Machine Code	High	High	High	Yes	No	No
Modkit	Education	Blockly	Machine Code	High	High	High	Yes	No	No
micropython	Making	Python	Program text	Low	Moderate	Moderate	No	No	Yes
Arduino	Making	C/C++	Machine Code	High	High	High	No	No	Yes
Espruino	IoT	JavaScript	Program text	Low	Moderate	Moderate	No	Yes	Yes
mbed OS	IoT	C/C++	Machine Code	High	High	High	No	No	Yes
.NET MF	Professional	C#, Visual Basic	Byte code	Low	Low	Low	No	Yes	No
Squawk	Professional	Java	Byte code	Low	Low	Low	No	Yes	No
Rust	Professional	Rust	Machine Code	High	High	High	No	Yes	No
MakeCode and CODAL	Education	Blockly, TypeScript, C++	Machine Code	High	High	High	Yes	Yes	Yes

as program text, byte code, or machine code. One of the most popular visual programming environments, Scratch, has all the features citizen developers require, *except the ability to program physical computing devices*.

Table 3.2 summarises the programming environments and languages discussed in this section. We order languages and programming environments according to the design requirements (P1—P3) posed in IQ1. We consider low flash consumption as below 128 kB and high as above 256 kB. Low RAM consumption is less than 8 kB, and high is greater than 32 kB.

3.2 Hardware composition

This section explores wired protocols used for hardware composition. We start by covering existing and emerging wired protocols (Section 3.2.1) and we then analyse their use in tool kits designed for easier hardware composition (Section 3.2.2). Section 3.2.3 provides a deeper analysis, and categorises protocols according to the design requirements (H1—HC3) posed in IQ2.

3.2.1 Wired Protocols

Connecting peripherals to microcontrollers is a key component to building a physical computing device. Without peripheral sensors and actuators citizen developers would not be able to write interactive applications. Wired protocols are used to standardise communication interfaces between microcontrollers and peripherals. Each wired protocol is designed to support a different scenario and protocol implementation complexity correlates with its purpose.

Electrical conductors facilitate the communication between microcontrollers and peripherals. Conductors can be wires, copper PCB traces, or custom connectors and are used to form a communication line between microcontrollers and peripherals. Microcontrollers and peripherals modify the voltage level of the line to transmit data. A binary one is usually represented by a positive voltage, and a binary zero represented by absent or negative voltage. For digital communications, devices require a common ground reference to determine the logical line level.

Different protocols support different topologies. Point-to-point (1:1) protocols allow the efficient communication between two devices. One-to-many protocols allow one central (master) device to address and control many peripheral (slave) devices (1:N). And multi-point protocols allow multiple central devices to communicate with many peripherals (M:N).

For efficiency, protocols generally either support point-to-point or one-to-many topologies. Multi-point protocols enable the most dynamism, but are barely supported due to increased complexity.

Communication rate varies between wired protocols. Communicate rate is measured in baud, a scientific unit for the number of symbols that can be transmitted per second. The more symbols that can be communicated, the higher the throughput of the line. To determine the baud rate of the line, some protocols provide an additional clock line. Data can be decoded by correlating logical line level with pulses on the clock line. Protocols that require a separate clock signal are referred to as *synchronous*. Other protocols do not require a separate clock line. Instead, the baud rate of the line must be pre-determined and symbols are decoded by comparing the logical line level against an accurate clock source. Start and stop signals are used to delimit the start and end of a symbol. Protocols that do not require a separate clock signal are referred to as *asynchronous*.

For the remainder of this section, we categorise wired protocols by whether they operate asynchronously and synchronously. We also provide an further category dedicated to emerging wired protocols.

Asynchronous

RS232 [254] enables point-to-point (1:1), full-duplex, asynchronous communications between two microcontrollers. RS232 is synonymous with Universal Asynchronous Receiver Transmitter (UART) and is the simplest of all protocols to implement. Rather than packets, UART devices send a stream of bytes. Byte streams allow developers to define their own application-level protocols and are ideal for variadic data streams like debug output. Byte format on the wire can be configured depending on the application. The default format, 8-N-1, uses one start bit, 8 data bits, and 1 stop bit. Parity bits can be added at the cost of greater line utilisation. Transmission rate is configurable and standard ranges start at 9600 baud (symbols per second) and finish at 1 megabaud. As no clock line is used to dictate the transmission rate, devices must decode bytes *asynchronously* by sampling them as they are transmitted. This means that both byte format and communication rate must be known by devices in advance. UART can also operate in two modes. *Full-duplex* mode uses a dedicated line for transmission and reception, whereas *half-duplex* uses a single wire for both transmission and reception. Half-duplex mode reduces data throughput as only one device can transmit at a time.

RS422 [263] is fundamentally the same as RS232 but allows one way communication between a central device and up to 10 peripheral devices (1:N). RS422 extends communi-

cation range to 1,500 metres by using differential signalling, known as *differential drive*. Differential drive uses two physical wires to represent one communication line. Each physical wire is usually driven simultaneously to different logical levels to represent a one, and the opposite level combination to represent a zero. Differentially driven protocols do not require a common ground as relative changes in voltage are easily detected. As well as improving range, RS422 increases the maximum supported baud rate to 10 megabaud.

RS485 [86] expands RS422 to support bi-directional communications from peripherals (M:N). A maximum of 32 central devices and 32 peripherals can be supported. As it supports multi-central operation, RS485 has seen wide adoption for industrial control applications. However, there is no means by which devices can contend for control over the wire (*bus arbitration*). Instead error detection is punted to the application, forcing developers to include a Cyclic Redundancy Check (CRC) in byte streams.

The OneWire [103] protocol uses a single wire for communication. Devices are connected to the same physical medium to form a shared bus. A dedicated central device powers and initiates all activity with peripherals on the bus (1:N). When the bus is idle, the voltage of the bus is positive and reads as a logical one. Each device carries a capacitor that draws current when the bus is idle and peripherals are powered from stored electrical charge during bus communications.

On the wire, the central device drives bus low for 15 microseconds to represent a binary one and for 60 microseconds to represent a binary zero. For peripherals, the central device clocks each bit, signalling the start of a bit by driving the line low for 15 microseconds. To send a binary one, the peripheral does nothing and the line returns high. To send a binary zero, the peripheral continues to hold the line low until 60 microseconds have passed. The slow symbol rate gives more flexible protocol timings so devices can use cheap clocks to decode and transmit signals asynchronously. Slow symbol rate combined with the need to provide power means that OneWire can only achieve a data rate of 16.3 Kbps.

A central OneWire device can dynamically discover connected peripherals by initiating the enumeration protocol. The enumeration protocol causes all connected peripherals to send a 64-bit identifier which contains a 48-bit number that uniquely identifies the peripheral, an 8-bit device type, and an 8-bit crc. After finding a corresponding device type, central applications can then use the 48-bit identifier to communicate with peripherals. Because of its dynamic device discovery and low microcontroller requirements, OneWire has seen great use as a means for plug-and-play employee/user identification [103]. OneWire has also been used to communicate between laptops and charger to display battery charge level in the Apple ecosystem [1].

The Control Area Network (CAN) [266, 249] protocol is designed to provide reliability in harsh environments. Wired transmissions are made reliable through the use of differential drive and fault tolerance guarantees at the physical layer. Faults arising from the hot plugging (dynamic connection) of devices and one of the differentially driven wires breaking does not affect reception. CAN operates as a broadcast bus and allows multi-central communication (M:N). CAN networks can also communicate up to 1 Mbps in high speed mode, or 125 Kbps in low speed (but reliable) mode. Because of its reliability and flexible topology CAN is the de facto communication protocol for critical control systems.

Differentially driven lines are used to represent binary ones and zeros. Lines are assigned two identifiers: CANH and CANL. When the voltage of CANH is greater than that of CANL, the line state is said to be dominant. A device has to drive the bus to create the dominant state, hence its name. When the inverse is true the line is state is said to be recessive. The recessive state requires no microcontroller intervention as two passive resistors bring the bus to its default state. A dominant line state represents a binary zero, and a recessive state represents a binary one.

Messages, not packets or byte streams, are sent on the CAN bus. Messages are split into requests (*remote frames*) and responses (*data frames*). Every message contains an 11-bit frame identifier that uniquely identifies a message, along with a size field, a maximum payload of 8 bytes, and a 15-bit cyclic redundancy check. A single bit is left for devices to acknowledge messages and another to describe whether the message is a data or remote frame. Frames a delimited by a 1 bit long start of frame marker (dominant) and a 7-bit long end of frame marker (recessive).

Deterministic bus arbitration enables multi-central operation. Bus arbitration is achieved through carrier sense multiple-access with deterministic error correction. During transmission of frame identifiers, each transmitter senses the state of the line. When transmitting a recessive bit, each device samples the line to see if another device is holding the line in a dominant state. If there is, the device transmitting the recessive frame identifier backs off and prepares to receive the message. Bus arbitration is therefore deterministic and non-destructive as messages with the lowest frame identifier always win arbitration.

The Universal Serial Bus (USB) [264] is designed for the plug-and-play connection of peripherals and computers. USB devices are connected in a star topology (1:1) to a central host. Communications are asynchronous and supported by a differentially driven physical layer. Since its introduction in 1996, the USB standard has exploded in popularity and has since undergone many revisions. Revisions have broadly introduced increased speed—now USB devices can communicate from 1.5 Mbps in low speed mode, to up to 40

Gbps in SuperSpeed+ mode. Connectors and wires have also decreased in size in keeping with the miniaturisation of electronics over the past few decades. Speed improvements and miniaturisation have broadly increase the complexity of USB. Speedy asynchronous transmissions require highly accurate clocks to correctly send and receive data. As a result, few microcontrollers support the USB standard in hardware, and those that do operate in low/full-speed modes, the simplest of all modes to implement.

USB devices are classified into host and peripheral devices. A host device directs traffic on the bus and polls peripheral devices for data in a round-robin fashion. When a peripheral USB device is first connected to a USB host, the host triggers the enumeration process. A reset signal triggers enumeration. During reset, the peripheral defines the data rate of upcoming transmissions. Once reset, the peripheral device sends a description of its device functions and is assigned 7-bit address by the host. The host then loads the corresponding software drivers for the device if available. When drivers are loaded, the peripheral is considered configured and can then be used by applications running on the host.

USB peripherals can adopt many device functions. It is common for devices to do so and those that do are known as composite USB devices. The host requires a software driver to interact with each device function. Standard software drivers are standardised and provided for common device functions, allowing hosts to support many peripherals by default. Class identifiers, sent during the enumeration process, allow hosts to load the corresponding software driver for each device function. USB also allows developers to define custom device functions. However, custom device functions require users install additional drivers.

Each device function is presented as a USB interface to the host during enumeration. Interfaces are composed of one or more endpoints that are divided into in and out endpoints. Counterintuitively, in endpoints are used by peripherals to communicate with a host, and out endpoints are used by a host to communicate with peripherals. A USB device can have up to 32 endpoints in total, 16 in and 16 out, and 1 in and 1 out endpoint are used for enumeration and other control operations. Each endpoint has a defined transfer type which can be one of: *Control*, *Interrupt*, *Isochronous*, or *Bulk*. Control transfers are used during an enumeration, and for configuring interfaces at run time. Interrupt transfers can only be used by peripherals and are used to gain the attention of the host. Isochronous transfers are generally used for continuous streams of data. No delivery guarantees are provided, but each transfer includes a cyclic redundancy check. Bulk transfers are used for large bursts of data and are given delivery and reliability guarantees. An understanding of all these deeply technical concepts is required to implement a USB driver.

Synchronous

Inter-Integrated Circuit (I2C) [255, 137] is a wired protocol for short-distance, low-throughput communications. I2C is traditionally used to connect a microcontroller to 1 or more peripherals on a PCB (1:N). All peripherals share a data and clock line to form a bus (Figure 3.6, left). The bus is connected to a central microcontroller that initiates every transaction. Multi-central operation is not well supported and usually only one central device can operate. Though I2C supports data rates ranging from 100 Kbps to 1 Mbps, communication speed is always constrained by the slowest peripheral connected to the bus.

Every I2C peripheral has a custom interface that defines a number of registers that can be written to or read from. Registers are accessed through two high-level operations: read from a register and write to a register. Each high-level operation is composed of two back-to-back low-level transactions. A read operation is made up a write transaction followed by a read transaction. The first write transaction sets the register address and the following read transaction contains the peripherals response. A write operation is made up of two write transactions, the first sets the register address and the second sets the value.

An I2C transaction is composed of bytes and can only be initiated by a central device. All bytes are transmitted most significant bit first and an additional 9th bit (not represented in memory) is set by the receiver on the wire to acknowledge that a byte has been successfully received. The first byte of an I2C transaction always contains the peripheral address in the upper 7-bits and the operation (read/write) in the least significant bit. No acknowledge bit after the address byte usually indicates the peripheral is not connected to the bus. An operation specific byte always follows the address byte. Due to the 7-bit address space it is often the case that different peripherals have the same address.

Serial Peripheral Interface (SPI) [199] is a one-to-many full duplex protocol for short-distance, high-throughput communications (1:N). SPI is traditionally used to connect microcontrollers and peripherals on the same PCB. All peripherals share data out, data in, and clock lines from the central microcontroller to form a shared bus (Figure 3.6, right). Peripheral chip select lines are connected from microcontroller to each peripheral and is used in lieu of an address to signal the active peripheral. It is also possible to daisy chain SPI devices together to act like a shift-register. In this configuration, peripherals consume a chunk of data and pass the remaining data onto the next peripheral in the chain.

Each SPI peripheral has a specialised interface that expects a command followed by a byte stream. The freedom for electronics manufacturers to define custom byte stream interfaces make SPI ideal for high-throughput input/outputs like displays. Possible command and byte stream formats are listed in a data sheet and it is the job of a developer to extract

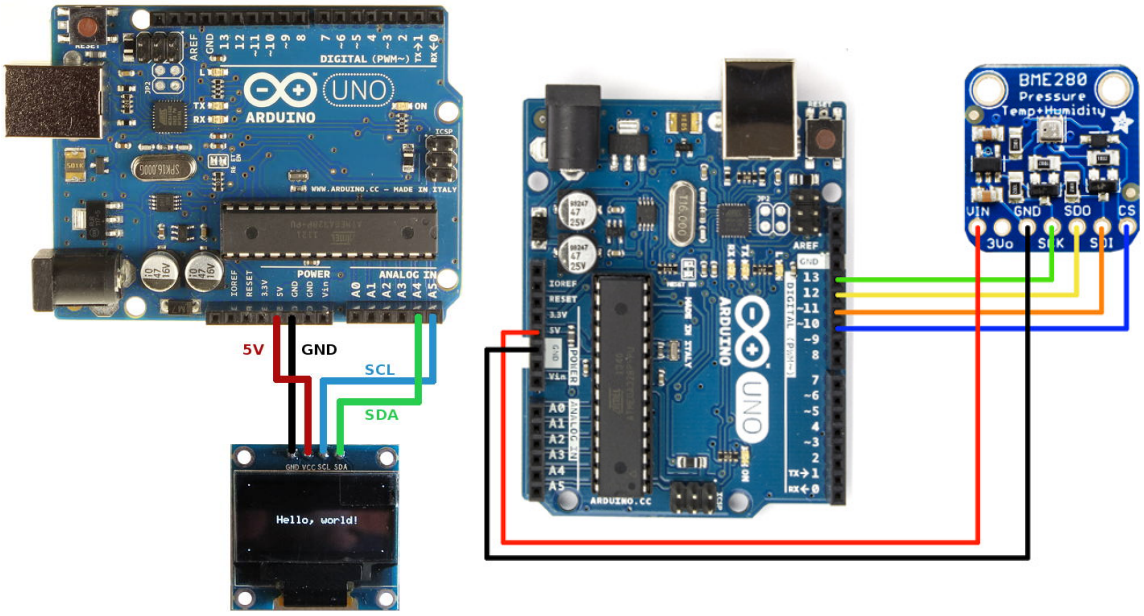

Fig. 3.6 Connecting an I2C peripheral (left) and an SPI peripheral (right) to an Arduino Uno.

information from data sheets to create a software driver. Software drivers simplify application development by providing more descriptive, abstract APIs. Interfaces are rarely shared between peripherals and so a software driver is required for each. The job of creating software drivers is difficult and requires deep technical experience.

Emerging wired protocols

Over the past few decades, the way we use technology has undergone massive change. However, the wired protocols used by many technologies has remained constant. New requirements, environments, and advances in technology mean that these more traditional wired protocols are stemming technological innovation.

Recognising the need for new wired protocols, a consortium of electronics manufacturers, including Intel and STMicroElectronics, formed the Mobile Industry Processor Interface (MIPI) alliance [226]. Together, the MIPI alliance are designing replacement protocols for the ever-changing technological landscape. However, whereas the embedded community has flourished on open software and open hardware, the MIPI alliance insists on defining standards behind closed doors, filing patents to secure their intellectual property, and hiding standards documentation behind paywalls. This behaviour prevents adoption and deeper inspection of the standards they contribute. Nevertheless, from the limited documentation

available, it is clear that the MIPI alliance are making strong contributions to the future of wired protocols.

Improved Inter-Integrated Circuit (I3C) [26, 226, 209] (also known as SenseWire) is pitched by MIPI as an inter-operable replacement for I2C that brings multi-central capability and faster data rates. I3C is synchronous, and in single data rate mode it can support communication rates up to 12.5 MHz. I3C also attempts to standardise a set of base registers across every device. Every I3C device must support register access to a 48-bit provisional identifier, an 8-bit bus characteristics field, and an 8-bit device characteristics field. The bus characteristics field defines the device role and whether it is high speed capable. The device characteristics bit field defines the class of device (i.e. accelerometer, button, temperature sensor). Though interoperability is claimed, standard I2C devices cannot be connected to an I3C bus without firmware modification. Even with modified firmware, when an I2C device is connected to the bus the majority of I3Cs features cannot be used.

I3C devices can have one of three roles. Devices with the *main-central* role are responsible for controlling all communication on the bus. *Secondary centrals* are central devices that are not in control of the bus. A secondary central can request to take control of the bus at any time. The final role, an I3C *peripheral*, is essentially a smarter I2C peripheral that can initiate communication and request in-band interrupts. An I3C peripheral still requires data transmission is clocked by the main-central device (synchronous) and it is not clear what happens when the main-central device is unexpectedly removed.

The physical layer is broadly the same as I2C but has additional capabilities. One such capability allows for asynchronous in-band interrupts to request the attention of the central device. This means that peripherals do not need a separate interrupt line to signal that data is ready. I3C peripherals can also generate hot-join events allowing central devices to detect dynamically connected peripherals. Both of these capabilities are signalled by the peripheral sending its own address on the bus.

As any device can now communicate, at power-on each device is dynamically assigned a 7-bit address by the main-central device. Once assigned, the dynamic address must be used for any subsequent transactions on the bus. Address assignment happens in sequence using a concatenation a 48-bit device identifier and bus characteristics field. As with CAN bus arbitration, the device with the lowest concatenated value has its address assigned first. Using the short 7-bit address over the 48-bit provisional maintains compatibility with I2C. Outside of address assignment, bus arbitration is performed using the dynamic address, where the lowest address always wins arbitration.

M-PHY [94] is a physical standard for high speed bi-directional inter-processor communications defined by MIPI. M-PHY is designed for high throughput data streams, is point-to-point, and full duplex. M-PHY has been used to connect microcontrollers to high data rate graphical interfaces and file systems—use cases where data throughput requires high, low latency connectivity. M-PHY supports a variety of speeds from 10 Kbps to over 10 Gbps. According to wikipedia, M-PHY uses low-voltage differential signalling resulting in low electromagnetic interference. M-PHY cannot be inspected more deeply as the specification is restricted to paying customers.

UniPro [63, 158, 197, 141] (or Unified Protocol) is a transport protocol for different physical layers, one of which is M-PHY. UniPro devices connect to each other through point-to-point connections, and UniPro devices can form larger networks through daisy-chaining. UniPro supports dynamic addressing of up to 128 devices and offers data integrity and reliability guarantees. Practically, UniPro cannot be used to connect devices together dynamically. Hot plugging is not yet supported. Due to, once again, a closed specification, we cannot provide further detail on UniPro.

3.2.2 Toolkits for hardware composition

Composing a physical computing device requires deep technical experience. Citizen developers are required to know the principles of wired protocols and competently connect low-level electronic components together. Recognising such complexity, the ubiquitous computing community have focussed efforts on simplifying the process of composing an embedded device via toolkits. Toolkits are designed to abstract the complexity of electrical components by using pre-built hardware modules. Modules are usually easy-to-connect removing compositional complexity and aligning with the iterative development practices of citizen developers.

This section covers toolkits that are closely aligned with the contributions within this thesis. We start by discussing toolkits for prototyping physical computing devices, followed by toolkits for creating modular circuits, finishing with toolkits designed to make it easier to integrate electronics into garments (wearable technology).

Composing physical computing devices

Phidgets [166] (Figure 3.7, left) aims to give users a set of re-usable hardware modules to build interactive hardware interfaces for PCs. The toolkit contains a number of peripherals and a main board. Peripherals are interface elements like switches and buttons and are

connected to a main board via USB. In turn, the main board presents a USB interface to the PC that reflects each currently connected peripheral.

A software programming environment allows developers to create Phidget-controlled applications. The programming environment dynamically updates the list of available components and APIs as peripherals are connected and disconnected from the main board. When peripherals detect interactions, events are propagated to Phidgets applications. Phidgets lowers the barrier to entry for building hardware interfaces by making use of the dynamic properties of USB.

The Calder toolkit [198] points out that building a prototype physical computing device requires individuals with highly specialised skills and tools. Calder seeks to make the process of building a physical computing device easier and is aimed at interaction designers to help guide the design of hardware interfaces.

The toolkit provides simple analogue and digital IO modules to users. Each type (analogue/digital) of module has a specialised non-reversible connector that connects to dedicated sockets on a hub board. The hub board is the central connection point for all modules and is connected to a PC via USB for communication and power. Applications can be written for the Calder toolkit using a custom programming environment that bears some similarity to the Phidgets programming environment. The hub board propagates connection events and interactions detected by peripherals to Calder applications running on a PC. Using the Calder toolkit, functional foam prototypes can be quickly created, simplifying the prototyping of physical computing devices.

Sankaran et al. [250] point out that many of the prior toolkits constrain developers to interface modules defined by toolkit creators. They therefore seek to give interaction designers the freedom to design their own interactive interfaces through Blades and Tiles.

Blades and Tiles are a model for re-usable modular hardware. Tiles are the base of any project and are typically a large surface PCB with connectors for Blades. Blades are equipped with a microcontroller and one or more peripherals for distributed sensing. All Blades follow the same form factor and pin layout and are mated with Tiles via a stacking-based connection mechanism, similar to Arduino Shields. By combining a variety or Blades and Tiles, users can prototype new interfaces and input modalities.

As prototyping is an iterative process, Blades may be mated with tiles at any point. To support dynamic connectivity, the authors use two wired protocols. OneWire is used for device discovery and I2C address assignment and a time slotted version of I2C is used for communication. A separate control blade is responsible for address assignment and time slot

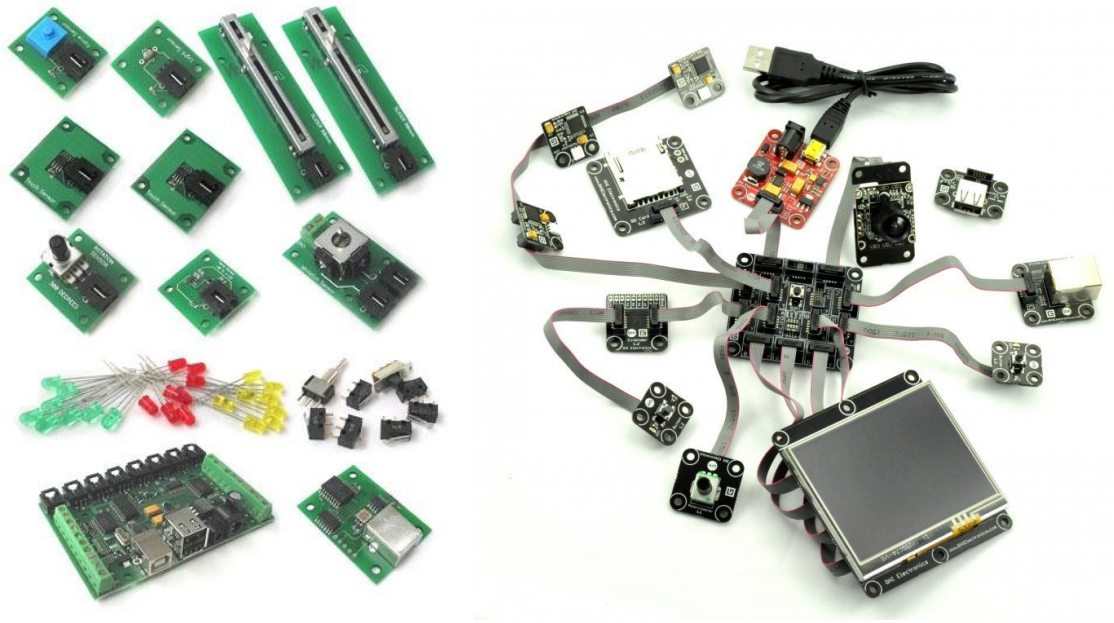

Fig. 3.7 Prototyping with Phidgets (left); and .NET Gadgeteer (right).

allocation. During their time slot, each blade has complete control over the I2C bus and it can act in a central capacity.

.NET Gadgeteer [286, 175] (Figure 3.7, right) is designed to make it easier to build physical computing devices for the Internet of Things (IoT). Gadgeteer consists of modular peripherals, a main development board, and an integrated programming environment for flexible prototyping. Peripheral boards include simple digital and analogue IO to more complex sensors and displays that rely on protocols like I2C, SPI, and UART. Peripheral boards are connected to the main board by a cable with a non-reversible 10 pin connector. The 10-pin connector supports a multitude of protocols including USB, I2C, SPI, and UART. The cable is specified in such a way that incorrect connectivity does not destroy modules.

The main board acts as the central connection point for all modules. It mostly consists of 10-pin sockets, each labelled with a connector number, and letter to indicate which protocols the socket supports. The main board also features a re-programmable microcontroller, connected to each socket via PCB traces. A dedicated USB programming port is absent from the main board. Instead, a USB client module can be connected to the board to power and program the microcontroller.

Programs are written in C# using an integrated development environment built atop Visual Studio. The environment use the .NET MF for microcontroller execution and hence,

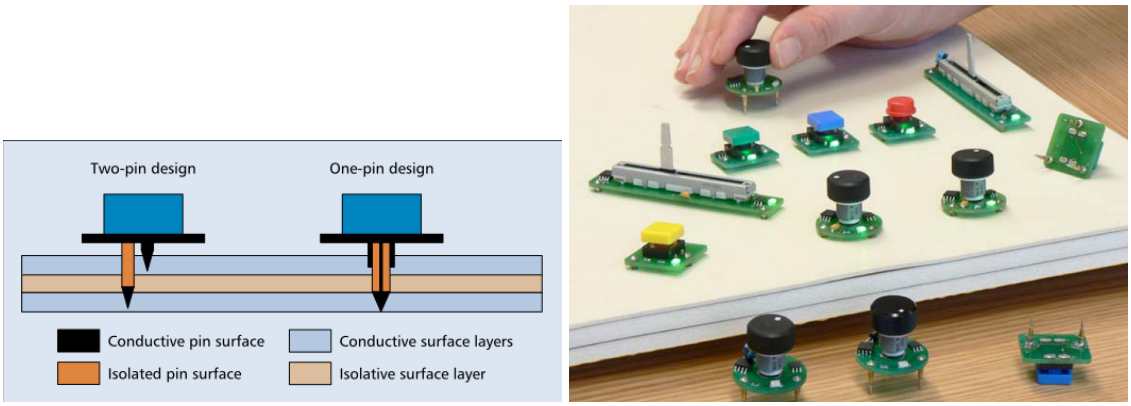

Fig. 3.8 Pin&Play pin connectors (left); and Surface-based interface composition using VoodooIO (right).

Gadgeteer applications are built on a common core that is not specific to any main board. Event-based programming is well supported making it an ideal environment for technologists. After a program has been written in the editor, users are shown a composition editor that describes how modules should be connected to the main board.

Even though populated with many sockets, users quickly hit the six I2C device limit of the main board. Gadgeteer therefore augments I2C with a daisy chaining protocol called DaisyLink. DaisyLink allows I2C devices to chain to one another, bringing more compositional flexibility and freeing up ports on the main board. DaisyLink discovers and assigns addresses to connected I2C devices in sequence, from the most upstream I2C device to the least. The most upstream device powers up with a default I2C of 127 and prevents power from passing to downstream devices. The main board probes address 127, discovers the peripheral type, and assigns it a different address. After an address has been assigned, the upstream peripheral allows power to pass down to the next peripheral in the chain. This process continues until no device responds to address 127.

Pin&Play [280] seeks to make every surface (e.g. walls) a networking medium for interface composition. Pin&Play makes use of a futuristic surface with two internal conductive planes. One plane is dedicated to power and data, the other plane is used for ground.

Pin&Play objects each have a pin that is designed to pierce the bi-planar surface. The pin internally exposes two electrical contacts for each surface and is designed in such a way that each makes appropriate contact (Figure 3.8, left). Communication between Pin&Play objects is enabled through OneWire. The surface is connected to a PC which acts as the controller of the OneWire bus. The PC regularly triggers OneWire's enumeration protocol to detect the connection of new Pin&Play Objects.

The Pin&Play approach was originally applied to a notice board where pins stored information about the documents they affixed. VoodooIO [284] (Figure 3.8, right) is another usage of Pin&Play where the technology was used to turn common household surfaces into dynamic interfaces.

Project ARA [171] (since discontinued) set out to give users the power to compose their own modular smart phone. An ecosystem of hardware modules for common device functionality like wireless communications, memory storage, processors, and battery storage were available. The more critical contribution was that users could customise their smart phone based on their needs—they could buy two battery modules if longevity was a priority. Users composed modules inside a generic smart phone frame designed to contain Project ARA modules. UniPro was used to interconnect each module.

Composing electronic circuits

littleBits [111] is designed to make prototyping circuits simple and intuitive for children. The littleBits ecosystem is made up of a number of easy-to-connect colour-coded hardware modules called littleBits (Figure 3.9, left). There are four types of bit: input, output, wire, and power. Input bits (pink) contain peripherals like sensors, buttons, and switches, and output bits (green) contain peripherals like lights, fans, and speakers. Wire bits (orange) are used to interconnect inputs and outputs but do not act only as wires. Wires can be simple logical components like AND, NOT and OR gates, entirely virtual using technologies like BLE, or even feature reprogrammable microcontrollers. Power bits (blue) feature adapters for many common energy sources like USB and household battery.

Magnets are used to physically connect littleBits to one another. The magnet-based connectors consist of two outer magnets and three inner electrical contacts. The former prevents users from connecting modules incorrectly, and the latter supplies power, data, and ground. Magnetic connectors enable simple and intuitive prototyping.

littleBits do not communicate via any sort of protocol. Instead, they use simple digital and analogue outputs. Output voltages of one littleBit are fed into the input of the next and integrated electronic circuitry or a small microcontroller is used to change the resultant behaviour. No computer is used in any part of the default littleBits kit. Intuitive connectivity and a relatively simple circuit abstraction makes it simple to build interactive circuits.

Circuit Stickers [176] (Figure 3.9, right) intends to make circuit design as simple as printing a document. The Circuit Stickers ecosystem features a number of predefined PCB modules. Each module has exposed electrical contacts and a special conductive substrate on

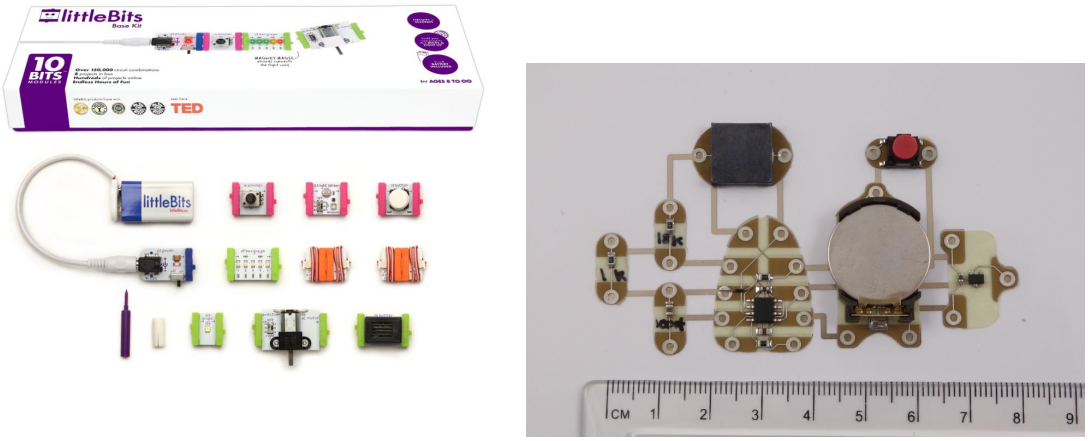

Fig. 3.9 littleBits modules (left); and Circuit Stickers (right).

its underside. The special substrate acts like sticky back plastic, allowing users to stick PCBs to conductive surfaces.

To use Circuit Stickers, users print a circuit onto paper using conductive ink and a commercially available inkjet printer. Circuit designs are created in graphics editors like Microsoft Visio and footprints for Circuit Stickers and common electrical components are made available via a Visio plugin. The combination of intuitive design tools and stickable electronics simplifies circuit design and hits a prototyping "sweet spot" between PCBs, breadboards, and higher level modular systems.

Composing wearable technology

Over the past decade wearable technology has seen huge growth with Apple reporting 10 billion dollars in revenue from its wearables offerings alone in Q1 of 2020 [95]. Wearable technology as we know it today devices that we wear on our bodies. However, the future of wearable technology lies in embedding electronics into the clothes we wear. With this vision in mind, the ubiquitous computing community have created a number of toolkits for embedding technology into garments.

Toolkits are designed with the unique requirements of electronic garments. Integrating technology into garments needs to be intuitive and simple for the technologically inexperienced. Toolkits also need to be designed in such a way that they become just another tool in the fashion toolbox. Wearing garments also creates wear and tear to which any integrated electronics are also exposed. Finally, supplying electronics with power and data is challenging when embedding electronics into garments. Wireless technologies require

Fig. 3.10 Prototyping with Lilypad Arduino (left) and i*CATch (right).

devices be equipped with individual power supplies, whereas wired protocols require power and data to be routed across garments to each device.

The LilyPad Arduino [123] (Figure 3.10, left) is one of the first toolkits for wearable technology. The toolkit features a reprogrammable Lilypad Arduino board and a number of compatible input and output modules, called Petals. Petals are components like sensors, speakers, and RGB LEDs.

Great care was taken throughout the design of the LilyPad hardware. Every module—including the LilyPad itself—is circular in shape and backed by conductive fabric. The conductive fabric overlaps with exposed GPIO electrical contacts on the underside of each piece of hardware to create an electrically connected *fabric pad*. Conductive thread is then used to connect the LilyPad to Petals, creating a familiar experience for fashion designers through the use of thread and sewing needles. Fabric pads have a large surface area, and Petals use digital and analogue interfaces to make routing wires across a garment as simple as possible—each Petal usually just requires power, data, and ground.

Though designed for easier e-textiles work, the LilyPad has seen wider adoption in education. Evidence suggests that the combined force of electronics and fashion leads to wider participation in computer science. However, its usage is somewhat stunted due to adoption of the low-level Arduino programming language to program the LilyPad.

Eduwear [182] builds on the work of the LilyPad Arduino but it refocuses the LilyPad ecosystem to target education. The Eduwear toolkit provides addition petals, vastly increasing the possible combinations of hardware. Petals, for the first time, include I2C and SPI based peripherals. To make it easier to wire bus-based protocols on garments, Eduwear also

provides pre-fabricated data buses. Data bases are pieces of fabric with conductive thread already sown in. Identifying complex programming as a barrier to adoption, Eduwear also supplies a custom visual programming environment to support novices.

The i*CATch [228] (Figure 3.10, right) toolkit is designed to better enable cross-curricular learning through 'wearable computing'. i*CATch garments are divided into patches—a similar approach to Quilt Snaps [122]. Internal wires made from conductive fabric run along the inside of each patch and snaps receptacles are pierced through each wire. Conductive snaps are used to connect patches to create a consistent I2C bus that is routed through the garment. Additional snaps on each patch allow for the connection of popper-plug equipped I2C peripheral boards.

A main controller board featuring a reprogrammable microcontrollers acts on inputs from peripheral sensors. The main controller board is programmed from a visual programming environment that wraps the C/C++ Arduino HAL APIs. The i*CATch programming environment adopts a flow-based programming model. Combined the programming environment and patch-based approach have been proven to widen participation and offer an easier introduction to creating wearable technology.

The Yet Another Wearable Toolkit (YAWN) [270] toolkit is designed to simplify bus-based communications between modules. The toolkit supplies a number of different hardware peripherals that are clipped onto fabric circuits using a custom 3D-printed connector. The protocol used to communicate between peripherals is the key contribution of YAWN. The authors point out prior toolkits are using static protocols I2C and SPI in dynamic contexts and therefore propose an alternative bus-based communications protocol built on UART that is designed to support hot plugging (dynamic connection/removal). A LilyPad, programmed using the Arduino IDE, directs all module communication on the bus.

Project Jacquard [240] (Figure 3.11, left) proposes new fabrics for gesture sensing that can be easily integrated into garments during mass manufacture. Gesture fabrics are conductive surfaces that can detect touches with high fidelity using a similar technique to smart phones. A gesture recogniser is attached to each piece of fabric and connected to a main microcontroller via I2C. The main microcontroller supplies power to each recogniser and transmits gestures to connected Bluetooth devices. For Bluetooth devices, like smart phones, Jacquard can be used as a touch input to change songs.

SensorSnaps [139] (Figure 3.11, right) take the place of conventional buttons in garments. SensorSnaps are designed to be compatible with off the shelf garments and quickly attachable and detachable. Each SensorSnap contains a Bluetooth capable microcontroller connected to an Intertial Measurement Unit (IMU) and a small lithium polymer battery. Due to its

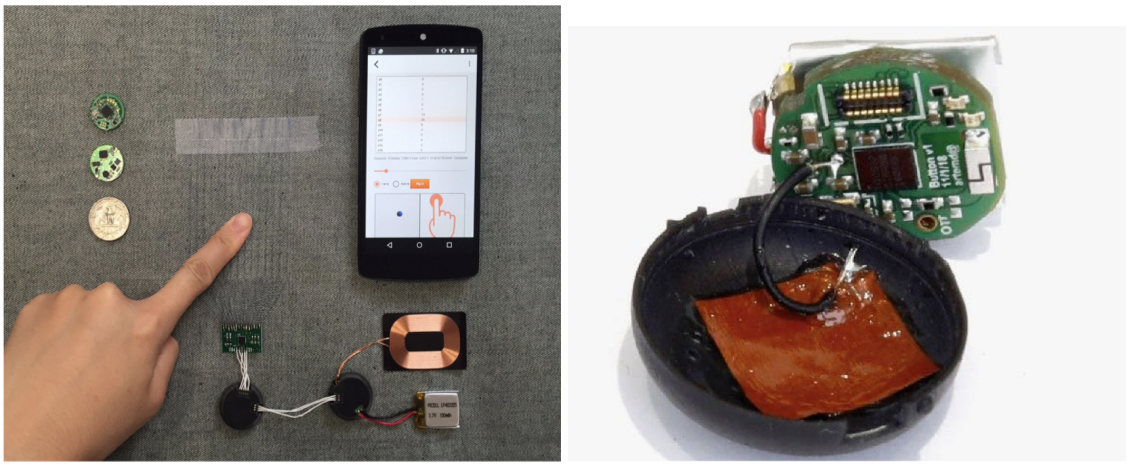

Fig. 3.11 Augmenting clothing with Jacquard (left) and SensorSnaps (right).

small battery (the size of a large button), SensorSnaps are designed for careful energy consumption—Snaps only send events (e.g. tap/rotate) and do not stream real-time IMU data. SensorSnaps are connected to a Bluetooth central device, which can be a consumer device like a smart phone or laptop, or a microcontroller with Bluetooth capabilities. Events received from Snaps over Bluetooth act as inputs to applications.

3.2.3 Analysis

Wired protocols underpin the composition of physical computing devices. They define the interfaces and protocols for communicating between microcontrollers and peripherals over conductive materials like wires.

Every wired protocol is designed with specific goals in mind. Protocols like I2C and SPI are designed for efficient connectivity in static environments. Protocols like CAN are designed for resilience and reliability in extreme environments. And protocols like USB and OneWire are designed for plug-and-play connectivity in dynamic environments.

Wired protocols each support a different topology. Point-to-point (1:1) topologies are the simplest to implement. Devices are directly connected to each other and communication is bi-directional and simultaneous. A one-to-many topology (1:N) allows a single central device to control many peripherals. The central device directs all communication on the bus and some one-to-many protocols allow devices to swap from a central role to a peripheral role (and vice versa). A many-to-many topology (M:N) allows any device to interact with another at any time, bringing the most flexibility and dynamism to wired composition. As

devices all share the same communications medium (i.e. a wire), devices must contend for control of the bus (bus arbitration).

Many toolkits have been created by the ubiquitous computing community to simplify hardware composition. There are toolkits for creating physical computing devices, prototyping circuits, and for creating interactive fashion garments. Broadly, each toolkit is composed of a number of hardware modules that each have a dedicated function (i.e. a button, switch, or sensor). Modules are connected using wires or custom physical connectors, and communicate using wired protocols. Some toolkits also recognise the complexity of programming hardware and also contribute custom programming environments.

Many toolkits seek to reduce compositional complexity and support more flexible development by reducing the number of individual wires. .NET Gadgeteer supports many protocols using a larger 10 wire cable and connector. The Eduwear toolkit supplies pre-fabricated data buses to reduce the complexity of routing bus-based protocols across garments. And Voodoo IO uses a single wire to connect devices together on a noticeboard.

Toolkits also saw great value in making hardware composition *more dynamic*. The dynamic connectivity (hot plugging) and device discovery offered by OneWire made it an ideal protocol for composing dynamic hardware interfaces in Pin&Play and VoodooIO. Dynamic addressing and device discovery added through DaisyLink made modular composition simpler and more flexible in .NET Gadgeteer. The dynamic properties of OneWire augmented I2C with dynamic address allocation and multi-central capability in Blades and Tiles. The prevalence of dynamism across all toolkits suggests there is an established link between adding dynamism and reducing the complexity of hardware composition.

A similar evolution towards protocol dynamism is taking place in industry. UniPro offers dynamic and high-speed interconnects and I3C brings dynamic device discovery, dynamic connectivity, and multi-central operation to I2C. These new and emergent protocols are also providing hardware abstraction through standardised software interfaces. Existing protocols that already provide hardware abstraction, like USB, have seen great success for hardware composition by technically inexperienced citizens.

There is little support for more dynamic protocols on low cost microcontrollers. More dynamism generally leads to an increase in complexity, ultimately requiring more silicon to implement. This increases microcontroller cost. The market demands low costs however, and the drive for cost efficiency means that more dynamic and easy-to-use protocols are not widely supported.

Table 3.3 summarises the wired protocols discussed in this section. Protocols are ordered by implementation complexity and are profiled by the design requirements (HC1—HC3)

	Communication paradigm	Data wires (minimum)	Support on microcontrollers	Dynamic connectivity (HC1)	Dynamic device discovery (HC2)	Hardware abstraction (HC3)
USB	1:N	2	Moderate	Yes	Yes	Yes
UniPro	1:1	4	Low	No	Yes	Yes
I3C	1:N	2	Low	Yes	Yes	Yes
Onewire	1:N	1	Moderate	Yes	Yes	Yes
CAN	N:M	2	Moderate	Yes	No	No
I2C	1:N	3	High	No	No	No
SPI	1:N	4	High	No	No	No
RS485	N:M	4	High	No	No	No
RS422	1:N	4	High	No	No	No
RS232	1:1	2	High	No	No	No
JACDAC	N:M	1	High	Yes	Yes	Yes

Table 3.3 Properties of different protocols for composing embedded devices. Protocols are ordered by implementation complexity and profiled by the number of design requirements (HC1—HC3) they enable.

from IQ2. For context, we place JACDAC, one of the contributions of this thesis, at the bottom of the table.

3.3 Wireless networking

This section explores wireless ad-hoc networking protocols that build upon the energy efficient 802.15.4 and Bluetooth Low Energy (BLE) wireless standards. We begin by outlining the common design considerations observed during the design of many wireless protocols (Section 3.3.1) followed by an in depth review of the BLE and 802.15.4 wireless standards (Section 3.3.2). We then discuss wireless protocols that build on these standards, grouped by they observe a routing-based (Section 3.3.3) or flooding-based (Section 3.3.4) approach to packet propagation. In Section 3.3.5, we analyse and profile these protocols by the design requirements (WN1—WN3) extracted from IQ3.

3.3.1 Design considerations

Protocols are always designed with specific use cases in mind. Different use cases bring different design consideration that ultimately affect the properties and behaviours of wireless

protocols. In this section we outline design considerations that are taken into account when creating a wireless protocol.

Hardware considerations

Devices, or *nodes*, in wireless networks are physical pieces of hardware, often PCBs with integrated antennas. Hardware is the actuator, the sensor, or the control mechanism through which distributed applications act. Node capability varies from node to node.

Energy consumption Some nodes may have unlimited power through connection to mains circuits, whilst others may be powered through low-power energy harvesting. Power hungry wireless protocols impact application lifetime for non-mains powered devices. We categorise related work into *low*, *medium*, and *high* power consumption protocols. Low power consumption protocols have the longest lifetime, whereas high power consumption protocols have the shortest.

Memory consumption Nodes have differing amounts of RAM and flash memory, and code to operate protocols consumes both types of memory. The memory consumption of protocols can therefore exclude devices as consumption grows. Not only do memory demands reduce device selection, but greater memory consumption increases the cost of each node. We categorise related work by *low*, *medium*, and *high* memory consumption with respect to both flash and RAM.

Quality of Service

Required service quality depends on the use case and also interplays with energy consumption. A high quality of service generally leads to greater energy consumption.

Latency is the duration from when a packet is sent to when it is received by the intended device. *High latency* protocols conserve energy by transmitting less often at the expense of real-time behaviour. They align well with long-term sensor deployments, offering latencies in the range of minutes. *Medium latency* protocols sacrifice some real-time behaviour for greater energy efficiency. They offer comparatively low latencies to high latency protocols, in the range of seconds rather than minutes. *Low latency* protocols offer real-time behaviour at the expense of energy efficiency. Low latency protocols have sub-second latencies.

Reliability is the measure by which packets are delivered successfully in a network. Protocols do not have to guarantee reliability, and those that do not, make it the responsibility of the sender to ensure packets are delivered successfully. Protocols that do have high reliability generally consume more energy and more memory. We provide two categories for related work, reliable, for protocols that guarantee reception, and unreliable, for protocols that do not.

Data throughput is how much data can be propagated through a network, usually measured in megabits per second. The data throughput of an ad-hoc or mesh network is limited by how devices are joined together and how packets are propagated through a network. Those with *high* throughput allow more data at the expense of energy efficiency, and those with *low* throughput, the inverse.

Scalability

Limits to scalability impact the kind of applications protocols can be used for. Mismatching protocol scalability with application demands can add unnecessary metadata to packets, and overheads like network maintenance.

Maximum network diameter Network diameter is a metric that equates to distance or ability to propagate. Networks that support large diameters generally span hundreds of metres, where small network diameters tend to cover less than a hundred metres. Protocols have limits where they become unusable. Using a protocol with a large network diameter for a small network adds unnecessary overhead. Applying a protocol to an application that far exceeds the supported network diameter likely leads to a network with poor reliability.

Maximum number of devices All wireless protocols have a limit to the number of devices they can support. Like network diameter considerations, using a protocol that supports a large number of devices for small networks adds unnecessary overhead. Exceeding the maximum number of supported devices with a large network is likely to lead to negatively impact protocol performance.

Ease of use

Designing a wireless protocol for use by citizen developers brings new and additional considerations to the design of wireless mesh networking protocols.

Latency The emerging domain of DIY IoT shifts projects towards more reactive and interactive deployments. *High latency* protocols are unintuitive for projects such as these, as users cannot discern between packets that have not yet been transmitted and errors that may have occurred during propagation. *Low latency* protocols suit reactive scenarios far better providing almost immediate response to transmission requests.

Reliability Again, the reactive and interactive nature of DIY IoT deployments brings requirements for reliability. Protocols prone to errors degrade user experiences, especially considering the lack of feedback capability on physical computing devices.

Energy consumption The applications of DIY IoT are numerous, and certainly some applications will require devices to be deployed with battery or energy harvesting technologies. Here, energy consumption becomes a concern and protocols that have high energy consumption are not suitable.

Configuration Some protocols require *pre-configuration* or *deploy-time configuration* of network parameters by users. Runtime parameters that determine protocol energy efficiency may also require configuration. Protocols that need to be configured require a technical understanding of protocol operation. When considering new types of re-programmable physical computing devices and iterative development practices, deploy-time configuration becomes a frustrating experience for users. The best protocols for citizen developers require little or *no configuration*.

Infrastructure topology Some protocols assume a constant infrastructure to provide a backbone for device-to-device communications. Networking communities differentiate between those with infrastructure as *mesh* networks, and those without as *ad-hoc* networks. However, in many cases ad-hoc networks with internet connected uplinks can be considered mesh networks. Instead we define the following terminology: *fixed infrastructure* for protocols that require backbone devices; and *dynamic infrastructure* for protocols that do not require backbone devices, but can be augmented to provide Internet connectivity or more reliable performance. For citizen developers, protocols that assume a fixed infrastructure require technical understanding to deploy; dynamic and emergent infrastructures are more intuitive.

Network partitioning A wireless network can be disrupted by removing a single node. This may result in a network being divided into multiple sub-networks, or *partitions*. Some wireless protocols are *tolerant* to partitioning and will gracefully recover when the critical node is connected or device positions are rearranged. Other protocols are *intolerant* to network partitioning and may require nodes to be reconfigured to replace missing nodes.

Commercial Availability Wireless protocols are only enabled by the hardware that supports them. Wireless protocols therefore have little utility if hardware is not commercially available for purchase.

3.3.2 Wireless standards

Here we discuss the two most popular wireless standards for short-range communications: Bluetooth Low Energy (BLE) and 802.15.4. Related work builds upon either of these wireless standards, so we provide a detailed account of each.

Bluetooth Low Energy (BLE)

Bluetooth Low Energy (BLE) is a physical standard for short-range, low-power communications. BLE defines all layers of the OSI model except applications, which can be defined by users. BLE is still developing as a protocol and it was first defined in specification 4.0 which outlined the wireless symbol format and the abstractions that form the basis of the BLE stack. This first iteration also defined two device roles: central and peripheral. A central device controls 0 or more peripheral devices, and a peripheral device can only communicate with one central device. A network of central and peripheral devices forms a *piconet*. A later evolution of the specification, version 4.1, allowed for devices to swap their roles dynamically, enabling intercommunication between piconets to form a *scatternet*. All supported topologies are shown in Figure 3.13.

Though initially multiple data rates were supported, the standard evolved to support just one, one megabit. To reduce interference with other prominent wireless protocols in the 2.4 GHz spectrum, such as WiFi, BLE divides the frequency spectrum into 100 different frequency bands, also known as *channels*. When operating, groups of BLE devices hop between channels to avoid interference and increase node density. Error detection is performed by calculating the received Cyclic Redundancy Check (CRC) and comparing it with the 24-bit CRC contained in the packet. Gaussian Frequency-Shift Keying (GFSK) is used to represent binary data on the air. Before transmission, data signals are passed through

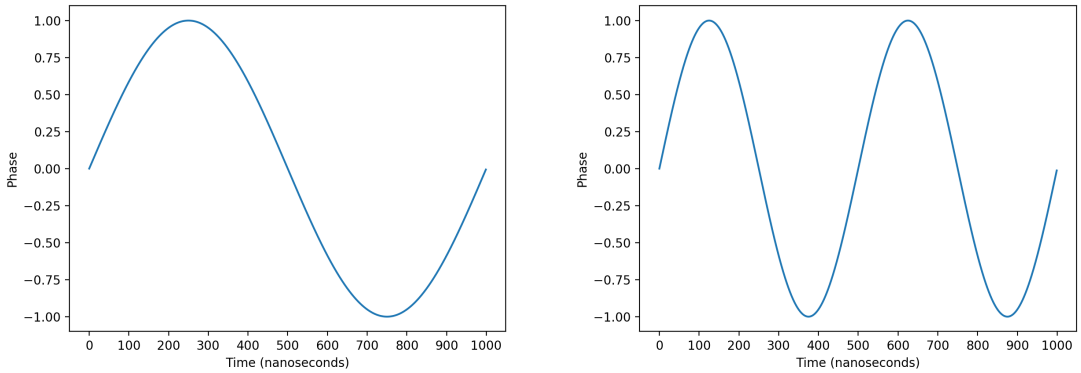

Fig. 3.12 BLE symbols at 2.4 GHz: a logical one (left); and a logical zero (right).

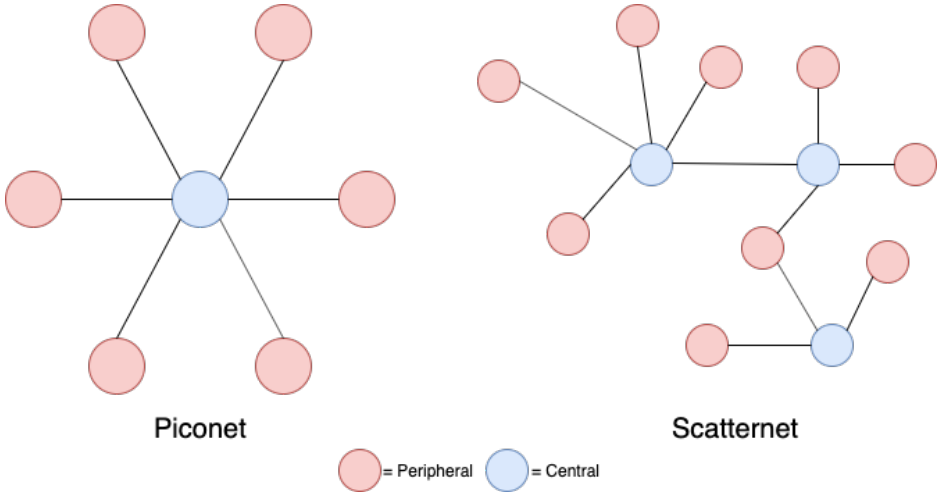

Fig. 3.13 Supported BLE topologies as of specification version 4.1.

a gaussian filter that smooths the beginning of each symbol. Corresponding waveforms are visualised in Figure 3.12.

BLE devices send *advertisements* in dedicated advertisement until a connection is established. Advertisements can be configured to periodically transmit at different rates, defined by the firmware developer. Contained in BLE advertisements are a list of identifiers that reference specific *services*. A service is a resource made available to other devices once a connection is established. It is the role of the central device to poll updates from the peripheral device, though peripherals can indicate to the connected central device that data is available.

802.15.4

802.15.4 is another physical standard for short-range, low-power communications. Whereas BLE defines all layers of the OSI model, 802.15.4 defines the physical and MAC layers leaving the remaining layers to be defined by protocol creators. The physical layer can operate at three different data rates across three unlicensed frequencies: 250kbps at 2.4GHz, 40kbps at 915MHz, and 20kbps at 868MHz. The lower the base frequency, the greater the permeation of the data signal. As a result, 802.15.4 based networks can travel greater distances on a single hop than technologies like BLE. Modulation of the data signal differs at each frequency. At lower frequencies, Binary Phase-Shift Keying (BPSK) is used, whereas at 2.4 GHz, Offset Quadrature Phase-Shift Keying (O-QPSK) is used for greater symbol clarity. Devices can divide transmissions between 16 channels in the 2450 MHz band, 30 channels in the 915 MHz band, and 3 channels in the 868 MHz band. All 802.15.4 transmitters are required to implement Carrier-sense Multiple Access with Collision Avoidance (CSMA-CA) to prevent simultaneous, disparate transmissions that cause interference.

Various topologies are presented in Figure 3.14, each with different connection properties. Topologies are composed of Full Functionality Device (FFD) and Reduced Functionality Device (RFD). A FFD is a normal node that can form many connections, whereas a RFD is a device that can only form one connection to an FFD due to limited memory or energy capacity. Each topology also requires at least one *coordinator*, which is a FFD responsible for managing the network in beacon mode. In beacon mode, each coordinator defines an active and inactive period dependent on the energy demands of the application. Nodes communicate with the coordinator during the active period and sleep during the inactive period. The active period is further divided into a Contention Free Period (CFP) and a Contention Access Period (CAP). In the CFP, devices use pre-allocated slots to transmit to the coordinator. In the CAP, devices contend for access to the channel by dividing the period into back off slots, using CSMA-CA to detect collisions. In non-beacon mode, devices exclusively use CAP to communicate, leading to reduced energy efficiency as transceivers need to be constantly active. Higher layers of the stack can be used to directly manage transceiver state.

Data communication takes the form of three types of transaction: coordinator to device; device to coordinator; and in the case of a mesh topology, device to device. Device to coordinator transmission require beacon synchronisation if applicable, followed by a CSMA-CA transaction. Coordinator to device transmissions take place after a device to coordinator transmission. In this case, the coordinator simply sends packets directly to the device. Device to device transactions can take place at synchronised times in beacon modem or at any time in non-beacon mode as long as devices are in transmission range of one another.

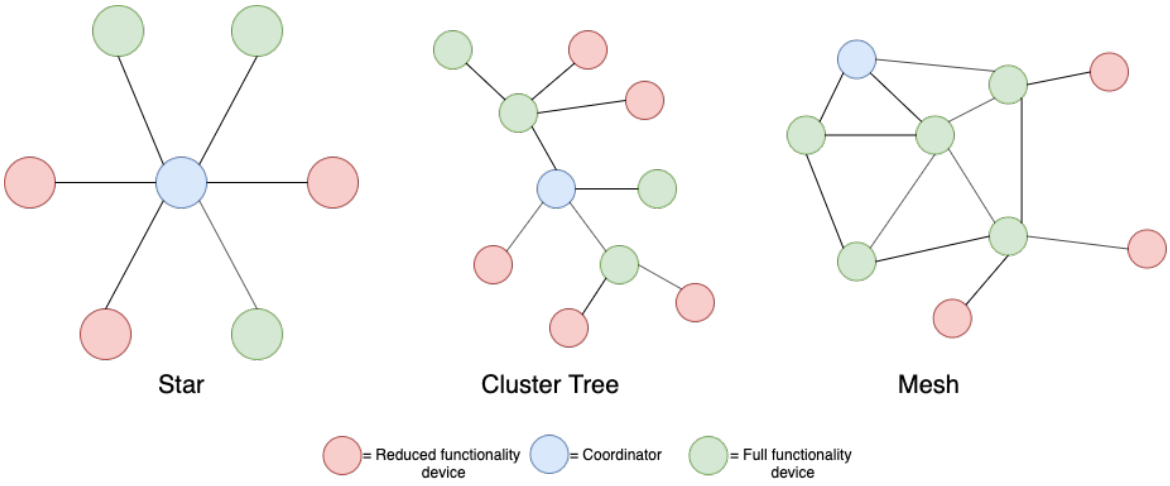

Fig. 3.14 Supported topologies in 802.15.4

3.3.3 Routing-based

Routing-based wireless protocols send packets through a network using routes that are set *statically* or computed *dynamically*. The former is better for use cases with a fixed-infrastructure, and the latter for those that have an emergent and dynamic infrastructure.

Static routing

BLE traditionally employs a star-based topology where one central node connects to one or more peripherals. Instead, Maharjan et al. [210] propose the use of a tree topology. At the root of the tree is a central BLE device with connections to N peripheral devices. Each peripheral device can act as a plain peripheral or as an intermediary node that acts as a central device to M peripherals. This continues layer on layer until the required number of nodes are connected. A pictorial representation is available in Figure 3.15, left. The addressing scheme allows for trees that are five levels deep, and transmission happens hierarchically from bottom of the tree to the top. As long as nodes remain connected the network exhibits high reliability and good energy consumption. Latency of packet transmission scales with the number of nodes in the network and on average remains low if communicating only with the root node. This solution does however require a fixed infrastructure and incurs a high configuration cost. Each leaf requires addressing information of its hierarchical central device, and intermediate nodes require peripheral and central device addresses. The nature of a tree topology means that if an intermediary node decouples from the network, an entire

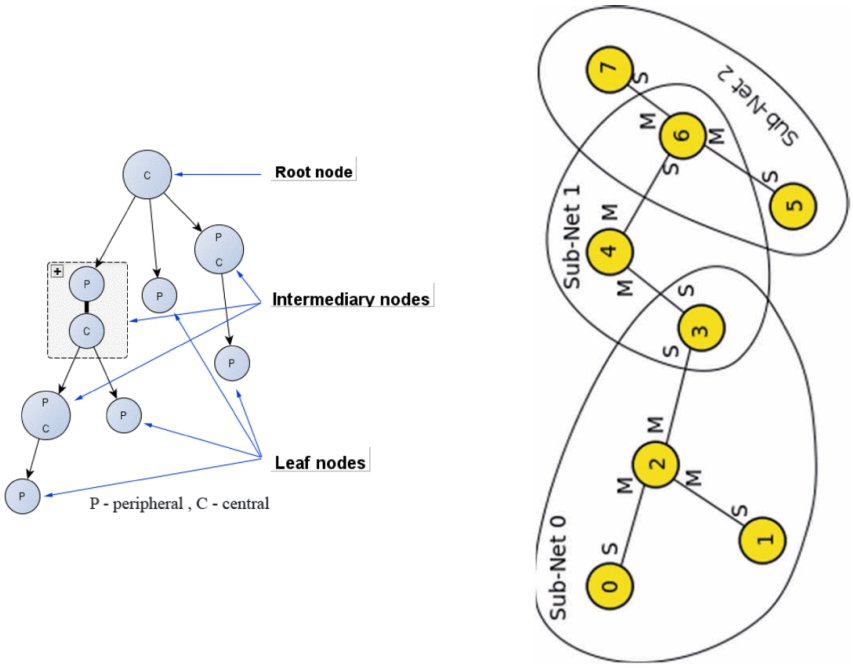

Fig. 3.15 Static tree-based routing for BLE (left); Static routing approach used in RT-BLE, where S=Slave and M=Master (right).

portion of the network can fail. The protocol also does not define a recovery mechanism for this case and is partition intolerant. No details are provided on memory consumption.

Realtime BLE (RT-BLE) [235] is another static routing-based solution for BLE. Each device can act as a central or peripheral device simultaneously, and a mesh network is formed through the interconnection of devices in central-peripheral relationships. The kinds of relationships that can be formed are pictured in Figure 3.15, right, and the central idea is that sub-networks are created and bridged via peripherals belonging to two separate sub-networks. To avoid sub-network collisions on bridged nodes, each sub-network defines its own connection interval, taking into account the connection intervals of nearby sub-networks. Each device also maintains a bit determining its network state which is checked before a central device initiates a connection. The overall reliability of the network is high and the energy consumption low, communicating using low frequency connection intervals of 20 milliseconds. The infrastructure of the network is however, fixed, requiring sub-networks contain one central node and one or more peripherals. The configuration overhead of RT-BLE is also high as connections must be defined before communication can begin. If a central device fails, there is no election process for a new central device and routes between networks fail, so RT-BLE is intolerant to partitioning.

Dynamic routing

To bring more flexibility to routing-based solutions, some works adopt dynamic routing. Dynamic routing is more supportive of evolving networks and network partitioning, qualities better suited for the kinds of applications citizen developers may create.

Introduced in the BLE 4.1 specification was support for the interconnection of multiple piconets—central-peripheral device clusters—to form scatternets. Guo et al. [169] define an approach for dynamic scatternet formation and multi-hop routing for scatternets. Scatternet topology is similar to that of RT-BLE, where a single device in a piconet acts as a bridge. Scatternets are self-forming and use a discovery process based on regular advertisements to find nearby devices in the vicinity. Packets are routed through a breadth-first search of the scatternet for the destination node. The algorithm almost works recursively, making its way through each piconet until a possible route from source to destination is found. Routes are cached in a lookup table to prevent overloading the network with routing requests. The authors' solution does require a fixed infrastructure through the user of central devices within each piconet, but low configuration and dynamic routing brings great flexibility to users. However, the routing approach and overheads introduced through bridging piconets together results in high latency. Route discovery also leads to higher energy consumption

BLE Mesh Network (BMN) [242, 262] also uses BLE advertisements to send data and form networks dynamically. BMN uses a Directed Acyclic Graph (DAG) to route packets. Each BMN node maintains two parents, one acting as a fallback if the primary parent fails. Nodes all contain a rank that represents its energy level and its distance to the root of DAG. This allows nodes to form into a directed graph that spares nodes with lower energy levels. Each node is required to store a routing table generated from its children and parents. Transmissions are sent based upon node priority which is defined statically, and not computed based on node demand. Although the approach leads to greater energy efficiency and enables flexible deployment use cases, it suffers from high latency and low throughput. It also requires manual configuration to define device identifiers and does not compute the transmission priority of nodes dynamically.

Leon et al. [201] build upon the foundation of BLEMesh, but seek to tackle the limitations of manual configuration of device identifiers and static transmission priority. Device identifier allocation is performed through an auto configuration protocol. Nodes send out a request to find a network and any nearby networks respond with a message. The node then joins and finds a free 7-bit address to use within the mesh. To maintain ownership over the address, nodes must send a keep alive message. Routing is performed via Proactive Source Routing (PSR). In PSR, each node maintains a map of the network relative to their position.

Individual maps are propagated through the network until each node has a full map. If a node disconnects, nearby nodes propagate a message to make all nodes in the mesh aware. 24 nodes can be represented in a single packet, problematic for large networks with higher node density. No evaluation was performed on physical hardware, but one can imagine that in dynamic scenarios with regular disconnection, the overhead of PSR may result in reduced throughput or dropped packets.

Zigbee [187] brings mesh, not bounded by transmission distance, to the 802.15.4 physical layer. To do so, Zigbee makes use of a well-studied routing protocol called Ad-hoc On Demand Vector (AODV) [129]. Zigbee networks still have a coordinator, but FFD nodes are known as *routing nodes*, and RFD nodes are known as *end nodes*. These distinctions stem from the need to enable low power long-term IoT deployments. For example in the Zigbee model, end nodes, like doorbells and light switches, only have to transmit when users interact with them. Network coordinators and routing nodes are expected to have a constant power supply. The AODV routing protocol requires on-demand construction of routes that are then preserved for lifetime of the network. The overhead of route construction requires the propagation of routing requests until the destination node is reached. Once reached, the destination node sends a reply via unicast transmission and through the lowest cost path. Zigbee assumes some semblance of fixed infrastructure to operate and if used dynamically, route construction can incur a significant cost and consume valuable bandwidth.

Thread is an emerging mesh networking standard for use in the home. It is compatible with IPv6 through the use of 6LoWPAN [224], allowing for the fragmentation and reassembly of IPv6 packets over 802.15.4. This reduces complexity of bridging Thread devices with the wider Internet. Thread also runs the Constrained Application Protocol (CoAP) [118] over User Datagram Protocol (UDP) [239] to transport frames across the network. Routing within Thread is performed using Routing Information Protocol (RIP). RIP requires nodes periodically broadcast routing costs to all other nodes, and costs are maintained in a table linked to device identifier. Key nodes broadcast information about nearby networks and other servers available to the Thread network. As key nodes must maintain large routing tables they require a reliable source of power to maintain network reliability. In addition, the individual cost of each device maintaining costs to all other nodes within the network adds memory overhead.

3.3.4 Flooding-based

Flooding-based wireless protocols do not maintain routes to send packets across a network. Instead, packets are propagated through networks through *retransmitting* or *repeating* transmissions. There are different approaches to flooding. *Bounded* approaches analyse packets before retransmitting them, using packet metadata to determine if packets do need to be repeated. *Probabilistic* flooding approaches forward packets only if there is a chance proximal nodes have not yet received the packet. *Directional* flooding propagates messages informed by the position of nodes within a network, usually incorporating some form of routing protocol. Finally, *concurrent* flooding uses the fundamental physical properties of radio waves to improve the reliability and reception rate of packets. Without the need for fixed nodes to provide routing infrastructure, flooding-based protocols allow nodes to more freely and deployed more easily.

Bounded Flooding

CSRMesh [293] is custom flooding-based mesh protocol for BLE. Like routing-based networking approaches for BLE, CSRMesh floods networks using advertisements. Advertisements are repeated across the three general purpose broadcast channels. Each time a packet is repeated, a Time to Live (TTL) embedded inside the packet is decremented to bound the number of repetitions. Because there is no medium access control layer, devices can communicate over one another reducing reliability. To receive advertisements, devices need to constantly scan 3 channels consuming energy, and increasing the likelihood of successful packet reception.

BLEMesh developed by Kim et al. [185] also applies bounded flooding to BLE advertisements. Rather than send individual packets, BLEMesh batches packet transmissions for greater energy efficiency. Batches are transmitted in the payload of BLE, but since advertisement payload can only fit 31 bytes, the approach yields a low data density per packet. The flood mechanism used in BLEMesh is smarter than that of CSRMesh. To minimise the energy impact of each flood, the network maintains a list of forwarding nodes considered likely to give a chance of packet delivery. Only the nodes in the forwarding list forward the packet. Upon reception, nodes that have forwarded the packet put their identifier into a secondary list. The number of nodes in the forwarding list scales with the size of the network. Using this approach BLEMesh optimises the number of repetitions for the majority of nodes to receive a packet. However, the throughput of the network is low and latency is high.

To more directly address the limitations of BLE as a ad-hoc networking protocol, the Bluetooth SIG introduced Bluetooth Mesh [106]. Although presented as a separate standard, Bluetooth Mesh builds upon the physical layer of Bluetooth Low Energy. Bluetooth Mesh uses bounded flooding and is implemented by means of a Time to Live (TTL) decremented upon each repetition. The use of flooding rather than routing leads to dynamic network creation and lowers memory overheads for individual nodes. To receive packets, nodes need to leave transceivers in receive mode which leads to greater power consumption. For nodes that require ultra low power consumption, Bluetooth Mesh specifies "friends" that act as caches for messages to low-power nodes. This is ideal for deployments that include reliably powered nodes, but less applicable for applications that do not. Moreover, networks are formed from pre-configured settings stored on individual nodes.

Probabilistic Flooding

As an alternative to bounded flooding, probabilistic flooding seeks to optimise the number of repetitions based on the probability a node has already seen a packet. MISTRAL [238] was the first to simulate a probabilistic approach to flooding. Each node evaluates the probability that proximal nodes have already received a copy of the packet and determine whether to repeat or not. The authors recognised that even traditional probabilistic approaches result in incorrectly dropped packets, especially so when applied to a flooding-based network. To guard against such over optimisation, nodes broadcast "compensation packets" that encodes a list of packets that the node has determined do not need forwarding. Missed packets can be requested from nodes who have a copy of the original packet. The authors show through simulation that this compensation mechanism significantly increases coverage when compared to purely probabilistic approaches. The need to retain copies of packets for forwarding has practically constrains networks as data throughput and network size grow.

Alternatively, RAPID [146] inverts the paradigm. Using Gossip [170], RAPID disseminates packets that it has received. Interested parties can then request packets be forwarded on them if required. No direct comparison is available between RAPID and MISTRAL, though it can be concluded both are more energy efficient than traditional probabilistic flooding.

Directional flooding

Rather than probabilistic or bounded means, other solutions have explored directing floods across portions of a network. Kum et al. [195] introduce AODV-DF that can be considered an flood augmented version of AODV. Devices periodically broadcast hello packets towards

a gateway node, allowing the gateway node to obtain the logical layout of the network (i.e. number of hops between nodes). Informed by this logical layout, packets can then be efficiently routed across the network. In AODV-DF nodes send less packets and combined with gains from directed routing, leads to greater energy efficiency. But of course this brings topology and layout constraints to the network, not suited to all types of applications.

Recognising fixed infrastructure as a weakness, Kim et al. [186] create a Semi-directional Flooding (SDF) routing protocol to remove fixed infrastructure from AODV-DF. The authors apply a bounded depth tree-based search to discover routes to local nodes. The approach significantly reduces the number of repetitions required to construct a route, and constrains the search domain to local nodes. Of course, when routes are involved with dynamic networks, approaches are required to prune stale routes and detect new routes to disconnected nodes. The authors apply SDF reducing the number of packets for route recovery.

Concurrent Flooding

Not addressed by the prior work discussed above is the issue of radio on time. Radio on time is the measure by which the radio is in an active receive or transmit mode. The higher the radio on time, the higher the energy consumption. Flooding-based approaches avoid such a metric because for good reliability, nodes are left in receive mode when not transmitting. However, consider a network where receivers power down their transceiver after a packet has been received. Nodes at the centre will power off first, and the nodes at the extreme of the network last. The time taken for outer nodes is variable and depends on when nodes choose to transmit. Therefore it has been the goal of some to reduce radio on time, by minimising the time taken for a message to propagate through a network. As packets are repeated some time after a node has received a packet, there is a chance that two nodes will repeat the same packet at different times. As pointed out in Section 2.5.1, this may result in degradation or complete cancellation of the radio signal, reducing network reliability.

One way to reduce radio on time and improve reliability is through the use of *concurrent transmissions*. Concurrent transmissions aim to transmit the same data at the same time, amplifying the signal (as alluded to in Section 2.5.1). Glossy [148, 149] and Low-power Wireless Bus were the first works to combine concurrent transmissions and flooding. Figure 3.16 demonstrates the basic flooding principle without radio on time optimisations. To prevent multiple devices from initiating a flood simultaneously, Glossy requires a master device to maintain a network clock and dictate the device transmission schedule through regular beacons. Each node synchronises with the network clock and schedule and transmits during their slot. Each packet contains a TTL, identical to bounded flooding protocols, which

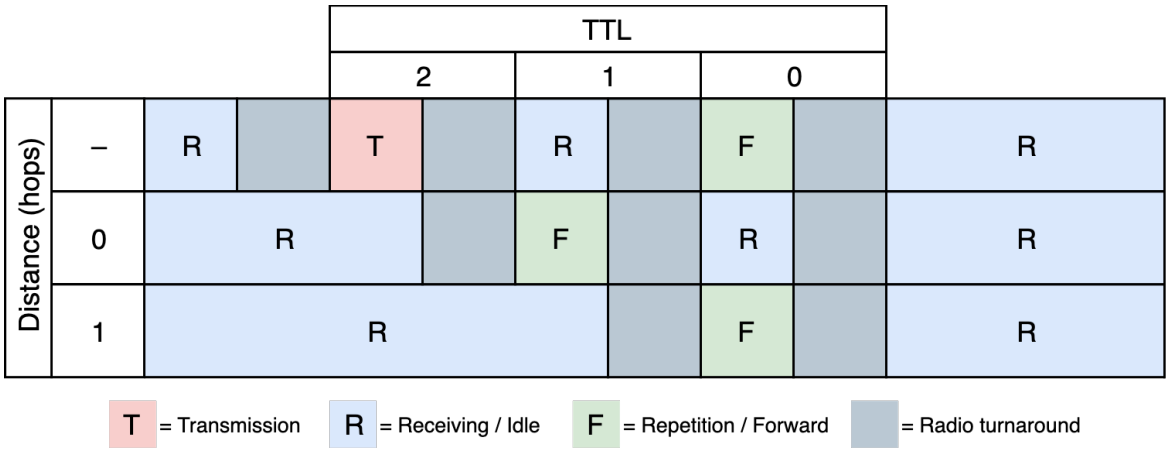

Fig. 3.16 Visualisation of a Glossy round.

is decremented upon retransmission. The authors reported high reliability after applying glossy to 802.15.4 radios operating at 250 kilobits a second.

Splash [144] improves on the original work of Glossy. It improves reliability by varying the number of re-transmitters per flood, as more re-transmitters can degrade transmissions. Splash also distributes retransmissions across multiple channels, and conducts a channel assignment phase to ensure even distribution of nodes and that each node receives each packet. An approach called 'opportunistic overhearing' causes nodes to swap channels upon detecting an error, where a future transmission of the missed packet is guaranteed. Each packet is encoded using an XOR encoding scheme giving devices a higher likelihood packet recovery in the case of sequential reception errors.

Ripple [292] points out that repeated transmission of the same data is often redundent. Instead, Ripple uses multiple channels to transmit different packets, improving network throughput. All packets are repeated on other channels to ensure nodes receive all packets. Whereas Splash requires a channel assignment phase, Ripple does not and instead distributes nodes across channels automatically. Even though Ripple focuses on throughput over reliability, it still achieves over 99% by adopting forward error correction.

Whisper [119] points out that swapping between receiving and transmitting during a flood is inefficient. In Whisper, devices retransmit packets concurrently once received until a maximum count is reached. A maximum count is used instead of a TTL due to software limitations; software cannot modify packets and reliably retransmit packets at the required timing tolerance.

Blueflood [93] applies concurrent transmissions to Bluetooth Mesh as an alternative to the bounded flooding approach currently used by the protocol. Blueflood is the first protocol that applied concurrent transmissions to commodity Bluetooth hardware. The authors provide positive results, and an expanded version of the paper [225] profiles radio wave propagation, aligning with the properties of radio waves described in Section 2.5.1.

3.3.5 Analysis

Wireless ad-hoc networking protocols can be used to create distributed applications that operate across vast spaces. Each ad-hoc networking protocol is designed for a specific purpose, but the way a protocol propagates its packets informs its memory and energy consumption, its support for interactivity, and its infrastructure and configuration complexity.

Routing-based protocols propagate packets via directed paths across a network. These paths are either computed *statically* or *dynamically*. Statically routed protocols define fixed routes between devices and require infrastructure to support their operation. Route definition requires manual configuration, leading to a high configuration complexity. Dynamically routed protocols on the other hand compute routes dynamically, allowing devices added or removed from a network with ease. This flexibility is enabled by fixed infrastructure devices that have more memory and a constant power to maintain routes between nodes. Routes also require regular maintenance, reducing the throughput of the network. Dynamically routed protocols therefore consume more memory and require infrastructure and configuration to support their operation.

Flooding-based protocols require each device retransmit a packet until it has been propagated across a network. With no routes to maintain, flooding-based protocols generally require less memory and less configuration to operate. They do however come with increased energy consumption through sometimes redundant packet retransmission. Seeing the value of flexible ad-hoc networking however, much existing work has looked at ways to optimise the energy consumption of flooding-based approaches.

Bounded flooding protocols, for example, limit the number of retransmissions to optimise energy consumption. Communication between nodes is therefore proximal, reducing network flexibility. Many of the bounded flooding protocols discussed in this section also retransmit packets from different devices, sometimes simultaneously. Simultaneous retransmission of different data creates more interference, reducing the reliability of flooding-based approaches and wasting energy.

	Packet Propagation	Physical standard	Energy consumption	Memory consumption	Configuration Complexity (WN1	Infrastructure free (WN2)	Supports interactivity (WN3
Glossy	Concurrent Flooding	802.15.4	Low	Low	Low	No	Yes
BLE Mesh	Bounded Flooding	BLE	Low	Low	Low	Yes	No
Zigbee	Dynamic Routing	802.15.4	Low	Moderate	Moderate	No	Yes
MISTRAL	Probabilistic Flooding	–	Low	Moderate	Moderate	Yes	No
SDF	Directional Flooding	–	Moderate	Moderate	Low	Yes	No
AODV-DF	Directional Flooding	–	Moderate	Moderate	Low	No	No
Thread	Dynamic Routing	802.15.4	Low	Moderate	Moderate	No	Yes
Bluetooth Mesh	Bounded Flooding	BLE	High	High	High	No	Yes
RT-BLE	Static Routing	BLE	Low	High	High	Yes	Yes
Droplet	Concurrent Flooding	BLE	Low	Low	Low	Yes	Yes

Table 3.4 A summary of the wireless protocols discussed in this section. Protocols are profiled by the design requirements (WN1—WN3) contained in IQ3.

Probabilistic flooding protocols optimise energy consumption by using probability to limit the number of retransmissions. Upon reception of a packet, if it is likely nearby devices have already received a packet, that packet is not retransmitted. This approach does however result in missed packets, and these protocols compensate for decreased reliability by including more metadata in packets. Metadata allows devices to request packets that were not forwarded, which also means devices must retain copies of packets. Probabilistic approaches therefore increase memory consumption for individual nodes and reduce overall energy consumption but not uniformly across the network.

Directional flooding protocols optimise energy consumption by directing floods across a network. Direction is obtained from mapping the network topology, and these protocols therefore make use of dynamic routing protocols. As noted before however, dynamic routing protocols increase memory and energy consumption through regular route maintenance.

Concurrent flooding protocols increase network throughput and reliability by repeating packets simultaneously. The use of concurrent transmissions reduces overall radio on time leading to more optimal energy consumption. Concurrent flooding protocols also typically incorporate a scheduler that allows only one device to initiate a flood at a time, thereby improving reliability. The scheduler divides time into discrete slots and a fixed infrastructure device is required to allocate and synchronise devices to a common network clock. Concurrent flooding protocols therefore require infrastructure to operate.

Table 3.4 summarises this discussion and profiles ad-hoc wireless networking protocols based on the design requirements (WN1—WN3) contained within IQ3. We place Droplet, one of the contributions of this thesis, in context.

3.4 Summary

This chapter has explored specific questions (IQ1–IQ3) derived from extensive discussion in Chapter 2. The upcoming sections map each question onto its corresponding area of physical computing (programming, hardware composition, wireless networking) where we answer each question.

3.4.1 Programming

IQ1 *Are there any programming languages/environments for microcontrollers that support installation-free (P3), event-based (P2), visual programming (P1)? Do any such environments support these features without compromising memory and processor efficiency?*

In search of an answer to IQ1, Section 3.1 explored programming languages and environments for microcontrollers. We found that efficiency is particularly important for processor and memory constrained microcontrollers and that compiled programming languages, which generate machine code, offer the greatest efficiency. Compiled languages however are often harder to use.

Higher-level languages offer a more intuitive development experience, but they sacrifice processor and memory efficiency. This is because they interpret pre-generated byte code or program text at runtime *on the microcontroller*. Lower processor efficiency leads to higher energy consumption and less real-time applications.

There are highly intuitive visual programming languages that exist for microcontrollers. However, these languages typically build on higher-level programming languages and observe their associated costs. Some do build on C/C++ HALs, but they do not offer the event-based paradigm recognised as beneficial to citizen developers.

We therefore conclude that, to the best of our knowledge, there are no visual programming environments for microcontrollers that support installation-free, event-based programming whilst not compromising on memory and processor efficiency.

3.4.2 Hardware composition

IQ2 *Are there any low-infrastructure wired protocols for hardware composition that support dynamic connectivity (HC1), device discovery (HC2), and hardware abstraction (HC3)? Are any such protocols as widely supported by microcontrollers as I2C or SPI?*

Seeking an answer to IQ2, in Section 3.2 we explored wired protocols and their use across toolkits for easier hardware composition. Across these toolkits we found that many tried to make existing protocols like I2C and SPI *more dynamic*. We also recognised a trend towards reducing the amount of infrastructure (individual cables) for these protocols to operate.

There are a number of existing and emerging wireless protocols that provide hardware abstraction and support dynamic connectivity and detection. These features however appear to come with an increase in implementation complexity, requiring additional silicon to operate. They are therefore not widely supported by microcontrollers.

We therefore conclude that, to the best of our knowledge, there are no widely supported low-infrastructure wired protocols with dynamic connectivity, device discovery, and hardware abstraction.

3.4.3 Wireless networking

IQ3 *Are there any wireless ad-hoc networking protocols that require no configuration (WN1) and no infrastructure (WN2) to operate? Are any such protocols able to support interactive applications (WN3) without sacrificing energy efficiency?*

To answer IQ3, Section 3.3 explored existing wireless ad-hoc networking protocols, profiling them by their packet propagation technique. We found that routing-based protocols generally require more memory, more configuration, and infrastructure to operate. These overheads make networking less intuitive for citizen developers

Flooding-based protocols, on the other hand, require minimal memory and no configuration to operate. Because protocols generally repeat packets in sequence, packet propagation is less supportive of interactive applications and requires more energy than routing-based protocols. Concurrent flooding protocols support interactivity and offer greater energy efficiency by propagating packets simultaneously. Existing concurrent flooding protocols however require infrastructure to operate.

We therefore conclude that, to the best of our knowledge, there are no ad-hoc networking protocols that are memory and energy efficient, whilst being configuration and infrastructure free.

Chapter 4

CODAL: intuitive microcontroller programming

Across Chapters 2 and 3 we identified properties of existing technologies that make programming more intuitive:

P1 *Visual programming*: Higher level programming languages, and in particular visual programming languages, prove more intuitive to citizen developers when compared to low-level text-based programming languages like C/C++ [114].

P2 *Event-based programming*: Event-based programming is used across the programming language spectrum. Its use, especially in visual programming languages, has been shown to be more intuitive to citizen developers [211, 212, 276].

P3 *Installation-free*: Environments that require software installation make programming inaccessible to some. Installation-free programming environments allow all citizen developers to access programming.

Resource constrained microcontrollers, and their use in battery-powered and embedded physical computing devices, demand efficiency not commonly seen in programming languages with the properties above. Limited flash and RAM requires programming languages to be memory efficient, and battery powered operation requires programming languages processor efficient so not to unnecessarily expend energy.

As we concluded in Chapter 3, this combination of features is not common amongst the literature and we found that there was no programming language or environment that offered visual, event-based, installation-free programming, whilst being processor and memory efficient.

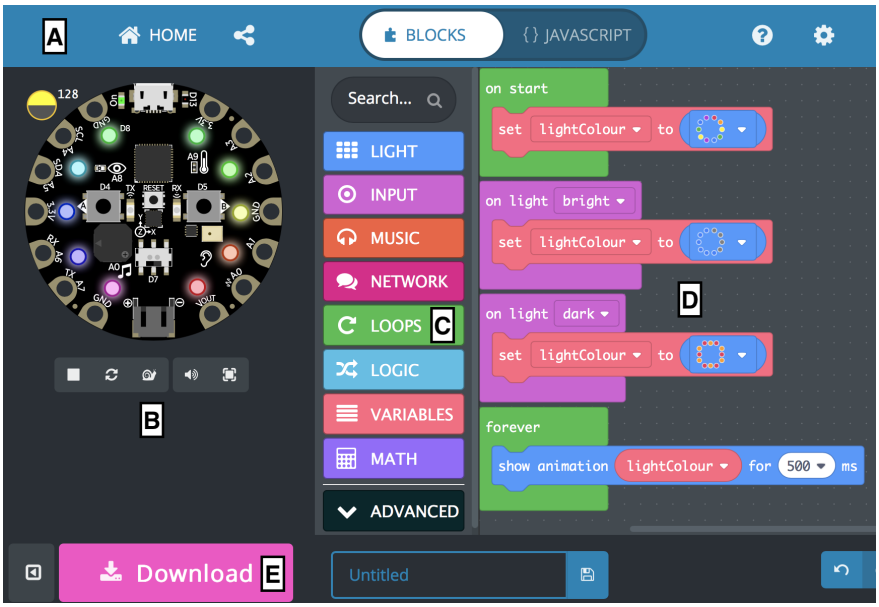

Fig. 4.1 Screenshot of the MakeCode web app for the Circuit Playground Express (CPX).

This chapter describes the design, implementation, and evaluation of CODAL, a lightweight C++ runtime environment for microcontrollers. CODAL supports the Microsoft MakeCode programming environment, which combined, creates an installation-free, event-based, visual programming environment for microcontrollers. CODAL supports applications written in MakeCode to deliver memory and processor efficiency, and as a result, applications are up to 50 times more performant than other solutions in the space. MakeCode and CODAL and see over one million active users every month.

We begin in Section 4.1 by describing Microsoft MakeCode, which is vital to understanding the role of CODAL, but is not a direct contribution of this thesis. We then go onto describe the core components and features of the CODAL runtime (Section 4.2), and the build system used to build CODAL applications (Section 4.3). In Section 4.4, we discuss how CODAL APIs are propagated to MakeCode, followed by a systems evaluation of both MakeCode and CODAL in Section 4.5. We then provide examples of how MakeCode and CODAL has already been used by citizen developers in Section 4.6 before providing a summary of the chapter in Section 4.7.

4.1 Microsoft MakeCode

Microsoft MakeCode (Figure 4.1) is a web-based programming environment for resource constrained microcontrollers. From the visual programming language, to the no-installation

web-based programming environment, to the in-browser compiler, every facet of MakeCode is designed to democratise access to programming physical computing devices. MakeCode gives users the familiar event-based semantics of Scratch with the means to progress to text-based programming and the power to explore every feature of a device without the use of low-level programming languages. To allow for quicker development of code and democratise access to less-privileged demographics, an in-browser device simulator executes user programs as they are composed within the editor. The editor can also be used offline through browser caching mechanisms and a separate desktop application. Though just one instance of the MakeCode editor for the Circuit Playground Express (CPX) is shown in Figure 4.1, MakeCode instances exist for a variety of physical computing devices including the BBC micro:bit.

Stepping through Figure 4.1 in more detail, the web app has five sections: (A) the menu bar allows switching between visual and text-based programming editors; (B) a physical computing device simulator for the CPX that provides feedback on user code as it is composed; (C) the toolbox provides access to device-specific APIs and programming elements; (D) the visual programming canvas where code is composed; and (E) the download button that invokes an in-browser compiler, producing a binary executable that is downloaded to the users' filesystem.

Transferring binary executables to the physical device is also simple and requires no installation. When plugged into a PC, MakeCode devices appear as a USB flash drive. After downloading the file locally, users simply use a file-copy operation to program the target device.

4.1.1 In-browser program compilation

The architecture of the MakeCode web application is encapsulated in Figure 4.2. Programming editors for Blockly [156], a visual programming language, and static TypeScript, a text-based superset of JavaScript, allow users to easily build applications for physical computing devices. As programs are composed, editors generate Static TypeScript (STS). STS is converted to an Intermediate Representation (IR) for simulation by the in-browser device simulator and for compilation by the in-browser compiler. The in-browser compiler produces a user binary which is later combined with a pre-compiled *runtime binary* when the download button is clicked. The pre-compiled runtime binary contains wrappers/foreign function interfaces for binding typescript applications to CODAL, an efficient C++ runtime for resource constrained microcontrollers.

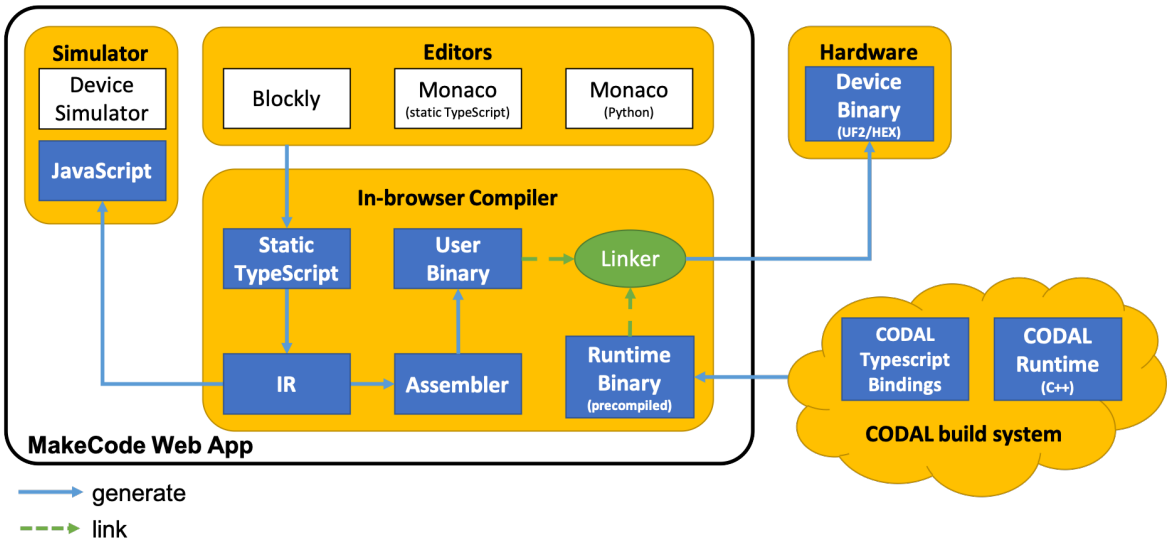

Fig. 4.2 MakeCode and CODAL program compilation.

4.2 The CODAL runtime

CODAL is a lightweight, object-oriented, componentised C++ runtime for microcontrollers designed to provide an efficient abstraction layer for higher level languages, such as JavaScript and Blocks. CODAL has six key elements:

1. *a unified eventing subsystem* (common to all components) that provides a mechanism to map asynchronous hardware and software events to event handlers;
2. *a non-preemptive fiber scheduler* that enables concurrency while minimizing the need for resource locking primitives;
3. *a simple memory management system* based on reference counting to provide a basis for managed types;
4. *a stream processing framework* based on a composable, receiver-driven component model;
5. *a set of drivers*, that abstract microcontroller hardware components into higher level software components, each represented by a C++ class;
6. *a parameterised object model* composed from these components that represents a physical device.

There are discussed in detail below.


```
1  #include "CircuitPlayground.h"
2  CircuitPlayground cplay;
3
4  void onBright() { // user defined code }
5  void onDark() { // user defined code }
6
7  int main() {
8      cplay.messageBus.listen(ID_LIGHT_SENSOR,
           LIGHT_THRESHOLD_HIGH, onBright);
9      cplay.messageBus.listen(ID_LIGHT_SENSOR,
           LIGHT_THRESHOLD_LOW, onDark);
10 }
```

Fig. 4.3 An example MakeCode application for the CPX to detect the brightness level of a room (left); and a representative C++ application that demonstrates the message bus, the enabler of event-based programming in MakeCode (right).

4.2.1 Message bus and events

CODAL offers a simple yet powerful model for handling hardware or user defined events. Events are modeled as a tuple of two integer values - specifying an *id* (namespace) and a *value*. Typically, an id correlates to a specific software component, which may be as simple as a button or something more complex as a wireless network interface. The value relates to a specific event that is unique within the id namespace. All events pass through the CODAL message bus. Application developers can then *listen* to events on this bus, by defining a C/C++ function to be invoked when an event is raised. Events can be raised at any time simply by creating an *Event* C++ object, which then invokes the event handlers of any registered listeners. Figure 4.3 shows a MakeCode Blocks application for the CPX, and a representative C++ snippet to demonstrate how MakeCode uses the CODAL message bus to enable event-based programming.

Unlike simple function pointers, CODAL event handlers can be parameterised by the event listener to provide decoupling from the context of the code raising the event. The receiver of an event can choose to either receive an event in the context of the fiber that created it, or can be decoupled and executed via an Asynchronous Procedure Call (APC). The former provides performance, while the latter provides decoupling of low level code (that may be executing, say, in an interrupt context) from user code. Each event handler may also define a threading model, so they can be reentrant or run-to-completion depending upon the semantics required.

4.2.2 Fiber scheduler

CODAL provides a *non-preemptive* fiber scheduler with asynchronous semantics and a power efficient implementation. CODAL fibers can be created at any time but will only be descheduled as a result of an explicit call to yield(), sleep() or wait_for_event() on the message bus. The latter enables condition synchronization between fibers through a wait/notify mechanism. A round-robin approach is used to schedule runnable fibers. If at any time all fibers are descheduled, microcontroller hardware is placed into a power efficient sleep state.

The CODAL scheduler makes use of two novel mechanisms to optimize for microcontrollers. Firstly, CODAL adopts a stack paging approach to fiber management. Microcontrollers do not support virtual memory and are heavily RAM constrained, but relatively cycle rich. Therefore, instead of overprovisioning stack memory for each fiber (which would waste valuable RAM), we instead dynamically allocate stack memory from heap space as necessary and copy the physical stack into this space at the point at which a fiber is descheduled (and similarly restored when a fiber is scheduled). This copy operation clearly incurs a small CPU overhead, but brings greater benefits of RAM efficiency - especially given that microcontroller stack sizes are typically quite small (~200 bytes is typical).

Secondly, the CODAL scheduler supports transparent Asynchronous Procedure Calls (Asynchronous Procedure Call (APC)s). Any function can be *invoked* as an APC. Conceptually, this is equivalent to calling the given function in its own fiber. However, the CODAL runtime provides a common-case transparent optimization for APCs we call *fork-on-block* - whereby a fiber will only be created at the point at which the given function attempts a blocking operation such as sleep() or wait_for_event(). Functions which do not block therefore do not incur all of the context switch overhead.

When invoking an APC, the scheduler snapshots the current processor context and stack pointer (but not the whole stack). If the scheduler is re-entered before the APC completes, a new fiber context is created at the point of descheduling, and placed on the appropriate wait queue. The previously stored context is then restored, and execution continues from the point at which the APC was first invoked. This mechanism provides potentially high RAM savings for the processing of message bus event handlers in particular.

CODAL's scheduling and eventing models are shared by both high and low level languages, and therefore handled uniformly. As a result, when a foreign function call is mapped to C++, that C++ function is capable of blocking the calling fiber without infringing on the concurrency model of the higher level language. This enables, for example, a C++ device

driver to block a JavaScript program when awaiting data without changing the behavior of other JavaScript code acting asynchronously (as in Figure 4.1).

4.2.3 Memory management

CODAL implements its own lightweight heap allocator, introducing reentrant versions of the libc malloc family of functions, permitting universal access to heap memory in user or interrupt code. The heap allocator is flexible and reconfigurable, allowing the specification of multiple heaps across memory and it is optimised for repeat allocations of memory blocks that are commonplace in embedded systems.

CODAL also makes use of simple managed types, built using C++ reference counting mechanisms. C++ classes are provided for common types such as strings, images, and data buffers. A generic base class is also provided for the creation of other managed types. This simple approach brings the benefits of greater memory safety for application code, but with the expense of suffering from the issues related to circular references. We take the view that such scenarios are rare in microcontroller applications, justifying this approach over a more complex garbage collection scheme and its overhead.

4.2.4 Streams

Whilst some embedded systems applications only process discrete data, many operate on data flows. Examples include digital signal processing (DSP), audio recording/playback, and gesture and voice recognition. CODAL defines a standardised mechanism to efficiently handle data streams that can be then exported to higher level languages as a data flow model. Following on from the Object Oriented approach adopted by CODAL, data streams are modelled using two well defined C++ abstract base classes - *DataSource* and *DataSink*, as defined in Figure 4.4. These two simple interfaces contain the minimal set of functionality to enable simple, efficient, safe and extensible stream processing. As can be seen, all data is conveyed using a managed type (ManagedBuffer). This ensures that any device drivers that source streamed data or DSP software or application code that process it need not be burdened with the detail of memory management.

Any component capable of generating stream data (an *upstream* component) implements the DataSource interface, and likewise a consumer (*downstream* component) implements DataSink. A receiver driven approach to data streaming is adopted. DataSinks register with their upstream DataSource through an explict call to the connect method. When data is subsequently generated by a DataSource, it then executes the pullRequest() method on its

```
1  class DataSink
2  {
3      public:
4      virtual int pullRequest();
5  };
6
7  class DataSource
8  {
9      public:
10     virtual ManagedBuffer pull();
11     virtual void connect(DataSink &sink);
12 };
```

Fig. 4.4 CODAL Stream Interfaces.

downstream component to indicate that data is available for processing, which can then be drained from the DataSource through its pull() method. This receiver driven approach allows for simple flow control to be implemented - placing the control of precisely when and how processing takes place in the hands of the application.

Components may be both a DataSource and a DataSink, allowing the creation of a data flow graph of components. The CODAL runtime contains reusable components that can be composed to undertake common operations. For example, analog to digital converter modules act as DataSources and digital to analag converter modules as DataSinks. A *Mixer* component allows multiple DataSources to be aggregated into one DataSink (with clear applications for audio processing). A *LevelDetector* provides configurable smoothing and thresholding functionality with asynchronous outputs reported through the CODAL message bus. A *DataStream* provides user configurable first-in-first-out buffering functionality, and a *Synthesizer* frequency and phase shifted datastreams based on a user defined waveform template. For example, Figure 4.5 illustrates an event-based, direct-memory-access enabled, asynchronous sound level detector of a live data source from an attached microphone.

4.2.5 Device driver components

CODAL drivers abstract away the complexities of the underlying hardware into reusable, extensible, easy-to-use components. For every hardware component there is a corresponding software component that encapsulates its behavior in a C++ object. CODAL has three types of drivers:

1. A hardware agnostic abstract specification of a driver model (e.g. a Button, or an Accelerometer). This is provided as a C++ base class.

```
1  CircuitPlayground cplay;
2  SAMD21DMAC dmac;
3  SAMD21PDM microphone(cplay.io.microphoneData,
       cplay.io.microphoneClock, dmac, 10000);
4  LevelDetector level(microphone.output, 70, 30);
5
6  void onLoud(DeviceEvent)
7  {
8      cplay.serial.printf("LOUD\n");
9  }
10
11 int main()
12 {
13     microphone.enable();
14     cplay.messageBus.listen(DEVICE_ID_SYSTEM_LEVEL_DETECTOR,
15         LEVEL_THRESHOLD_HIGH, onLoud);
16 }
```

Fig. 4.5 CODAL Stream processing example - a sound level detector.

2. The concrete implementation of the abstract driver model, which is typically hardware specific. This is implemented as a subclass of a driver model, such as a LIS3DH accelerometer, as manufactured by ST Microelectronics.

3. A high level driver that relies only on the interfaces specified in a driver model (e.g. a gesture recognizer based on an Accelerometer model).

This approach brings the benefits of abstraction and reusability to CODAL, without losing the hardware specific benefits seen in flat abstraction models where every microcontroller is made to look the same, even though their capabilities are different (as in the Arduino and mbed APIs, for example).

Finally, we group together the components of a physical device to form a *device model*. This is a singleton C++ class that, through composition of device driver components, provides a configured representation of the capabilities of a device. Such a model allows: an elegant OO API for programming a device, and a static representation that forms an ideal target for the MakeCode linker to bind high level STS interfaces to low level optimised code.

An example device model for the CPX is shown in Figure 4.6 for reference.

MakeCode is further supported by an annotated C++ library (*MakeCode wrappers*) defining the mapping from CODAL to TypeScript and Blockly. The use of MakeCode wrappers ensures that different MakeCode targets that use CODAL share a common TypeScript and block API vocabulary[1].

[1] See https://github.com/microsoft/pxt-common-packages

```
1  class CircuitPlayground : public CodalDevice {
2    public:
3      message bus                 messageBus;
4      CPlayTimer                  timer;
5      SAMD21Serial                serial;
6      CircuitPlaygroundIO         io;
7      Button                      buttonA;
8      SAMD21I2C                   i2c;
9      LIS3DH                      accelerometer;
10     NonLinearAnalogSensor       thermometer;
11     AnalogSensor                lightSensor;
12     ...
```

Fig. 4.6 Device Model for the Adafruit CPX

4.3 Building CODAL applications

As shown in the previous section, CODAL provides users with powerful yet simple C++ abstractions for writing applications for microcontrollers. However, a compilation process must take place to turn C/C++ applications into binary instructions for microcontrollers. For this, we designed a custom build system.

The CODAL build system is designed for modularity in a similar way to the many package systems seen in higher level languages like JavaScript (i.e. node modules). The key difference being of course that the CODAL build system allows the modular composition C/C++ libraries to create C/C++ applications. Modularity enables the efficient reuse of libraries across different applications.

Because libraries can change and develop over time, the CODAL build system incorporates versioning. Versioning allows applications to target specific versions of libraries for reproducible builds. This is especially important when supporting a user facing application like MakeCode where known working, reproducible builds are a must.

The CODAL build system is also designed to make C/C++ programming more accessible to less experienced embedded developers. Only three tools, that work across the main operating systems (Windows, Mac OS, Linux), are required to build a CODAL application:

1. *CMake [188]*—for configuration of compilers and linkers and gathering files for compilation.

2. *A compilation toolchain*—to compile and link detected files for the desired microcontroller (applications for most ARM-based microcontrollers can be compiled by the *arm-none-eabi-gcc* [96] suite).

```
{
    "device":"CIRCUIT_PLAYGROUND",
    "processor":"SAMD21G18A",
    "architecture":"CORTEX_M0_PLUS",
    "toolchain":"ARM_GCC",
    "post_process":"python ...",
    "generate_bin":true,
    "generate_hex":true,
    "config":{
        "CODAL_TIMESTAMP":"uint64_t",
        "USB_MAX_PKT_SIZE": 64,
        ...
    },
    "definitions":"-DSAMDX1 -D__SAMD21G18A__",
    "cpu_opts":"-mcpu=cortex-m0plus -mthumb",
    "asm_flags":"-fno-exceptions -fno-unwind-tables
      --specs=nosys.specs",
    "c_flags":"-std=c99 --specs=nosys.specs",
    "cpp_flags":"-std=c++11 ...",
    "linker_flags":"-Wl,--no-wchar-size-warning...",
    "libraries":[
        {
            "name":"codal-core",
            "url":"https://github.com/lancaster-university/codal-core",
            ....
    ]
}
```

Fig. 4.7 The CODAL target JSON file for the CPX.

3. *Git [135]*—to provide version control across all files.

The entire CODAL build system is defined in a single Git repository hosted on Github [61] (https://github.com/lancaster-university/codal). This repository is where users write applications for CODAL and invoke the build system to produce binaries for embedded development boards. The repository contains a CMake file in its root which defines the build process for an application and contains functions for parsing JSON files and cloning Git repositories.

As APIs are closely bound to specific development boards such as the CPX or BBC micro:bit, users must select their development boards before applications can be compiled. In the CODAL build system, specific development boards are called *targets* and targets are selected by way of adding a JSON file to the build system repository. This JSON file references a downloadable target library.

Target libraries are also git repositories and they contain a single JSON file that defines parameters for linking and compilation, CODAL runtime configuration, and version locked references to any supporting libraries. Specifying such parameters in JSON makes customis-

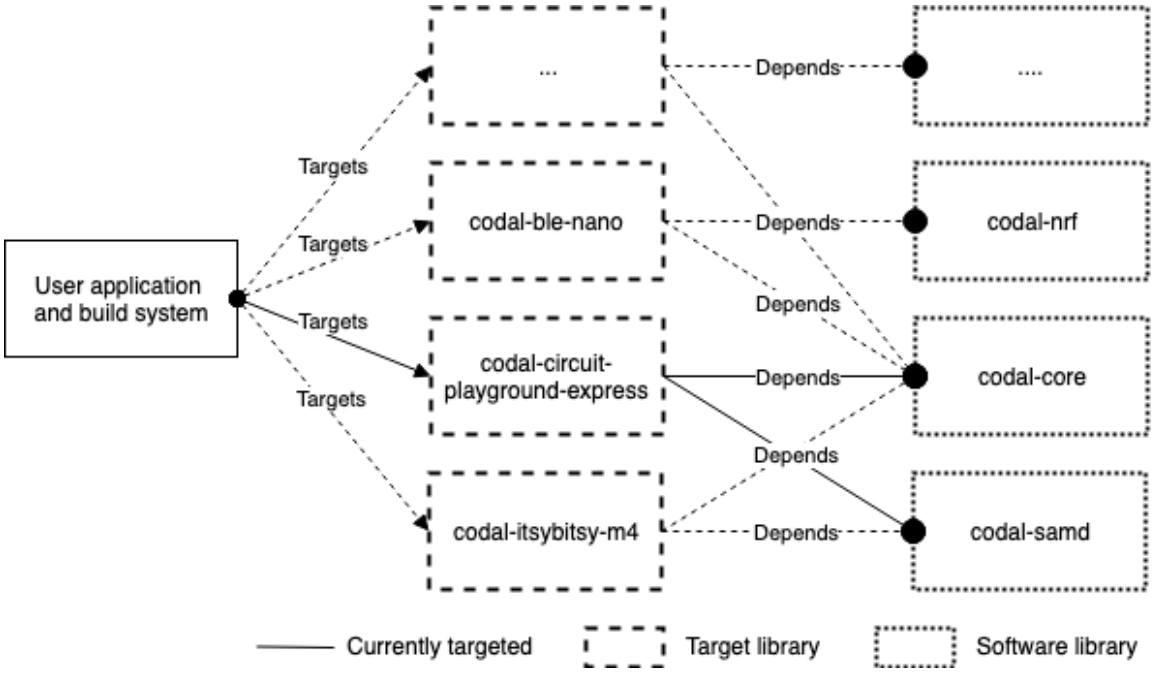

Fig. 4.8 The CODAL build architecture where each box represents a git repository. A solid box represents the root build repository where user applications and the main build system reside, a dashed box represents a target library, and a dotted box represents a standard library.

ing the compiler toolchain more accessible. Targets also contain the device driver model and additional APIs that are derived from supporting software libraries. These software libraries are also git repositories, and they typically contain supporting drivers and abstractions used by target libraries; the CODAL runtime, for example, is a software library.

Targets can be specified and the build system invoked from an optional python helper script. During the build process, CMake downloads the target libraries and all of its software dependencies. Once all libraries have been downloaded, each is added to the CMake build tree to resolve dependencies and determine the compilation and linking order. As all libraries are just Git repositories, no infrastructure investment is required and developers can use existing services like Github to host CODAL libraries.

Figure 4.8 shows the dependency relationship of some of the many targets available in the CODAL ecosystem. As can be seen in the figure, all targets depend on *codal-core*, in addition to platform dependent libraries. For instance, the codal-circuit-playground-express target depends on *codal-core* and *codal-samd*, a library that contains specific hardware drivers for SAMD21/SAMD51 processors.

4.4 From MakeCode to CODAL

MakeCode supports a simple foreign function interface from STS to C++ based on namespaces, enumerations, functions, and basic type mappings. Functions and typings typically come from CODAL, and the resulting C++ files take the place of conventional user applications in the CODAL build system.

MakeCode uses top-level namespaces to organize sets of related functions and these top-level namespaces are mapped to toolbox categories in the MakeCode editor (as seen in Figure 4.1C). Preceding a C++ namespace, enumeration, or function with a comment starting with *//%* indicates that MakeCode should map the C++ construct to STS. Within the *//%* comment, attributes specify the visual appearance for that language construct, such as for the input namespace in C++ for the CPX:

```
1 //% color="#B4009E" weight=98 icon="\uf192"
2 namespace input { ...
```

Mapping of functions and enumerations between C++ and STS is straightforward and performed automatically by MakeCode. For example, the following C++ function *onLightConditionChanged* in the namespace *input* wraps the more complex C++ needed to update the sensor and register the (Action) handler with the underlying CODAL runtime:

```
1 //% block="on light %condition"
2 void onLightConditionChanged(LightCondition condition, Action handler) {
3     auto sensor = &getWLight()->sensor;
4     sensor->updateSample();
5     registerWithDal(sensor->id, (int)condition, handler);
6 }
```

MakeCode generates a TypeScript declaration file (here called a shim file) to describe the TypeScript elements corresponding to C++ namespaces, enumerations and functions. Since the C++ function above is preceded by a *//%* comment, MakeCode adds the following TypeScript declaration to the shim file and copies over the attribute definitions in the comment. MakeCode also adds an attribute definition mapping the TypeScript shim to its C++ function:

```
1 //% block="on light %condition"
2 //% shim=input::onLightConditionChanged
3 function onLightConditionChanged(condition: LightCondition, handler: ()
    => void): void;
```

Since the *//%* comment also contains a *block* attribute, MakeCode creates a block (named "on light"), which can be seen in Figure 4.9. This process can be directly observed in any

Fig. 4.9 The resulting “on light” block defined using //%.

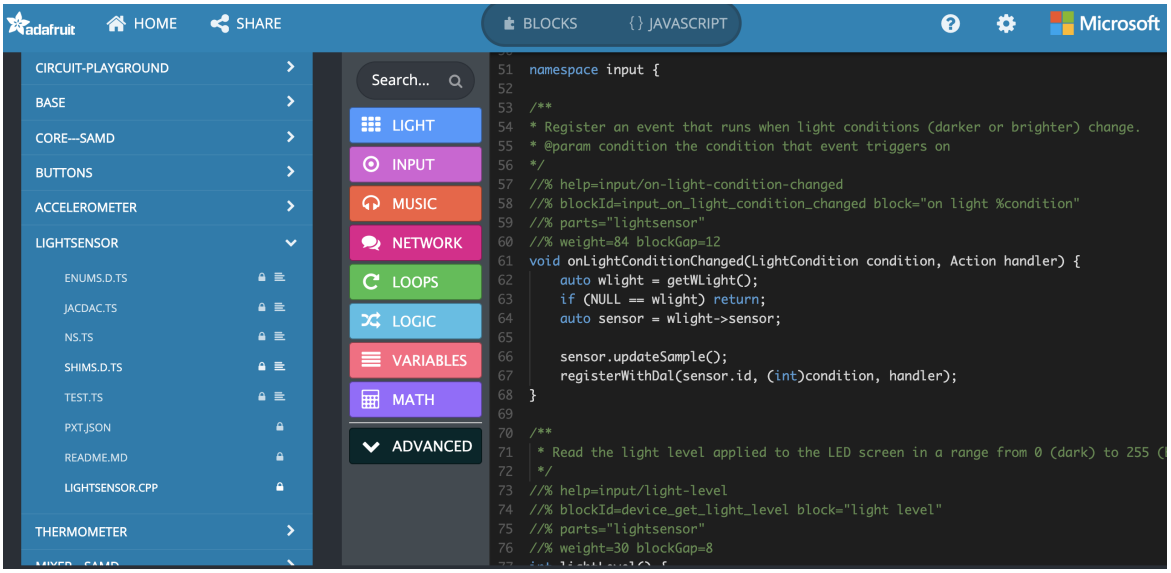

Fig. 4.10 The MakeCode C++ wrapping of the on light condition, compiled by the cloud compiler.

MakeCode editor. The Adafruit CircuitPlayground Express editor is given as an example (Figure 4.10).

To support the foreign function interface, MakeCode defines a mapping between C++ and STS types. Boolean and void have straightforward mappings from C++ to STS (bool $\rightarrow$ boolean, void $\rightarrow$ void). As JavaScript only supports number, which is a C++ double, MakeCode uses TypeScript’s support for type aliases to name the various C++ integer types commonly used for microcontroller programming (int32, uint32, int16, uint16, int8, uint8). This is particularly useful for saving space on 8-bit architectures such as the AVR. MakeCode also makes use of Managed Types from the CODAL runtime C++ types for strings, and uses the CODAL message bus to map STS lambdas and functions to C/C++.

4.5 Systems evaluation

Our platform has been actively deployed for over three years, bringing the benefits of a safe and intuitive programming environment for microcontrollers to millions of users. In this section we provide a broad, quantitative evaluation of the cost at which these benefits are realised. We do this with several micro-benchmarks that give insight into the performance of MakeCode and CODAL across the Uno, micro:bit, and CPX devices. We break down results by layer (CODAL and MakeCode) to give an insight into how each performs.

4.5.1 Benchmarks, devices, and methodology

To analyse the performance of our solution, we have written a suite of programs to evaluate different aspects of MakeCode and CODAL on a representative selection of real hardware devices. Throughout, we use the C++ CODAL benchmarks as a baseline; the STS benchmarks show the overhead added by MakeCode. These programs were written in both C++ and STS, and evaluated on three devices: The micro:bit (Nordic nRF51 microcontroller), the CPX (Atmel ATSAMD21 microcontroller), and the Uno (Atmel ATmega microcontroller).

The Uno is the simplest of these devices, consisting of an 8-bit processor running at 16 MHz, with only 2kB of RAM and 32kB of flash. The micro:bit has a 32-bit Cortex-M0 clocked at 16MHz, with 16kB RAM and 256kB of flash. The CPX is a 32-bit Cortex-M0+, which offers greater energy efficiency and performance; it clocks at 48 MHz, has 32kB of RAM and 256kB of flash. The benchmarks are classified into two types, each with their own methodology:

1. *Performance Analysis*: Tests that capture time taken to perform a given operation. For these benchmarks, we toggle physical pins on the device at key points in the test code. We then measure the time to execute the operation, by using a calibrated oscilloscope observing these pins. This allows us to derive highly accurate real time measurements without biasing the experiment.

2. *Memory Analysis*: Tests that capture the RAM or FLASH footprint of a certain operation. A map of memory is logged before and after the execution of an operation, allowing us to compute the cost. A serial terminal captures the output of these tests.

Note that memory and performance analysis are done in separate runs to ensure logging does not affect time-related measurements.

Table 4.1 A comparison of execution speed between: native C++ with CODAL; MakeCode compiled to native machine code; MicroPython; and Espruino. The first line lists the C++ time, while subsequent lines are slowdowns with respect to the C++ time.

	UNO	micro:bit	CPX
CODAL	171ms	102ms	31ms
MakeCode	2.4x	2.1x	7.3x
MicroPython	-	101x	183x
Espruino	-	1139x	-

4.5.2 Tight loop performance

To place the performance of MakeCode in context, we perform a comparative evaluation of MakeCode against two state-of-the-art solutions adopted by citizen engineers, using native C++ as our baseline. The two points of comparison are MicroPython [161], an implementation of Python for microcontrollers, and Espruino [289], an implementation of JavaScript for microcontrollers. For the CPX, a fork of MicroPython known as "CircuitPython" was used. Both MicroPython and Espruino use virtual machine (VM) approaches.

To give an indicative general case execution time cost of each solution, we created a simple program that counts from 0 to 100,000 in a tight loop in each solutions' respective language; the results are shown in Table 4.1. On AVR we count to 25,000 (to fit within a 16 bit `int`) and scale up the results.

For MicroPython and Espruino on the micro:bit, the run is *two or more orders of magnitude slower* than a native CODAL program. MakeCode performs only 2x slower. The slowdown reflects the simple code generator of the STS compiler. It should be noted that MakeCode for the CPX uses the tagged approach, which allows for seamless runtime switching to floating point numbers, resulting in a further 3x slowdown. For both devices, we can observe that MakeCode outperforms both the VM-based solutions of MicroPython and Espruino by at least an order of magnitude. MicroPython and similar environments cannot run on the Uno due to flash and RAM size limitations.

4.5.3 Context switch performance

To evaluate the performance of CODAL's scheduler we conducted a test that created two fibers, continuously swapped context, and measured the time taken to complete a context switch. We performed this test in both STS and C++ and the resulting profiles can be seen in Figure 4.11, which breaks the context switch down into three phases: (1) CODAL, the

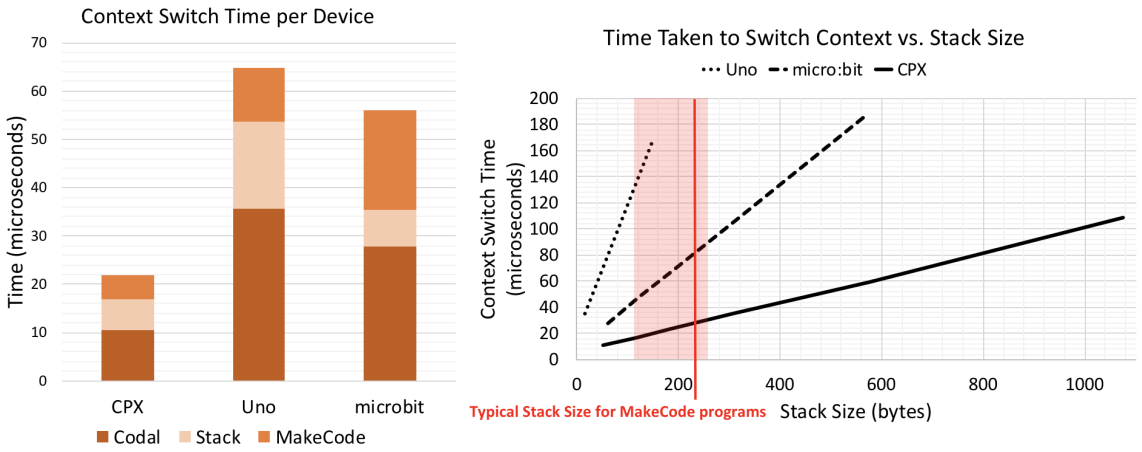

Fig. 4.11 Base context switch profiles per device (left); Time taken to perform a context switch against stack size (right).

time it takes to perform a context switch in CODAL; (2) Stack, the time taken to page out the MakeCode stack; and (3) MakeCode, the overhead added by MakeCode.

From these results, we observe that context switches generally take tens of microseconds. The cost of CODAL's stack paging approach can also be a significant, but not dominant cost. The cost of stack paging would of course grow with stack depth. Figure 4.11 profiles the time a context switch takes with an increasing stack size across all three devices in CODAL. This is similar to the previous test, except we placed bytes (in powers of 2) on the stack of each fiber, starting from 64 and finishing at 1024. The difference in gradients, and ranges of values can be put down to device capability. For instance, the Uno has an 8-bit word size, which means more instructions are required to copy the stack, this results in a steeper gradient than the other two devices. The vertical band indicates typical stack sizes for MakeCode programs based on a representative set of examples.

4.5.4 Performance of asynchronous operations

To gauge the cost of asynchronous operations in CODAL, we tested three commonly used code paths, designed to determine the efficiency of CODAL's *fork-on-block* Asynchronous Procedure Call (APC) mechanism that underpins all event handlers in MakeCode and CODAL. We measured the RAM and processor cost of: (1) creating a fiber; (2) handling a non-blocking APC call; and (3) handling a blocking APC call. We used the CPX for this experiment.

Table 4.2 Flash consumption of a MakeCode binary (kB)

	CPX	micro:bit	Uno
MakeCode	20.46	12.14	7.79
CODAL	29.85	34.35	13.7
Supporting Libraries	14.99	24.28	-
C++ Standard Library	43.14	24	1.03

Non-blocking APC calls, the best case, have a small overhead of 32 bytes of RAM and 4.01 microseconds of processing time. Blocking APC calls, the worst case, incur a large overhead of 204 bytes of RAM and 32.4 microseconds of processor time. Creating a fiber costs 136 bytes of RAM and 35.4 microseconds of processing time. These results highlight the performance gains of the opportunistic fork-on-block mechanism over a naive approach that would execute every event handler in a separate fiber.

4.5.5 Flash memory usage

Microcontrollerss make use of internal non-volatile flash memory to store program code. Table 4.2 shows the per device flash consumption of each software library used in the final MakeCode binary. To obtain these numbers, we analyzed the final map file produced after compilation. The ordering of the table aligns with the composition of the software layer: MakeCode builds on CODAL which builds on the C++ standard library and supporting libraries. MakeCode and CODAL consume 108 kB of flash, whereas CircuitPython consumes 201 kB, MicroPython consumes 228 kB, and Espruino consumes 142 kB of flash. This means that users can write sizeable applications in MakeCode, without the worry of running out of flash memory.

From the bottom up, the profile of the standard library changes dramatically for each device: The Uno has a very lightweight standard library; the micro:bit uses 64-bit integer operations (for timers) which requires extra standard library functions; and the CPX requires software floating point operations pulling in more standard library functions.

The size of CODAL and MakeCode scales linearly with the amount of functionality a device has, due to the component oriented nature of CODAL and transitively MakeCode. For instance, the Uno has few onboard components when compared to the CPX and micro:bit. The modular composition of CODAL allows us to support multiple devices with a variety of feature sets, while maintaining the same API at the MakeCode layer.

Table 4.3 Static RAM consumption of a MakeCode binary (kB)

	CPX	micro:bit	Uno
MakeCode	0.612	1.069	0.074
CODAL	0.369	0.214	0.156
Supporting Libraries	0.312	0.923	-
C++ Standard Library	0.161	0.149	0.074

4.5.6 RAM memory usage

Table 4.3 shows the per device RAM consumption of each software library used in the final MakeCode binary. To obtain these numbers, we analyzed the final map file produced after compilation. At runtime, MakeCode dynamically allocates additional memory: 1.56 kB for the CPX, 560 bytes for the micro:bit, and 644 bytes for the Uno. We also can see that in all cases, the RAM consumption of MakeCode and CODAL is well within the RAM available of each device.

MakeCode and CODAL consume a small amount of resources in comparison: CircuitPython (a derivative of MicroPython) consumes 12.8 kB, MicroPython consumes 9.5 kB, and Espruino consumes 5.3 kB of RAM. On the micro:bit, the Bluetooth stack requires 8 kB of RAM to operate. Due to MicroPython's RAM consumption this means that Bluetooth is inoperable. Comparatively, Espruino does enable the Bluetooth stack, but users have just ~300 bytes available for their programs due to the overhead incurred.

4.5.7 Extensibility

Adding a new device in CODAL is trivial once a microcontroller has been ported. The porting of a microcontroller is where we observe the largest development overhead, as low-level implementations of drivers for I2C, Serial, and SPI may have to be re-written. Due to CODAL's abstraction model, once low-level drivers have been implemented, drivers for higher level components like Accelerometers (which depend on high-level interfaces for low-level drivers) can be immediately adopted if hardware is present. A similar technique is used in MakeCode for simulators.

Fig. 4.12 A micro:bit watch form-factor wearable for playing the rock/paper/scissors game (left); and a decorative necklace created using the CPX.

4.6 Applications of MakeCode & CODAL

MakeCode and CODAL now support over 40 physical computing devices and are used by millions of users every month [102]. This section shows how citizen developers are using the intuitive programming experience offered by MakeCode and CODAL to create diverse and advanced applications. We provide example applications for the BBC micro:bit the Circuit Playground Express (CPX), take note of the battery powered and embedded nature of each.

Wearables

Many physical computing devices are used to create interactive wearables. Figure 4.12 (left) shows a simple but highly popular micro:bit project: a micro:bit 'watch' that plays the rock/paper/scissors game by randomly displaying a rock (3x3 square), paper (5x5 square with center empty) or scissor icon on the 5x5 LED display when the device is shaken.

Figure 4.12 (right) shows the CPX used as the decorative component of a necklace. The CPX is powered by battery, connected to RGB LEDs (in addition to its on-board ones), and is programmed to change light patterns based upon the state of the on-board switch.

Digital crafting

Other popular applications augment physical computing devices with widely available crafts supplies. This allows citizens to quickly integrate devices into low cost, playful, and practical housings. Actuation adds a further dimension to applications. For example, Figure 4.13 (left) shows how cardboard, paper clips, and crocodile clips can be used to build an animated

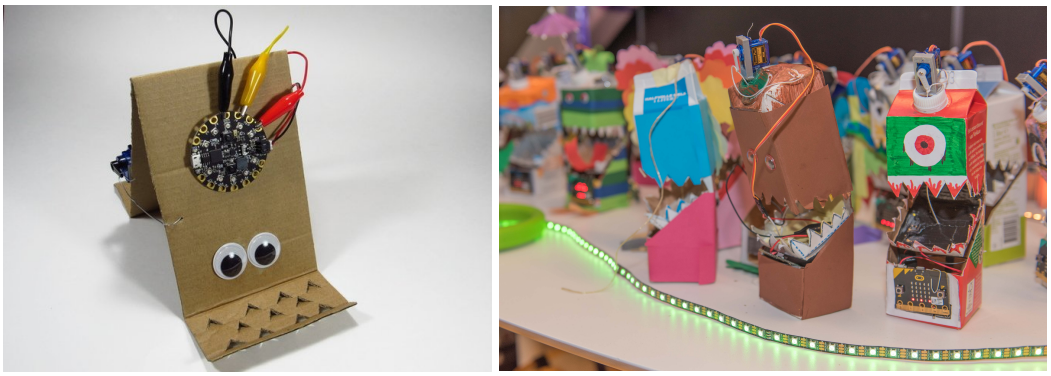

Fig. 4.13 Example projects: a cardboard inch worm (left); and light-reactive cardboard robots (right).

inch worm. The CPX is attached to folded cardboard and connected to a servo via crocodile clips from its GPIO. A paper clip is connected between the the front of the folded cardboard and the servo, and a program on the CPX actuates the servo causing the contraption to progressively 'inch' along surfaces.

The micro:bit can also control servos and motors via its edge connector. This has resulted in the creation of inexpensive, cardboard based creations that can react to their environment, such as those shown in Figure 4.13 (right). The simple robots pictured here open and close their mouths in response to light stimulus. Other similar projects for these devices include musical instruments such as a 'guitar' that changes pitch based on its physical orientation, and goal line technology for tabletop football games.

Science and measurement

MakeCode and CODAL are also being used to create applications for scientific exploration. The micro:bit and the CPX make ideal devices for this purpose due to their small and embeddable form factor. A great example of this is provided by the Bloodhound project (http://www.bloodhoundssc.com), a UK initiative to set a new world land speed record. As part of their remit to inspire students about STEM subjects, the 'Race to the Line' project was launched across the UK. In this project, students design, build and race model rocket cars in competition, learning about physics, aerodynamics, engineering and measurement. A micro:bit is integrated into the car's design, as shown in Figure 4.14, left. The micro:bit captures 3-axis accelerometer data of the rocket car during its race. After the race, students upload the data from the micro:bit and analyse the performance of their cars.

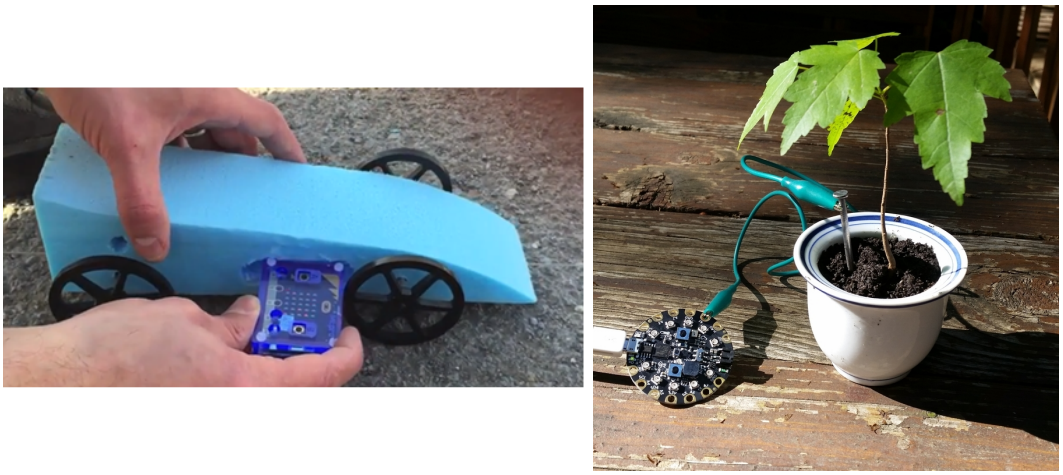

Fig. 4.14 Example projects: a Bloodhound model rocket car instrumented with a micro:bit (left); and measuring soil moisture using the GPIO of the CPX (right).

Similarly, Figure 4.14 (right) illustrates an environmental project that uses the CPX to measure soil moisture. The combination of water and nutrients in soil affect its conductivity—the more water, the greater the conductivity. This can be directly measured using metallic probes (note the use of inexpensive nails as probes in this example) and the CPX's integrated analogue voltage sensor. Then, the CPX is programmed to periodically take a moisture reading and record the results into the device's internal flash file system for later analysis.

Interconnected devices

Our final class of projects are those that make use of multiple, wirelessly networked devices. Whilst the micro:bit is capable as acting as a BLE device, we observed that the greatest level of innovation emerged from a simpler, custom built packet radio protocol. With the micro:bit radio API, micro:bits can form low level peer-to-peer multicast groups. Any data sent from one micro:bit is seen by all members of their group thus enabling a simple yet powerful basis for projects involving group collaboration in a way not feasible with BLE. Examples here include remote control vehicles, such as that illustrated in Figure 4.15 (left). This example uses two micro:bits sharing data over radio: one integrated into the vehicle to control steering and speed, and a second integrated into a handheld steering wheel that is used as a remote control. A second example is also illustrated in Figure 4.15 (right). This application makes use of the CPX's on-board infrared receiver and transmitter to create laser tag. Cardboard targets, each with a CPX, are placed on walls or integrated into object and players must run around and score as many points as possible.

Fig. 4.15 Example projects: a micro:bit-based vehicle controlled wirelessly by a second micro:bit (left); and laser tag using multiple CPXs (right).

4.7 Summary

This chapter has presented MakeCode and CODAL, a visual programming environment (P1) for microcontrollers. Applications can be created for microcontrollers using event-based APIs (P2) and without installation from any device with a web browser (P3). By mapping visual blocks onto C++ CODAL APIs, resulting applications near the efficiency of C++ (GP4).

Through our evaluation we show that our approach is up to 50 times more processor efficient than other solutions in the space. For physical computing devices, this translates to lower energy consumption and therefore a longer lifetime on battery (GP4). MakeCode and CODAL are also more RAM efficient than other solutions, and can even support extremely resource constrained devices *with just 2 kB of RAM* (GP4, GP2). We have applied MakeCode and CODAL to over 40 devices (GP2) demonstrating the extensibility of our approach (GP3).

Finally, we have illustrate how intuitive MakeCode and CODAL make microcontroller programming by reporting on how they are used to enable many advanced citizen projects (GP1). We also report on statistics which show that over one million citizen developers now use MakeCode and CODAL every month (GP1).

Chapter 5

JACDAC: intuitive hardware composition

Across Chapters 2 and 3 we identified properties of existing technologies that make hardware composition more intuitive:

HC1 *Dynamic connectivity*: Hardware composition is iterative and dynamic, and protocols that support dynamic device connection and removal (i.e. hot plugging) prove more intuitive to citizens.

HC2 *Dynamic device discovery*: Protocols that support dynamic connection also typically discover the capabilities and services offered by devices. Capabilities and services can then be automatically mapped to applications without intervention from citizens, making composition more intuitive.

HC3 *Hardware abstraction*: Standardised software abstractions and interfaces for common devices (i.e. mice and keyboards) allow citizens to compose hardware without having to perform any additional software installation. This makes hardware composition more intuitive to those with little technical expertise.

Another emergent trend across the literature was simplifying composition by reducing the number of wires (low infrastructure). However, existing wired protocols that support the properties above are more complex and require more memory and silicon to implement. This means that low cost microcontrollers typically do not support such protocols. We therefore concluded in Chapter 3, that there are no low-infrastructure wired protocols with dynamic connectivity, device discovery, and hardware abstraction, that are supported by low cost microcontrollers.

This chapter introduces Joint Asynchronous Communications; Device Agnostic Control (JACDAC), a single-wire protocol designed for the dynamic and iterative composition of microcontrollers and peripherals. It was designed to offer a high level of dynamism and abstraction, yet maintain the universal applicability enjoyed by existing low-level protocols through the reuse of peripherals common to nearly all microcontrollers. JACDAC abstracts peripherals and resources as a set of reusable high-level services that are shared among devices. The high-level nature of JACDAC services allows the run-time introspection of packets for an improved debugging experience. We also provide tooling to unify the world of the microcontroller and the Web, allowing users to develop new services, debug existing ones, and diagnose issues with a JACDAC network directly from a web browser.

We begin by providing an overview of the protocol in Section 5.1, which outlines the protocol stack discussed in Sections 5.2, 5.3, and 5.4. We then perform a comparative systems evaluation of JACDAC in Section 5.5 and show how JACDAC makes hardware composition more intuitive to citizen developers in Section 5.6. This final section describes the use of JACDAC by *fashion designers* as part of a fashion show in Brooklyn, New York.

5.1 Protocol overview

Microcontrollers that run the JACDAC protocol are known as JACDAC *devices* (Figure 5.1). They communicate JACDAC *packets* to each other across a shared bus. Each JACDAC device has a simple stack featuring: (1) a *physical layer* that handles hot plugging and the transmission and reception of JACDAC packets; (2) a *control layer* that performs device discovery, address allocation, and the routing of JACDAC packets; to (3) *services* which exchange JACDAC packets (Figure 5.2).

JACDAC devices optionally host *services* (Section 5.2) which abstract hardware peripherals and software resources. Services are akin to the REST-ful interfaces of the web and each service has an accompanying specification that standardises its software interface. At run time, application developers invoke APIs to send packets to services which follow the form defined in the service specification. As the format of messages is standardised, devices running compatible service implementations can replace one another without the modification of application code. Since the underlying communication medium is a broadcast bus, services can also operate in one of two communication paradigms: *host-client* (1:N), as in the world of web services, or *broadcast* (M:N), where packets are shared in a peer-to-peer topology.

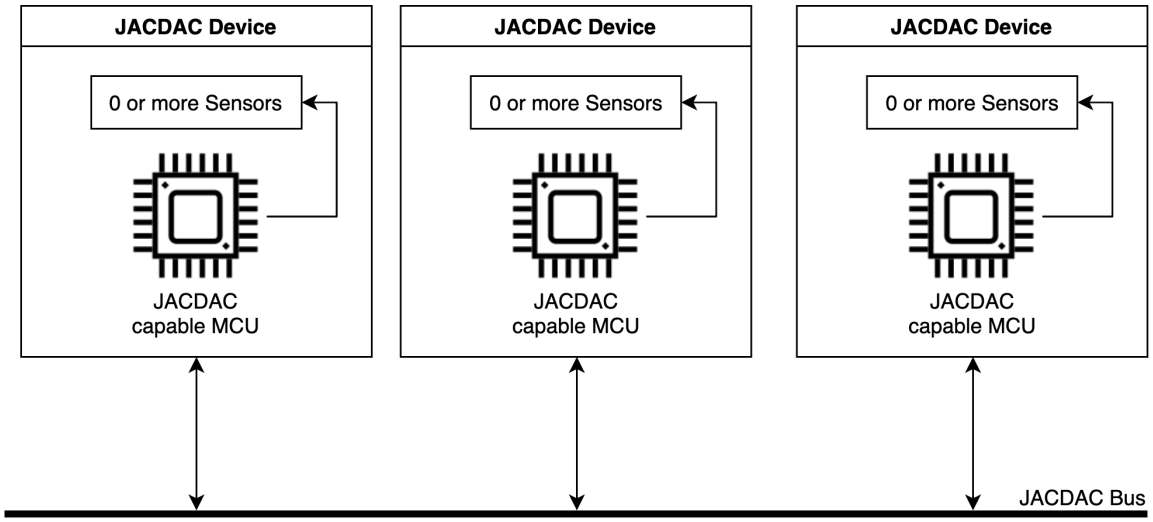

Fig. 5.1 The hardware organisation of a JACDAC device.

The *control layer* (Section 5.3) is responsible for device discovery, address allocation and the routing of packets to services. To enable dynamic device discovery, the control layer periodically transmits *control packets* that provide detail about a device and the services it makes available to the bus. Control packets can also contain optional advertisement data for each service. The presence or absence of control packets indicate the connection or removal of a device from the bus.

Also provided in control packets is a short 8-bit address that reduces the amount of address meta data required for routing standard JACDAC packets. To obtain a short address, a device must enumerate itself using control packets and propose an address to use on the bus. Devices need only enumerate themselves if operating a host or broadcast service; un-enumerated devices acting only as clients are free to use enumerated services without enumerating (transmitting control packets) themselves. A subset of the control layer can be implemented to support low capability microcontrollers.

The *physical layer* (Section 5.4) is designed to support hot plugging and communicates data using the UART module common to nearly all microcontrollers. Instead of separate wires for transmission and reception, JACDAC operates UART in half-duplex mode, using just one wire for both. This reduces the number of wires required to connect devices together.

Any JACDAC device can initiate a transmission on the bus, and devices must arbitrate for control of the bus before transmitting. Bus arbitration is performed using a short pulse whose duration dictates the upcoming baud rate of the transmission. The pulse can be one of four durations that map to corresponding baud rate: 1Mbaud, 500Kbaud, 250Kbaud,

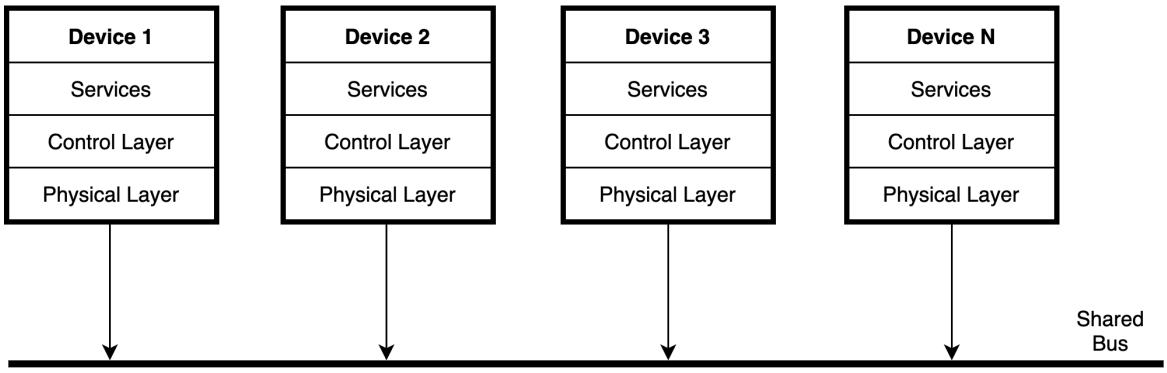

Fig. 5.2 The JACDAC stack required by each device

125Kbaud, allowing cheaper microcontrollers to be used. After the pulse, the physical layer communicates a JACDAC packet which have a four-byte header for addressing and error checking, and a maximum payload of 255 bytes.

To provide citizen developers with an easy and universal debugging and development experience, JACDAC leverages WebUSB [44] to bring the worlds of microcontrollers and the web closer together. JACDAC over WebUSB allows the web browser to act as *its own JACDAC device*, where JavaScript can be used to communicate and share services with devices on a physical JACDAC bus.

5.2 Services

REpresentational State Transfer (REST) defined a mode of interaction, that when applied to HTTP, changed the architecture of the Web [150]. Web-based REST services model resources and communicate using standardised messages written in an object interchange language such as JSON or XML. Standardised message formats allow the transfer of information between client and server. JACDAC services similarly specify messages, except they allow devices to share hardware resources in addition to software resources.

Message specifications for services are written in markdown [168] using a custom packet definition language. This language allows message formats to be translated into structures for many other programming languages, including typescript and C++.

Each specification is reviewed and approved by a consortium of JACDAC users and is subsequently assigned a 32-bit *service_class* to identify it on the bus at run time. This number guarantees a service implementation meeting the corresponding specification is present on the bus. By standardising packet formats in this way, hardware running compatible services

```
# Basic Gesture Service
A service that transmits
    gesture events produced by
    a hardware accelerometer.
```
```
metadata BasicGestureService {
    identifier: 0x00000008;
    mode: Host;
}
```
```

## Enumerations
```
```
enum Gesture: uint16 {
    TiltUp = 1, TiltDown = 2,
    TiltLeft = 3, TiltRight = 4,
    FaceUp = 5, FaceDown = 6,
    FreeFall = 7, ThreeG = 8,
    SixG = 9, EightG = 10,
    Shake = 11, TwoG = 12, Step
        = 13
}
```
```

```
Advertisement Data

```
```
control GestureControl {
    lastGesture: Gesture;
}
```
```

Packets
Event
```
```
report GestureEvent {
    event: Gesture;
}
```
```
Event configuration
```
```
command EventConfiguration {
    enabled: uint8
    eventType: Gesture
}
```
```
```

Fig. 5.3 An example service specification for a basic gesture service.

can act as drop-in replacements for one another, regardless of the underlying hardware. For example, two JACDAC devices may operate accelerometer services each using a different accelerometer peripheral, but clients of each service need not be aware of this detail.

While REST services adopt a single communication paradigm (client-server), JACDAC services allow the selection of two communication paradigms. In the first paradigm, *host-client*, *host services* provide access to a resource and allow other devices to augment their functionality by acting as *clients* to the service. The second communication paradigm, *broadcast*, allows devices to make host-to-host (peer-to-peer) communication. The selection of communication paradigm gives service developers the freedom to pick the correct paradigm for their service, rather than having the paradigm imposed on them by the protocol. Any future usages of *host* refers to services operating in either paradigm.

### 5.2.1 Specifying services

Rather than use difficult-to-parse interchange formats like XML and JSON, JACDAC services standardise messages as byte-level packet structures. Byte-level packet structures mean that even the lowest capability devices can parse JACDAC packets with ease.
```

A service creator specifies packet layouts for services using text and code snippets in markdown. We chose to use markdown due to its wide use, simple formatting, and easy dissemination through conversion to HTML. Contained in each code snippet is a custom packet definition language that is programming language agnostic. It can be used to automatically generate structures for languages like C/C++ and serialisation and deserialisation classes for higher-level languages like TypeScript or Python.

Figure 5.3 shows a markdown service specification for a gesture service that depends on an accelerometer. Specification files start with a short service description and metadata that defines any dependencies. In this case, the gesture service depends on a physical hardware accelerometer. The metadata field also allows for the inheritance of common JACDAC service interfaces like the sensor interface, which clients use to configure the streaming state of a sensor. An additional section, omitted here for conciseness, describes the full operation of a service in plain-text.

In the figure, code snippets in the "Enumerations" section are prefixed with the *enum* type. The enum type defines any constants that will be used in upcoming packet format specifications. In this case, the service defines all possible Gesture types.

The advertisement data section specifies any optional contextual information that the service may place in control packets sent by the control layer. Code snippets in this section are denoted by the special *control* type and in the figure, the gesture service advertises the last gesture that was detected.

The final section of the specification defines the packet formats expected to be sent to or from a host. The code snippets in this section are prefixed with either *report* or *command*: commands are sent to a host device and reports are sent by a host device.

Once a service is specified, its creator submits the specification via a pull request to the JACDAC Github repository (https://github.com/jacdac/jacdac) where it is reviewed by a consortium of JACDAC users. Once approved, service structures/classes are automatically generated and the specification is made available on the JACDAC website.

5.2.2 Developing services

To lower the barrier to entry and to aid citizen developers in iterative development, JACDAC is implemented in both C++ and TypeScript. Protocol layers from each stack can be combined and reused across different environments as shown in Figure 5.4. We discuss each of the environments in turn below.

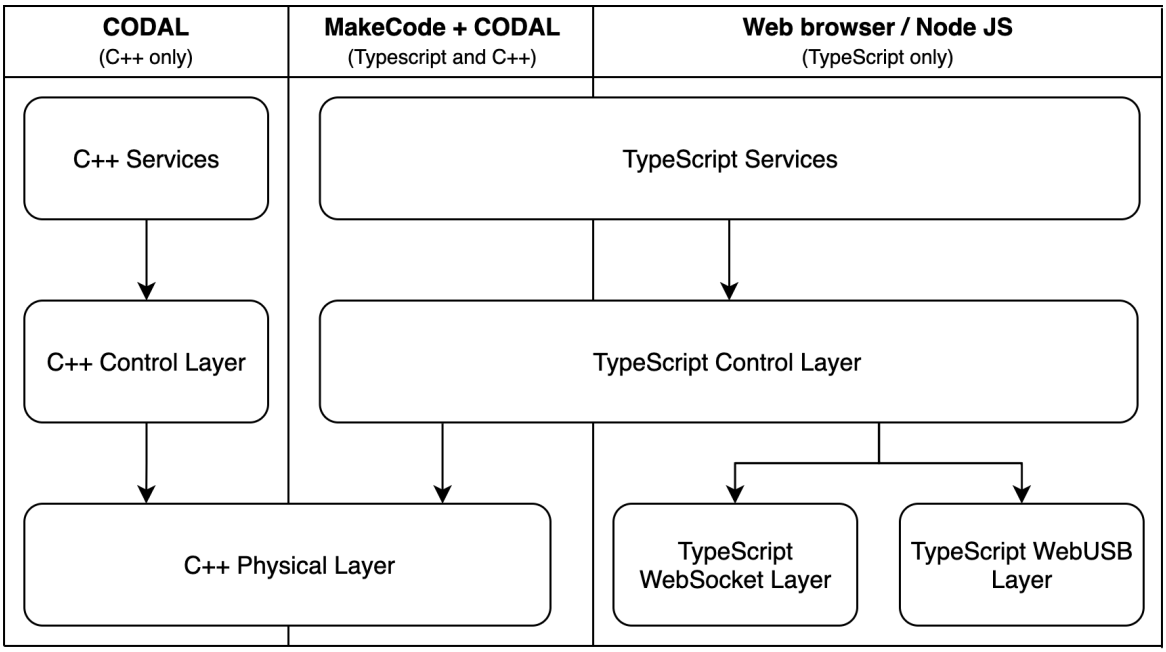

Fig. 5.4 Three alternate JACDAC implementations: in pure C++ via CODAL (left); using a mixture of TypeScript and C++ via MakeCode and CODAL (middle); and TypeScript only using the Web browser or NodeJS [273].

Web browser / Node JS

With powerful desktop JavaScript runtime environments and ubiquitous web browsers, the JavaScript ecosystem is one of the largest in existence. The JACDAC TypeScript stack (as shown in Figure 5.4; right) allows citizen developers to interact with JACDAC devices from any JavaScript environment, including NodeJS [273] and the web browser.

Emerging web technologies like WebUSB [44] even let web browsers to interact with USB devices. A JACDAC device that can also act as a USB device can therefore connect the web browser to the JACDAC bus. This allows for the universal development and debugging of JACDAC applications and services from any computer with a web browser and a USB port.

MakeCode

As MakeCode adheres to most of the TypeScript specification, MakeCode can compile the JACDAC TypeScript stack into binary instructions for physical computing devices. By extension, this means that any service written for the JACDAC TypeScript stack can be compiled and subsequently executed on any MakeCode device. Moreover, services written

in TypeScript can be easily presented in Blocks making programming networks of physical computing devices more intuitive This workflow is shown in Figure 5.5.

As shown in Figure 5.4 (middle), not all aspects of the MakeCode JACDAC stack are written in TypeScript. Since the code density of MakeCode compilation is less efficient than C++ compilation, a foreign function interface to the CODAL-based physical layer from TypeScript provides efficient real-time access to the hardware.

CODAL

Not all devices have the required processing power or memory resources to operate the JACDAC typescipt stack from MakeCode. For this class of device, CODAL provides an equivalent JACDAC stack written in C++ (Figure 5.4; left) which has a very close relationship with the TypeScript JACDAC stack. Of course, the use of C++ comes at the cost of complexity and the inability to exploit the wealth of TypeScript-based services already in existence. Regardless of the stack adopted by a developer, compatibility is guaranteed as long as service specifications are followed.

5.2.3 Using services

Figure 5.5 presents a basic gesture service implemented using the specification from Figure 5.3. The figure shows a host gesture service intended to run on a microcontroller that has a physical accelerometer, and a client that requires notification of gesture events. Both services inherit from a base class, *JDService*, which provides common interfaces and a base implementation for sending and receiving packets.

The host service interfaces with the accelerometer peripheral and performing gesture detection. When a gesture is detected, the host broadcasts the gesture on the JACDAC bus. The host service also implements *addAdvertisementData* to advertise the last detected gesture as per the specification. Upon reception of a gesture, clients use the *GestureEvent* class—automatically generated from the specification—to deserialize the byte buffer contained in a JDPacket, and subsequently raise an event for use by applications.

Bottom-left of Figure 5.5 is an example application written using the basic gesture service. When the application detects a Step gesture from the “LEFT_LEG” device, the lights change to red, and green when it detects a step from the “RIGHT_LEG” device. The ability to name devices allows citizen developers to refer to devices by application role rather than low-level address. Finally, service developers can add annotations to TypeScript classes so that services appear in the MakeCode editor as visual Blocks.

```
const GESTURE_SERVICE_ID = 0x00000008

class BasicGestureServiceHost extends JDService {
  accelerometer: Accelerometer
  lastGesture: Gesture

  constructor(accelerometer: Accelerometer) {
    super(GESTURE_SERVICE_ID, JDServiceMode.Host);
    this.accelerometer = accelerometer
    // gesture forwarding
    this.accelerometer.onGesture(Gesture.Any, function (type:
        Gesture) {
      let gest = new GestureEvent();
      gest.event = type;
      this.send(gest);
      this.lastGesture = type;
    })
  }
  addAdvertisementData(): Buffer {
    return new Buffer(this.lastGesture, 2);
  }
}
```

```
class BasicGestureClient extends JDService {
  constructor() {
    super(GESTURE_SERVICE_ID, JDServiceMode.Client);
  }
  handlePacket(pkt: JDPacket) {
    let rec = new GestureEvent(pkt.data);
    this.raiseHostEvent(rec.event)
  }
}
```

```
let leftLeg = new BasicGestureClient();
leftLeg.requiredDevice.name =
    "LEFT_LEG";
let rightLeg = new
    BasicGestureClient();
rightLeg.requiredDevice.name =
    "RIGHT_LEG";

leftLeg.onGesture(Gesture.Step, ()=>{
  // set lights to red
})
rightLeg.onGesture(Gesture.Step, ()=>{
  // set lights to green
})
```

Fig. 5.5 A JACDAC basic gesture service host (top), client (middle), and an example application (bottom-left) all written in TypeScript. Finally, the propagation of TypeScript to MakeCode Blocks is shown bottom-right of the figure.

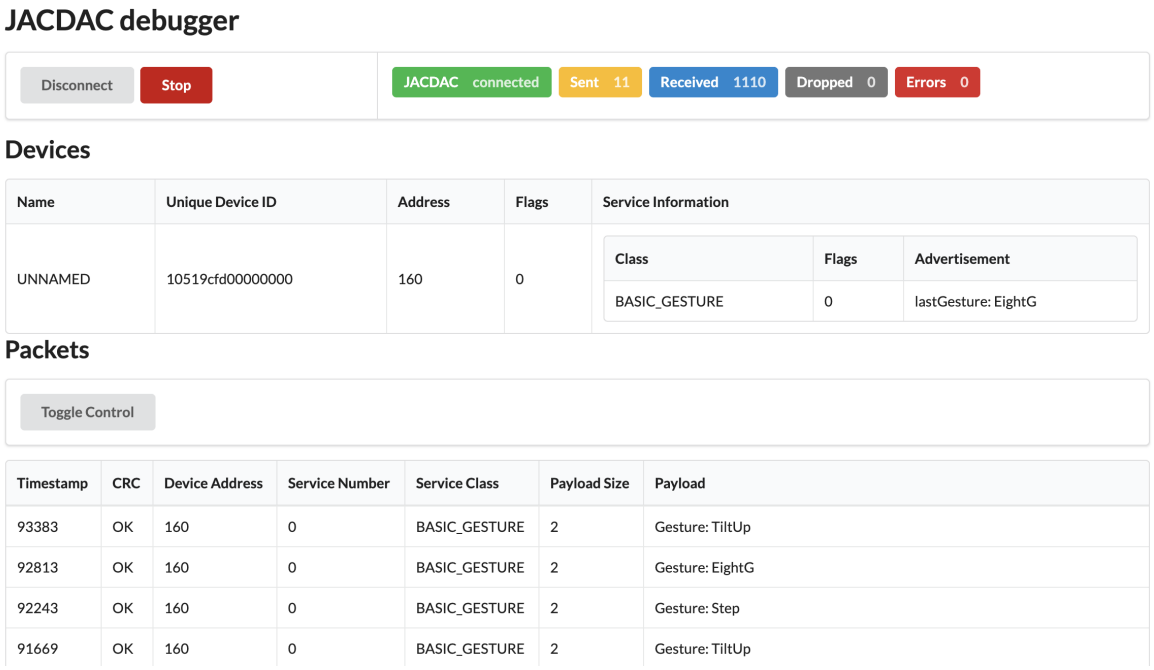

Fig. 5.6 The web-based JACDAC debugger, visualising packets received from a device running the basic gesture service.

5.2.4 Debugging services

As well as providing a familiar environment to develop JACDAC services, the Web browser also provides a universal, no-installation, debugging environment. Reusing WebUSB and the TypeScript JACDAC stack, the Web browser can be used to visualise packets sent on the bus. This allows citizen developers to more easily diagnose and debug error conditions. Furthermore, because services transmit standardised, well-defined packets, at run-time, packets can be decoded into a human comprehensible form. Figure 5.6 shows the debugger being used to decode the basic gesture service from Figure 5.5.

5.3 The control layer

This section details the control layer, which enables many of the dynamic aspects of JACDAC, including: dynamic device discovery, device address allocation, and routing JACDAC packets to services and clients.

5.3.1 Control packets

Control packets enable many of the dynamic features of the control layer and are simply JACDAC packets containing a standardised sequence of bytes, just like a normal JACDAC service. Control packets therefore contain the following: a 64-bit unique device identifier (UDID), an optional human-readable name, *device_flags* to indicate the status of a device, a list of hosted services, and an 8-bit condensed *device_address*.

The UDIDs contained within a control packet provide an absolute means to identify a device. They are used to detect and resolve address collisions, and also as a mechanism for devices to remember one another. JACDAC has two forms of UDID: (1) bus unique device identifiers, which may be generated based upon a serial number, a constant seeded random number generator, or stored into flash by the factory which produces the device; or (2) globally unique device identifiers which are assigned by an organisation. Devices use the seventh most significant bit of the UDID to indicate if their address is globally or bus assigned. This is similar to the IETF EUI-64 specification [173].

Each control packet also contains a list of hosted services and each service in this list is described by the following fields: a 32-bit service identifier, 8-bit service-specific flags, and optional advertisement data of up to 16 bytes. Clients to services are not enumerated on the bus, and are therefore no included in service lists. This allows the number of clients on the bus to scale infinitely.

The condensed 8-bit *device_address* inside a control packet can be thought of as a compressed form of the 64-bit UDID. These short addresses are dynamically allocated when a device is connected to the bus (described later). A 4-bit *service_number* is combined with a *device_address* to identify a host service running on a device. The *service_number* is computed from the order of services in the list of hosted services. By compressing addressing information, only 12-bits are required to route JACDAC packets between devices, ultimately increasing bus efficiency.

Control packets are typically sent by the control layer every 500 milliseconds and are ultimately used to detect the connection or removal of a device from the bus. Whilst a device is sending control packets it is considered connected, and when it stops, it is considered removed from the bus. The control layer provides connection or disconnection events to services with established links to devices. The use of regular control packets however introduces an overhead that becomes especially prevalent when many packets are being transmitted on the bus. The 500 millisecond timing is therefore flexible, and JACDAC devices can scale the number of control packets they send according to bus activity.

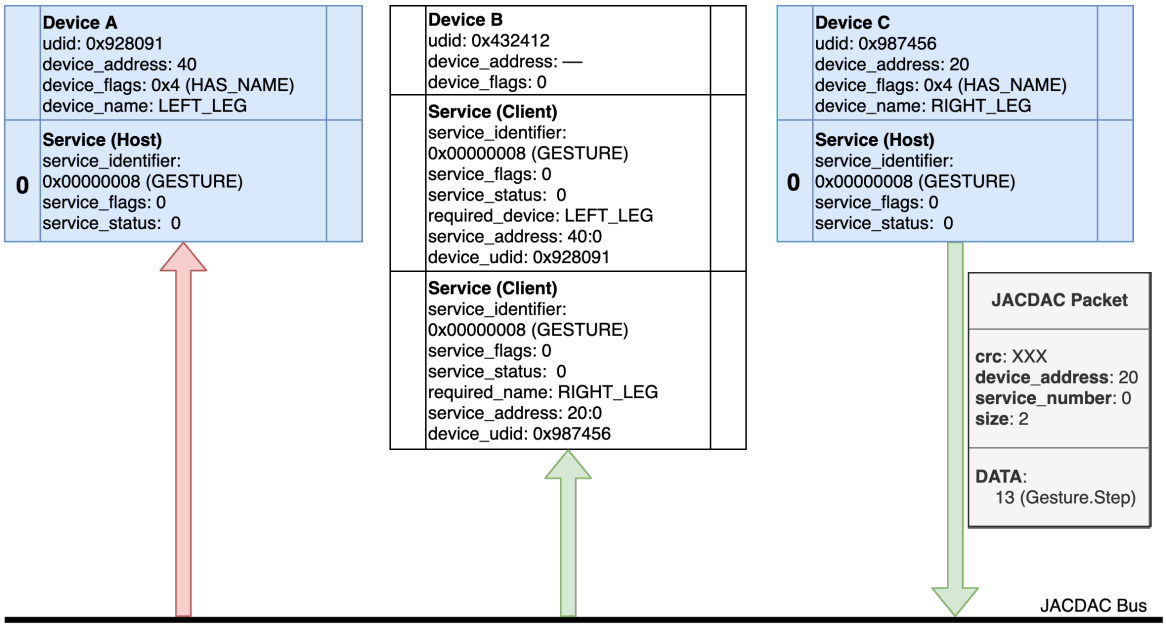

Fig. 5.7 A diagram of the devices from the example application. A blue box represents the control packet of an enumerated device, whereas a white box represents an unenumerated device. A green arrow indicates the device has received the packet. A red arrow indicates the device has ignored the packet.

5.3.2 Routing packets to services

The *device_address* and *service_number* fields in a packet identify a service operating on a device. Host services emit packets using their *device_address* and *service_number*, and clients address hosts by sending packets using the *device_address* and *service_number* of the host. No direction information (i.e. from client to host and vice versa) is provided in packets.

Figure 5.7 is a diagram of the underlying communications supporting the application from Section 5.2.3. The application device (Device B, middle) listens for gestures from a device positioned on the left leg (Device A, left) and another positioned on the right (Device C, right). Each device is assigned a name according to its application purpose. Device A has the address 40, Device C has the address 20, and both gesture services on each device have the *service_number* 0. Device C is emitting a packet from the gesture service containing the step gesture.

The client services on Device B are already connected to each of the host gesture services. To reach this stage, the control layer on Device B matched local connection information specified by the application. This information can be as broad as a service class (‘any gesture service’), generic as device name (‘any gesture service with this name’), or as specific

as unique device identifier ('this device'). The example application running on Device B specified devices by name: *LEFT_LEG* and *RIGHT_LEG*.

Once a client is connected to a relevant host service, information about host device, including connection state, is maintained by the control layer. Since device information is only stored when clients are connected to a host, maintenance overhead scales with the complexity of the application, not the network. Upon reception of packets from the bus, it is then the case of a simple lookup, matching the *device_address* and *service_number* from the packet to the locally running client service. If the host is removed from the bus then the control layer begins to look for a suitable replacement. As devices are specified by name in the example application, no replacement would be found.

5.3.3 Dynamic address allocation

Before a device can send a packet, the control layer must first obtain an address to use. The process to obtain an address is known as *enumeration* and a device is said to be *enumerated* when it has been assigned an address.

The algorithm to obtain an address is designed for simplicity and engineered for a broadcast environment. The algorithm itself is similar to the IPv4 address allocation process [173] but is augmented by exploiting the broadcast nature of the bus topology.

Before proposing an address to use, a device first listens to control packets emitted from already enumerated devices to determine a free address. A device then lays claim to an address by transmitting a control packet with the *PROPOSAL* flag bit set in the *device_flags* field. If an enumerated device on the bus already is using the proposed address, the existing device returns the same control packet with the *REJECT* flag set. The rejecting device optionally can recommend a free address in the rejection control packet. After receiving a rejection, the proposing device must pick a new address and begin the proposal phase again.

If at any point a proposing device receives a control packet containing its proposed address, it also must pick a new address and begin the proposal phase again. Two devices that propose the same address must observe the rules for address collision resolution (below). After two control packets without rejection, a proposing device removes the *PROPOSAL* flag from their control packets to indicate ownership of an address.

Address allocators

JACDAC devices can allocate address on the bus in many different ways. The C++ and TypeScript currently define three allocators, which can be used depending on device capability: *random*, *linked*, and *stateful*. Each allocator maintains an increasing amount of state.

The *random* allocator is the simplest. It maintains no state and upon each allocation it simply generates a random number between 1 and 254. Due to its lack of state, this allocator does not recommend addresses to other devices during an address collision.

The *linked* allocator maintains a small amount of bus address state. Initial addresses are allocated just like the random allocator (i.e. randomly), but once an address has been found, this allocator acts as a linked list, maintaining the next logical device address on from its own. When an address collision occurs through allocation or otherwise, the linked allocator generates a new address between its own address and the next. This helps to resolve allocations more quickly than purely random allocation.

With large networks, the prior two allocators will take longer to allocate addresses. There will be less free addresses and many unnecessary proposals will take place. The *stateful* allocator therefore maintains a bitmap of the entire address space, marking addresses as used or unused. At the expense of 32-bytes of memory (253 bits for all allocatable JACDAC address), this allocator is able to analyse free addresses to minimise address collisions. Only one device needs to operate a stateful allocator to reduce address allocation time. This single allocator can step in during any allocation and accurately recommend an unused address. This collaboration allows less capable microcontrollers (using stateless allocators) to offload computational complexity to more capable microcontrollers (using stateful allocators).

5.3.4 Resolving address collisions

It is possible for two different JACDAC devices to share the same device address, for example, if two separate JACDAC networks are joined together. Address collision detection is performed whenever a control packet is received and is a simple case of comparing the *device_address* and UDID in the packet to its own. If a control packet has the same *device_address* but a UDID, an address collision has occurred. To resolve this collision, the control layer observes a simple rule: the device that detected the colliding control packet must begin the enumeration process again to establish a new address. In other words, the first device to communicate a control packet retains ownership of that address.

In between control packets though services are communicating with one another. It is therefore possible for an address collision to occur for sometime without detection. During

Field Size (bits)	Field Name
12	CRC
4	service_number
8	device_address
8	size
8 * size	data

Table 5.1 The JACDAC packet format

this window, the control layer may route incompatible packets to services. Spurious behaviour would likely result if services were to attempt to decode these packets and so JACDAC guards against this eventuality by including the UDID in every CRC calculation.

5.3.5 Supporting low-cost microcontrollers

The address compression, allocation, and collision detection techniques used by the control layer add complexity. Devices must maintain state to route packets to services and must have the capability to receive all control packets which is problematic for resource constrained devices.

JACDAC enables resource constrained devices to operate in a *transmit-only capacity*, adhering to a reduced subset of control layer requirements. Transmit-only devices must send control packets regularly with all the same information as a full JACDAC device. However, transmit only devices do not observe short addressing conventions.

All transmit-only devices share a short *device_address* of 255. This signifies to the control layer that the device is using UDIDs as a means of addressing. In every standard packet emitted from a device with address 255, the UDID consumes the first 8 bytes and the control layer then routes packets to services based on UDID and *service_number*, rather than *device_address* and *service_number*. Control packets for transmit only devices remain exactly the same, except that the short address inside is fixed to 255.

5.4 The physical layer

I2C and SPI are pervasive protocols in the world of microcontrollers and most microcontrollers have dedicated peripherals to support their operation. RS232 [86], otherwise known as Universal Asynchronous Receiver Transmitter (UART), is another similarly pervasive protocol common to most microcontrollers. Unlike I2C and SPI, UART transmissions are streams of free-form bytes, with structure and length defined by application developers.

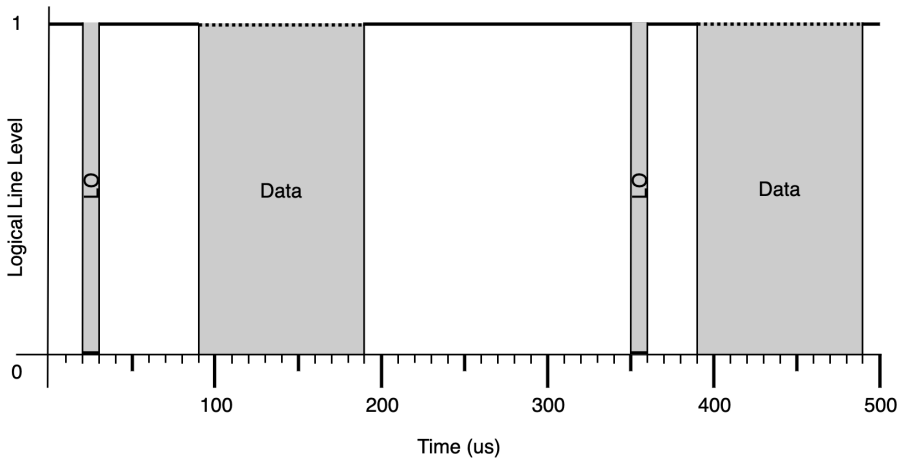

Fig. 5.8 A JACDAC frame

UART also usually operates in full duplex mode for point-to-point communications. This requires separate wires for reception and transmission.

The JACDAC physical layer builds on the foundations of UART. Instead of full-duplex, it operates UART in half-duplex mode to create a shared bus. JACDAC defines a fixed packet structure and adds bus arbitration for multi-central operation. A JACDAC packet (Table 5.1) is formed of a 4-byte header and a maximum 255-byte data payload. The packet header consists of: a 12-bit CRC to ensure packet integrity, a 4-bit *service_number* and an 8-bit *device_address* for addressing services, and an 8-bit size field that determines the length of the data payload.

The process of bus arbitration is simple to implement and is achieved by a device bringing the bus from the default logical one (3.3 volts) to a logical zero (0 volts) for a short period. Known as the low pulse, the duration of this pulse dictates the baud rate of the UART transmission. A low pulse always comes before a JACDAC packet and the combination of a JACDAC packet and the low pulse forms a JACDAC frame, illustrated in Figure 5.8. JACDAC supports four baud rates so that even the lowest capability microcontrollers can implement it.

5.4.1 Hardware requirements

To allow as many microcontrollers as possible to adopt the JACDAC protocol, the hardware requirements of the JACDAC physical layer are minimal, requiring: (1) the ability to receive / transmit UART-style bytes using a single wire (10 bits: 1 byte, 1 stop bit, 1 start bit); (2) the transmission of bytes at one of four baud rates for transmit only devices:

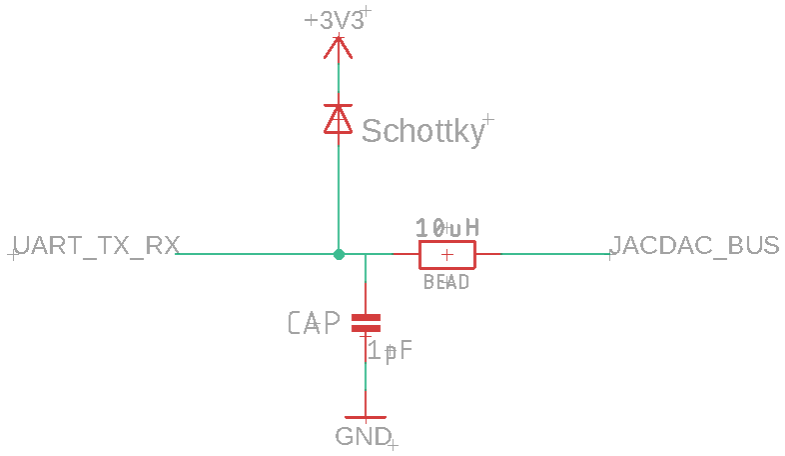

Fig. 5.9 Hardware schematic for an electromagnetically compatible JACDAC data line

1Mbaud, 500Kbaud, 250Kbaud, 125Kbaud; (3) a general purpose input/output, optionally with interrupt capabilities; (4) the ability to approximate time, whether through instruction counting or a hardware timer; and (5) the ability to generate random numbers (or at least seed a software random number generator). In Figure 5.9, we provide a schematic of the minimal electronic components required to ensure Electromagnetic Compatibility (EMC). These are components optional and are only required if JACDAC is going to be used in a consumer product.

5.4.2 Transmitting a packet

As mentioned previously, devices must arbitrate for possession of the bus before transmission. This arbitration phase is known as the *low pulse* and the duration of the low pulse also dictates the baud rate of the upcoming UART transmission.

To begin the low pulse, a transmitter must drive the bus low (0 volts; logical zero) for 10 bits (1 byte) at the desired baud rate. This can be implemented using a single GPIO write operation. At 1Mbaud the low pulse duration would be 10 microseconds, and for 125Kbaud 80 microseconds. To prevent simultaneous transmission, devices must check the bus state before commencing the low pulse. This can be implemented using a simple GPIO read operation. Combining bus arbitration and communication of the upcoming baud rate into a single operation simplifies software implementation, reduces hardware complexity, and allows microcontrollers to automatically filter packets they are not capable of receiving.

After the low pulse, the initiator configures UART hardware for transmission. Due to the lack of direct hardware support, the transmitter must wait 40 microseconds (4 bytes

at 1Mbaud) to allow receivers to complete low pulse detection and configure registers for reception. After transmission is complete, all devices connected to the bus resume listening for a low pulse.

5.4.3 Receiving a packet

The process of receiving a JACDAC packet is similarly simple. Devices use a single GPIO to listen for a low pulse. This can be either implemented by way of a GPIO interrupt, or synchronously waiting for the logical line level to change. The former however allows processors to enter a power efficient sleep in between bus activity.Once a low pulse has been detected, receivers have 40 microseconds (4 bytes at 1Mbaud) to configure UART hardware for reception.

UART reception can be implemented (and usually is) in two stages. The first stage receives only a JACDAC packet header (4 bytes) which contains the size of the variable length payload. Appropriately sized software buffers are then allocated based upon the size contained in the header and used to receive the remainder of the packet (stage two). Split reception therefore reduces the static memory overhead of JACDAC, allowing microcontrollers to dynamically allocate and size buffers to packets. Resource constrained devices can also implement packet filtering based upon size, or other metadata, like address. After completing packet reception, devices resume listening for packets.

5.4.4 Supporting hot plugging

To support the connection and removal devices at any point, even during transmission, the physical layer defines the following protocol timings:

Bus Idle Period is the minimum time before the bus is considered idle. An idle bus is defined as no activity (logical one) for two bytes at 125 Kbaud.

InterLoData Spacing is the minimum time before data can be sent after a low pulse. This spacing is a minimum of 6 bytes at 1Mbaud, and the maximum gap before data begins is two bytes at 125 Kbaud. Times are relative from the end of the low pulse.

Interbyte Spacing is the maximum permitted time between bytes and is defined as two bytes at 125 Kbaud.

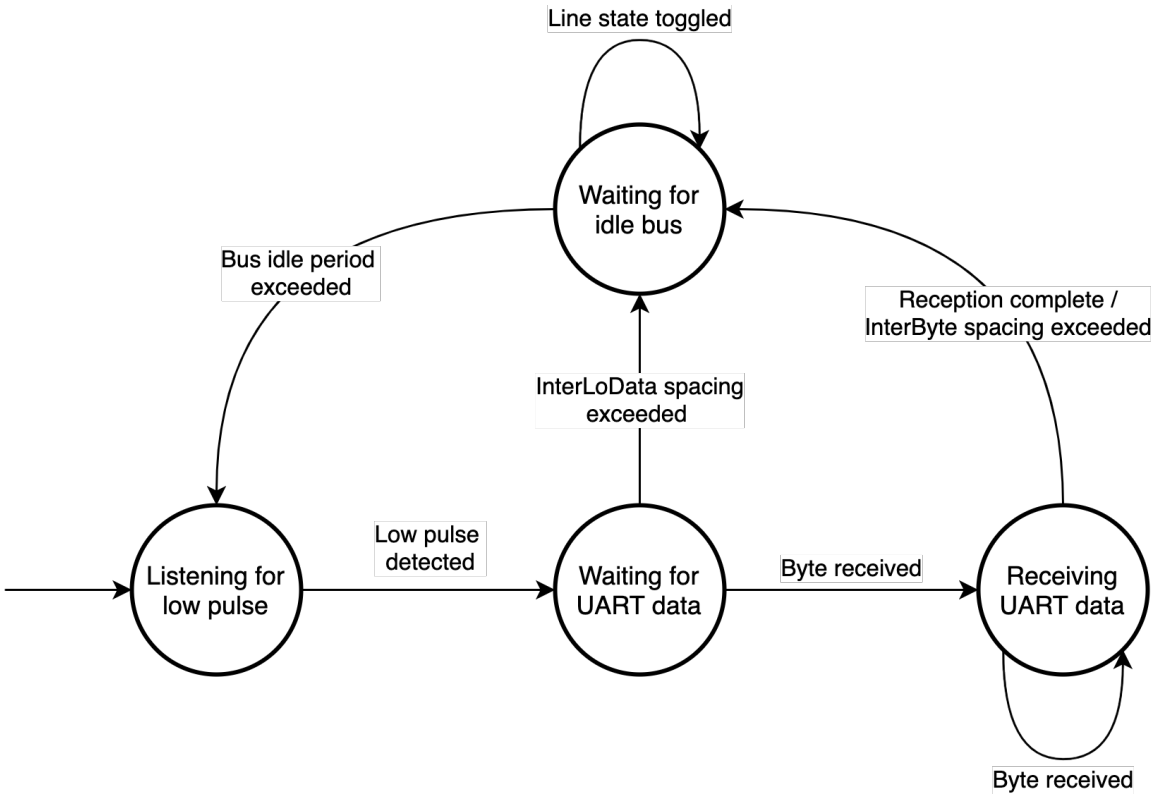

Fig. 5.10 State diagram for bus error detection during reception.

Interframe Spacing is the minimum space between frames and is defined as two bytes at 125 Kbaud.

During operation, JACDAC devices must check for any violation of the protocol timings above. If a violation is detected, an error condition is generated. Error conditions can only really occur during reception and when they do occur, receivers must enter an error state and resume listening for frames once the idle period has been detected. An idle bus is detected by maintaining the time from when the bus last transitioned from low to high, resetting this time if the bus transitions again. Figure 5.10 shows the state diagram for error detection during reception.

5.4.5 Preventing simultaneous transmission

It is possible for two JACDAC devices to arbitrate for the bus simultaneously—even with a low-latency physical medium. To guard against simultaneous transmission, devices check the bus state before initiating a transmission. If the bus state is low when performing this

check, devices enter the error state (as in Figure 5.10) and wait for the bus to return to idle. In some cases, on-board UART hardware is capable of detecting two devices driving the transmission line and this can be used instead. Regardless of hardware capability however the Cyclic Redundancy Check (CRC) contained within a packet will always guarantee packet correctness at the software layer.

5.4.6 Supporting low-cost microcontrollers

The JACDAC physical layer is designed to be implementable on super low-cost microcontrollers. At the physical layer, it does this by supporting multiple baud rates, allowing less capable microcontrollers to participate in communications.

Devices that are only capable of receiving at lower baud rates cannot receive packets from higher baud rate devices. This is problematic especially when address collisions need to be resolved. The JACDAC physical layer therefore specifies that microcontrollers that implement baud rates faster than 125Kbaud must also implement all slower baud rates to ensure that address collisions can be resolved correctly. Beneficially, this means that as less capable microcontrollers cannot receive packets from more capable microcontrollers, less capable devices will always win address contention.

5.5 Evaluation

This section evaluates many of the design decisions taken throughout the creation of JACDAC. We profile the performance of JACDAC, evaluate the efficiency of JACDAC compared to I2C, and analyse the value and efficiency of address compression and allocation.

5.5.1 Performance

This section focuses on the base performance metrics of the protocol, including: latency, packet loss, and memory consumption. Figures from the upcoming sections were obtained from binaries compiled using C++ JACDAC implementation from CODAL. We used command line utilities to inspect resulting binaries for the size of individual JACDAC layers.

To profile timing and latency we used two different devices that support JACDAC: the Adafruit Circuit Playground Express (CPX) and the Brainpad Arcade [10]. The CPX has been discussed extensively in this thesis, but as a reminder the device has a SAMD21 microcontroller with 32 kB of ram and 256 kB of flash. The Brainpad Arcade is a hand-held

reprogrammable gaming device with an STM32F4 microcontroller. The microcontroller has 256 kB flash and 96 kB of RAM, and runs at 84 MHz.

Protocol timings were obtained from devices using a specially modified JACDAC stack. The stack toggled various GPIO at different points of the reception and transmission process. Device GPIO were attached to a logic analyser for measurement.

Latency

To determine the latency of packet reception and transmission time, we used two CPXs. One device was designated as the transmitter and the other as the receiver. The transmitter was configured to send a JACDAC packet with a payload of 10 bytes and both devices were connected to each other using crocodile clips. Each device was programmed to set a GPIO high for the duration of receiving or transmitting a packet to obtain a visible signal for timing.

The total time taken to transmit a frame (as seen by the transmitter) was 215 microseconds. The total time taken to receive a frame (as seen by the receiver) was 222.7 microseconds. And the total line time for frame transmission was 214 microseconds which means that packet latency is 8 microseconds for the SAMD21. We expect this figure will vary between processors.

Packet loss

To calculate packet loss, we connected two CPXs and a Brainpad Arcade to form a single JACDAC bus. The two CPXs were programmed to each transmit 5,000 packets as fast as possible upon receiving a digital signal over a GPIO routed from the Brainpad Arcade. The Arcade was programmed to act as a bystander, only receiving transmissions and displaying diagnostic information from the physical layer on its screen. After repeating this test three times, we calculated the average packet loss to be 1.7%. Packet loss could certainly be reduced with direct hardware support.

Memory consumption

To give an indication of the total code size for a complete JACDAC stack, we compiled the JACDAC C++ layer for the SAMD21 and read the size of the each layer in bytes from compiled object files.

In total, including optional services, a full JACDAC host implementation consumes 12.1kB of flash and requires 496 bytes of RAM for driver operation. With optional services omitted, flash consumption decreases to 11.2kB. The predominant factors in the total flash

size of the JACDAC C++ layer are the control (3.1kB) and physical (4.7kB) layers. We expect that for a client-only implementation this figure would be significantly reduced. A transmit only implementation of JACDAC for the PIC12F consumes just 185 bytes of flash and 32 bytes of RAM.

5.5.2 Implementation complexity

We have written JACDAC implementations for microcontrollers with varying capability. At the highly capable end of the microcontroller spectrum, we found it easy to implement JACDAC on the following microcontrollers: STM32F4, STM32F1, SAMD51, SAMD21, NRF52. These microcontrollers typically have dedicated UART peripherals with Direct Memory Access (DMA) and an abundance of flash (256 kB) and RAM (from 32 kB to 96 kB). As a result, they cost the most (~$0.99+).

We have also written JACDAC implementations for medium capability microcontrollers like the STM32F0, STM32G0, and NRF51. These microcontrollers have dedicated UART peripherals but typically DMA is not supported. RAM offerings range between 4 and 16 kilobytes, and as a result, these microcontrollers cost less the highly capable microcontrollers (~$0.40+).

We have also implemented JACDAC on the PIC12F, a microcontroller that is representative of the lowest capability microcontrollers. Unlike prior processors, the PIC12F is 8-bit, has 128 bytes of RAM and just 1 kB of flash. As mentioned previously, a JACDAC transmit only implementation for this processor consumes just 185 bytes of flash and 32 bytes of RAM. Whilst slightly more challenging than other processors, implementing the protocol was not challenging and we plan to create an implementation for the 4-bit PADAUK PMS150C (~$0.03).

5.5.3 Electromagnetic Compatibility (EMC)

JACDAC is currently used to compose consumer devices together, and as such it is subject to European Electromagnetic Compatibility (EMC) regulations. We tested the circuit from Figure 5.9 for EMC compatibility in a electromagnetically shielded chamber. Figure 5.11 shows our findings.

The figure shows that JACDAC is well within the required tolerances. The top-most black line indicates the European threshold for industrial applications, and the red–line is the threshold for consumer applications. To be EMC compliant, electromagnetic emissions must be below these thresholds so to not interfere with other appliances and devices. As can be

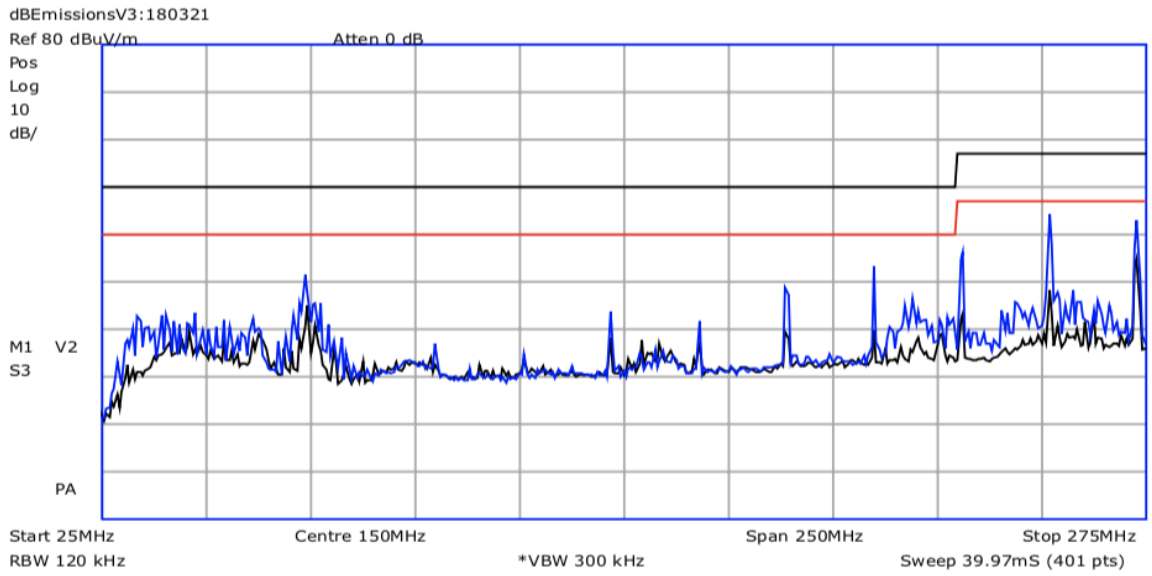

Fig. 5.11 Electromagnetic Compatibility (EMC) test results. The top-most black line indicates the European threshold for industrial applications, and the red–line for consumer applications.

seen from the figure, JACDAC, the blue and bottom-most black lines, are well below these thresholds.

5.5.4 Comparison with I2C

As part of our evaluation, we compare JACDAC to I2C. Table 5.2 compares packets transmitted at 250 Kbaud in both I2C and JACDAC, showing the time taken taken to transmit a packet of incremental payload sizes. An additional column provides the percentage difference in timings between the two protocols.

From the table, it is clear to see that JACDAC incurs a slight performance penalty. This can be mainly attributed to the bus arbitration technique adopted by JACDAC which takes a total of 80 microseconds at 250KBaud. The remaining performance degradation can be attributed to the 4 byte header, compared with I2C's three transaction overhead, and 10-bit byte format, versus I2C's 9-bit byte format.

One key difference between JACDAC and I2C is that, unlike with I2C, JACDAC devices do not have specialised hardware to perform bus arbitration. Instead, JACDAC devices require application intervention to detect the start and end of the low pulse and configure buffers for reception. As a result, the time between the low pulse and the transmission of

Table 5.2 Time taken to transmit a packet at 250KBaud at various payload sizes in both I2C and JACDAC.

Packet size	I2C transmission time (us)	JACDAC transmission time (us)	Difference (%)
10	468	640	37%
25	1008	1240	23%
50	1908	2240	17%
100	3708	4240	14%
150	5508	6240	13%
200	7308	8240	13%
250	9108	10240	12%

data is intentionally large to account for different processor speeds. This could be reduced with direct hardware support.

Another key difference is that JACDAC is designed for dynamic hardware composition where *transmission errors are to be expected*. As a result, every JACDAC packet includes a Cyclic Redundancy Check (CRC) which increases the total packet size. I2C, which was not designed for dynamic hardware composition, does not incorporate such redundancy checks which is potentially detrimental to applications.

JACDAC does however make efficiency gains elsewhere. For instance, whereas I2C only allows point-to-point communications, JACDAC allows many devices to be addressed with a single packet. For many applications therefore, the use of JACDAC results in an efficiency gain as only one packet is required to address many peripherals.

Another efficiency gain is made with JACDAC's support for multi-baud operation. I2C can only communicate at the fastest rate supported by all devices on the bus. This therefore means that if an especially slow peripheral is connected to the bus, *communication between all peripherals takes a performance penalty*. This is in contrast to JACDAC where multi-baud operation is actively supported, allowing devices to pick the communication rate appropriate to their hardware without impacting the bus.

5.5.5 Address compression

To find out how beneficial address compression is to the throughput of JACDAC, we calculated packet overhead with and without address compression at various packet sizes. Figure 5.12 shows the results.

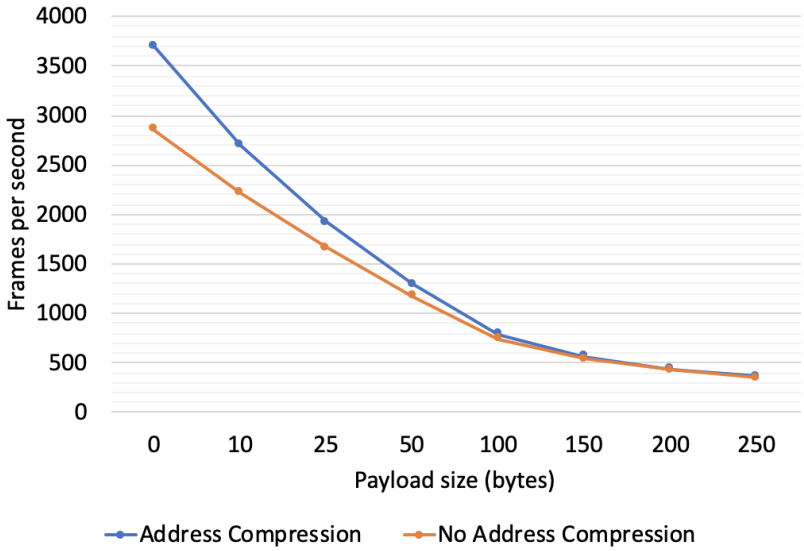

Fig. 5.12 Frames per second, with and without address compression.

The figure shows that for small packet sizes, address compression has a significant benefit, allowing 500 more frames per second for packets with 10 byte payloads. As the payload size increases however, the benefits of address compression decrease. As we expect applications to mostly send small packets, we conclude that overall address compression is beneficial.

5.5.6 Address allocation

To evaluate the performance of our distributed address allocation approach, we measure the allocation time of the three different address allocators discussed in Section 5.3: *random*, *linked*, and *stateful*. Due to the scale of these tests we used the JACDAC TypeScript layer and NodeJS [273] to simulate a JACDAC network. Each virtual device ran a full JACDAC stack and used a physical layer with the same properties as a real physical layer: packets sent by one device are received by all nodes with little latency. Because of the computational complexity of simulation, address allocation times scaled proportionally with the number of devices present on the bus. We therefore instead use the total number of packets sent by each allocator as a metric of success.

We evaluate each allocator in networks of varying size, and under three different test scenarios: (1) powering on all devices at once; (2) adding a device to an already established network; and (3) joining two established networks. Table 5.3 shows the results of our tests. We provide the most optimum number of packets ('Best (packets)') and the percentage

Table 5.3 The percentage of packets produced over the optimum, for the three different address allocations, with networks of incremental sizes under three different scenarios. The number of devices for the scenario where two established networks are joined should be doubled.

			Network size						
Scenario		Allocation Algorithm	10	25	50	100	150	200	250
1	Power on all networked devices at once	Random	=	=	=	1%	5%	17%	54%
		Linked	=	=	=	=	2%	6%	39%
		Stateful	=	=	=	=	=	=	=
		Best (packets)	30	75	150	300	450	600	750
2	Add a single device to an established network	Random	=	=	=	=	22%	44%	722%
		Linked	=	=	=	=	=	=	622%
		Stateful	=	=	=	=	=	=	=
		Best (packets)	3	3	3	3	3	3	3
3	Join two established networks	Random	=	19%	10%	34%			
		Linked	=	=	3%	25%			
		Stateful	=	1%	8%	19%			
		Best (packets)	3	6	33	120			

difference between the optimum and the actual. For the third scenario, where two established networks are joined, network size represents the number of devices in each network. The optimal number of packets for this scenario is based on the average number of address collisions we detected in our experiment. For two networks of size 10, we observed an average of 1 address collision, requiring 3 control packets to allocate a new address.

From the table, we can conclude that for the first two scenarios the random and linked allocators perform well up to network sizes of 150 devices but suffer significantly in congested networks. The stateful allocator performs optimally as expected for both scenarios 1 and 2. The difference between allocators is not so stark for the third scenario however. Here, the random and linked allocators perform almost as well as the stateful allocator. This can be attributed to the stateful allocator having out-of-date state due to the rapid addition of new devices.

Even though performance of the simpler allocators suffers with networks of larger sizes, the random allocator is the default allocator used by most JACDAC implementations. We consider larger networks to be a rare scenario, and expect that device networks rarely exceed 50 devices. We expect that in the wild, JACDAC networks will contain devices with many

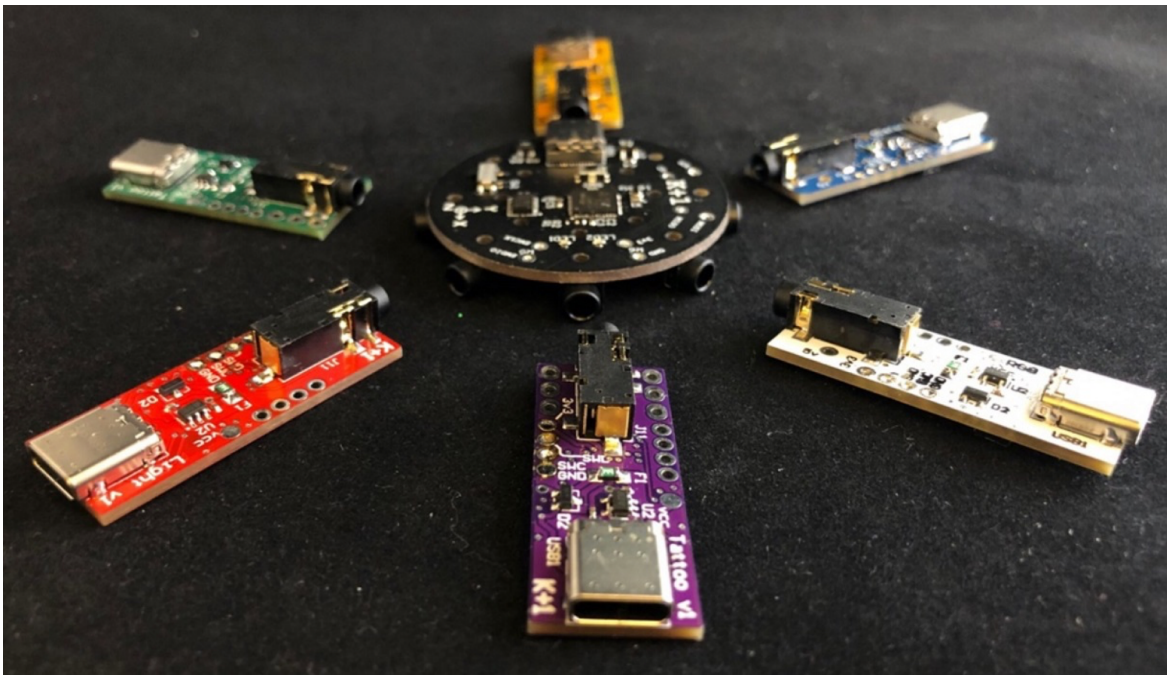

Fig. 5.13 Interactive devices used by fashion designers in Project Brookdale. Beads (outer) and Brain (inner).

different types of allocator and these different allocator combinations will reduce the total number of packets required for allocation.

5.6 Applications of JACDAC

In this section we highlight current applications of JACDAC to show its real-world applicability. The dynamic properties of JACDAC has lead to its adoption for *wearable technology* and *gaming*.

5.6.1 Wearable technology

There is a huge demand for wearable technology—Apple recently surpassed 10 billion dollars in revenue for for their wearable offerings alone (i.e. Apple Watch and Earpods) [95]. Evidence suggests that this trend of more integrated wearable technology is only going to continue and the next step is potentially for clothing manufacturers to embed electronics into garments. This is already happening in the research community via projects like Jacquard [240].

Fig. 5.14 A fashion designer prototyping and embedding JACDAC devices into a garments (left) and the final garment (right).

As shown in Chapter 2, members of the avant-garde fashion community are already integrating sensors, actuators, and lighting into garments [9]. This process is riddled with complexity however, and often dedicated technical teams are required to program microcontrollers and compose hardware for garments

In collaboration with Microsoft Research and the Brooklyn Fashion Academy, we created Project Brookdale [243]. Project Brookdale set out to simplify the process of building an interactive garment with the ultimate aim of empowering fashion designers to build interactive garments without the intervention of technologists. The project culminated in a high profile fashion show in Brooklyn, New York where fashion designers showed off their interactive garments.

Project Brookdale gave fashion designers a toolkit of easy-to-connect, re-programmable physical computing devices to create interactive garments (Figure 5.13). There were two types of device: user programmable devices called *Brains*, and input and output devices called *Beads*. Input Beads could sense changes in light, colour, motion, and environment, and output beads could control motors, servos, and RGB LEDs. Beads were colour coded depending on functionality.

Beads and Brains were connected to one another using 3.5 mm stereo audio jack cables. Jack cables were chosen because of their rugged, yet easy, connectivity and the already existing ecosystem of cables and splitters. Wired connectivity also allowed for easier power distribution between devices and the three rings of the jack connector provided power (tip), data (middle), and ground (base).

JACDAC was used to enable intuitive hardware composition by fashion designers (Figure 5.14) and was the means of communication between Brain and Beads. JACDAC's *single wire, low infrastructure* operation permitted it to be used as the communication mechanism

between devices over the middle ring of the jack connector. Its support for *dynamic connectivity* was ideal for the hectic runway environment where devices were regularly removed and added to garments. Lack of topology constraint brought great flexibility to hardware composition. The small form factor of the Beads made them prone to being lost in garment transit and here, *hardware abstraction* offered by JACDAC also proved beneficial. Beads with the same function could simply take the place of the lost Beads without any change to software.

Applications for Brains were written in MakeCode and special blocks were added to the editor to process data from input Beads and send commands to output Beads. Here, the *dynamic device detection* offered by JACDAC played a critical role. Electronics for garments could be prototyped and composed dynamically during prototyping, and easily dismantled and reassembled without any concern for topology. Combined, MakeCode and JACDAC made dynamic hardware composition more intuitive to technically experienced fashion designers.

5.6.2 Gaming

There has been a recent resurgence in retro-style gaming. Retro-gaming returns not as a tool for entertainment but as a tool for creativity. A number of programming environments for hand-held games consoles have appeared allowing technologists to create their own 8-bit retro games [72, 36, 3, 6, 31, 41].

MakeCode Arcade [258] applies our extensive work in education (via the micro:bit) to retro games programming. Citizen developers can use modern APIs and simple programming languages (Blocks or JavaScript) to build retro games. As building a game involves mathematics, physics, computer science, and art, Arcade also doubles as an education tool. As part of the experience, citizen developers can test their games out on physical hardware and there are now thousands of physical devices compatible with MakeCode Arcade. CODAL (Chapter 4) underpins every one of these applications.

Arcade devices can be easily networked together using an audio jack cable for on-the-go multiplayer communication (Figure 5.15). JACDAC underpins all communication between arcade devices and due to its tight integration with MakeCode, multiplayer operation can be added to games through the addition of a single visual programming block. On-the-go multiplayer would not be possible without JACDAC's multi-central operation and support for dynamic connectivity. The dynamic device discovery offered by JACDAC even allows users to attach external sensors and actuators to Arcade devices. These sensors and actuators

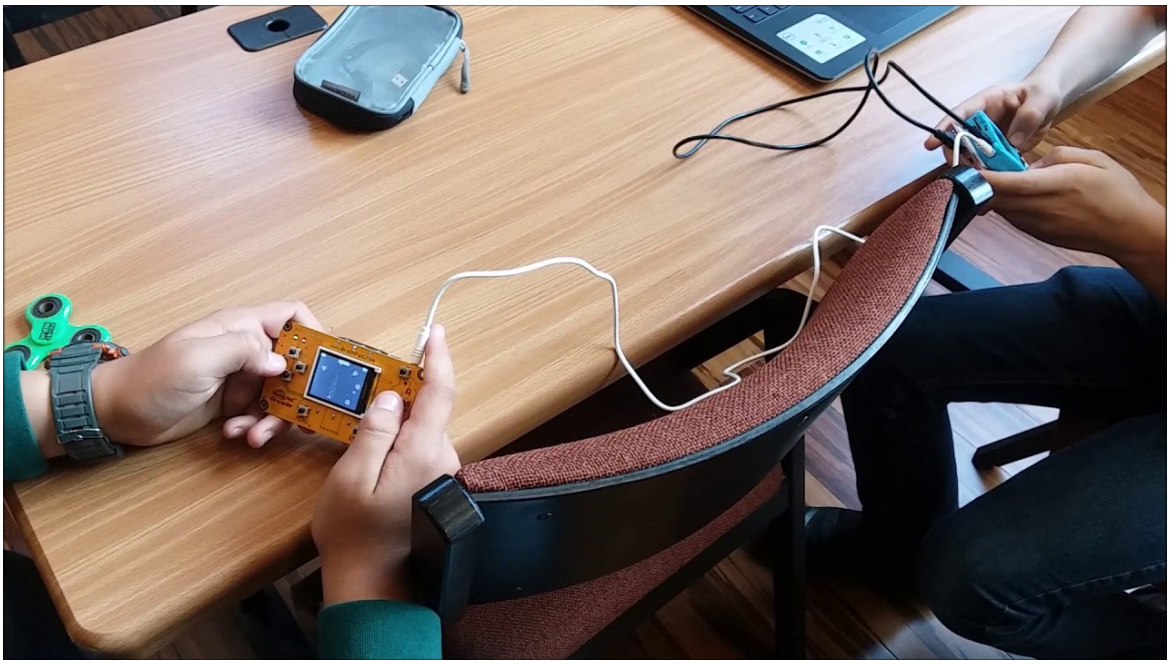

Fig. 5.15 JACDAC networking two different MakeCode Arcade devices together over 3.5mm audio jack

can be used as inputs and outputs to games, blending the physical world with the digital. Here, the dynamism offered by JACDAC and the ease of programming offered by MakeCode create an intuitive and simple experience for dynamic hardware composition by citizens.

5.7 Summary

This chapter has presented JACDAC, a single-wire protocol for intuitive hardware composition. From the ground up, each JACDAC protocol layer is designed to make hardware composition and application development more intuitive: *services* bring *hardware abstraction* so that applications can be written without reference to specific hardware (HC3); the *control layer* brings *dynamic device detection* so that packets can be automatically routed to services without application intervention (HC2); and the *physical layer* brings *dynamic connectivity* so that composition can happen at any time in any environment (HC1).

Applications for JACDAC devices can be written using MakeCode via the JACDAC TypeScript stack and CODAL. This means that citizens can develop applications for JACDAC networks intuitively from any device with a web browser and a USB port (GP1). Through reusing the JACDAC TypeScript stack and combining it with WebUSB, the web browser can even participate as its own JACDAC device (GP1, GP2), providing a universal environment

for extending, developing, and debugging JACDAC services (GP3). Moreover, the standardisation of services services leads to a more introspective and easy-to-understand debugging experience for citizens (GP1).

Through our evaluation we show that JACDAC supports these features at the small expense of efficiency when compared to I2C (GP4). We also demonstrate that JACDAC can be applied to a variety of high and low capability microcontrollers for a cost efficient, yet intuitive, hardware composition experience (GP2, GP3, GP4).

Finally, we provide detail on two existing applications of JACDAC by citizen developers. The first describes how JACDAC made hardware composition simple enough for fashion designers to use at a high profile fashion show in Brooklyn, New York (GP1, GP2). The second demonstrates how JACDAC is used to enable on-the-go multiplayer for citizens with MakeCode Arcade devices (GP1, GP2).

Chapter 6

Droplet: intuitive wireless networking

Across Chapter 2 and 3, we identified properties of existing technologies that make wireless networking more intuitive:

WN1 *No configuration*: Protocols like WiFi require just a name and password to network devices and protocols with minimal or no configuration are therefore more intuitive to citizens.

WN2 *No infrastructure*: The ubiquitous adoption of WiFi means that citizens do not have to install their own infrastructure in order to network devices. Protocols that do not require the installation of any additional infrastructure are therefore more intuitive to citizens.

WN3 *Supports interactivity*: Physical computing devices are increasingly used for interactive applications that require low latency connectivity. Protocols therefore have to supportive of interactive applications.

Our in depth exploration of wireless protocols illustrated that energy efficiency, memory efficiency, and distributed operation are paramount in the world of wireless networking. This is especially true for memory constrained physical computing devices that are typically distributed across environments and powered by battery. We identified a number of ad-hoc wireless networking protocols that meet many of these requirements but found none that are memory and energy efficient, whilst being configuration and infrastructure free.

This chapter introduces Droplet, a configuration and infrastructure free ad-hoc protocol for wireless networking. Droplet uses concurrent flooding to guarantee a high degree of reliability without compromising on flexibility and latency. Low latency operation is also supported by a distributed scheduler and network clock that carefully manages access to

the shared wireless channel. The scheduler and network clock are also used to efficiently manage the radio hardware allowing Droplet to support these features at the expense of a small amount of energy efficiency when compared to BLE.

We begin by providing an overview of the protocol in Section 6.1. We then discuss the distributed scheduler (Section 6.2) and network clock (Section 6.3), followed by a description of how Droplet networks are formed (Section 6.4) and how errors are detected (Section 6.5). We then describe the implementation of Droplet on NRF51822 microcontrollers (Section 6.6) and evaluate the performance of this implementation in Section 6.7. Finally, we describe how droplet was used by citizen developers in 30 schools across the UK in Section 6.8 and summarise our findings in Section 6.9.

6.1 Protocol overview

Droplet is designed for commodity Bluetooth Low Energy (BLE) radio transceivers. These BLE transceivers typically cannot transmit and receive at the same time, and careful software management is required to place transceivers into receiver or transmitter mode at the correct time. BLE radios also typically do not have the ability to check if another device is already transmitting (i.e. transmission collision detection). Collisions generally cause interference and to reduce the likelihood of collisions, and therefore interference, BLE transceivers usually frequency hop (as discussed in Section 3.3.2).

Droplet is based upon concurrent flooding and this requires that transceivers operate on a single frequency band to create a flood-able broadcast network. But without transmission collision detection nor the opportunity to frequency hop, it is likely that multiple devices will transmit over one another creating interference. Droplet therefore divides time into a discrete number of slots, an approach generally known as Time-Division Multiple Access (TDMA). Devices can contend for ownership over each slot and only the slot owner is allowed to transmit within each. A distributed scheduler manages slot ownership and by listening to transmissions in each slot, devices can synthesise the current network schedule.

Since the scheduler is closely bound to the time domain, each device needs to also synchronise to a common time base known as the *network clock*. Other concurrently flooded protocols usually require a dedicated device to provide the network clock, but Droplet's network clock is also distributed. Each device can synchronise to the network easily by listening to the first packet sent within each slot.

Both the network clock and the scheduler are designed for energy efficiency and they are used to optimally manage the radio hardware to reduce *radio on time*. The distributed

Table 6.1 Droplet packet format

Field size (bits)	Field name
8	length
8	slot_identifier
4	frame_identifier
4	flags
4	ttl
4	initial_ttl
64	device_identifier
8 * size	payload

design of the scheduler and network clock allows citizens to create networks without prior configuration or infrastructure deployment.

Droplet requires device communicate using a standard packet structure designed to support flooding. This packet structure is presented in Table 6.1. Bytes above the payload field are considered the packet "header" and the *length* field represents the total size of the packet, including the header. The *Time to Live (TTL)* field dictates the number of times a packet will repeated throughout the network as in Glossy. The maximum size of the 'payload' field is 243 bytes and the remaining fields will be explained in detail as this chapter progresses.

As Droplet relies so heavily on the foundational work of Glossy [148], it is recommended readers familiarise themselves with the work before continuing. We provide a short description of Glossy in Section 3.3.4.

6.2 Distributed scheduler

Each Droplet network has a distributed scheduler used to manage channel contention and reduce radio on time. The schedule, as depicted in Figure 6.1, is made up of a number of discrete time *slots* and each slot has an owner that, for the duration of the slot, has exclusive control over the channel. For every slot that has an owner, all devices in the droplet network must wake and prepare to receive and forward packets sent by the slot owner. For slots with no owner, devices can leave their transceivers disabled, leading to increased energy efficiency. A complete cycle of all time slots in the schedule is known as a *window* and the droplet schedule defaults to a window size of 50 and a slot duration of 20 milliseconds. This amounts to a 1 second long window, which can be adjusted by users (only if required).

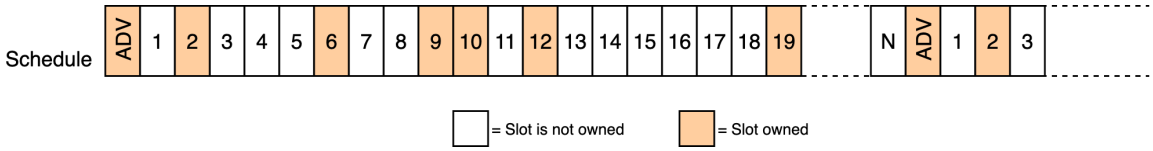

Fig. 6.1 A Droplet schedule consisting of N slots. Advertisement slots are depicted by 'ADV' and standard slots are depicted by a box containing the corresponding slot identifier. Boxes coloured orange have an owner, whereas boxes with no colour do not. Slot ownership corresponds with transceiver activity and unowned slots allow devices to leave transceivers disabled.

There are two types of slot in Droplet: *advertisement slots*, and *standard slots*. Advertisement slots allow devices to express ownership over a slot and are therefore used to propagate schedule changes to all devices in a Droplet network. Standard slots, on the other hand, refer to a slot that already has an owner. Whilst anyone can initiate a flood in an advertisement slot, only a slot owner can initiate a flood in a standard slot. Activity within each type of slot also differs and will be discussed in the upcoming sections.

6.2.1 Advertisement slots

At the beginning of each window, devices can contend for slot ownership in a dedicated advertisement slot. Schedules can contain more than one advertisement slot per window, and during an advertisement slot any device can advertise ownership over any currently unowned slot. All droplet devices consider the advertisement slot an 'owned' slot despite the slot not having any designated owner. This therefore means that all devices in a network must wake and prepare to receive and forward packets.

Packets sent during an advertisement slot are marked with the 'ADVERT' flag and contain a list of one or more *slot identifiers*—a zero based index into the schedule—to be owned. Upon receiving an advertisement packet, devices update their local copy of the schedule, associating each slot in the advert with the 64-bit *device_identifier* contained in the packet. The schedule will then cause devices to wake when the newly owned slots come around. To ensure schedule changes are widely propagated, the TTL of an advertisement packet is set to five regardless of network diameter (i.e. number of hops). This number should more than accommodate most applications but is automatically adjusted if the actual network diameter exceeds five hops.

As no device controls the advertisement slot, there is opportunity for more than one device to initiate a flood at the same time. Without attention, this would likely result in

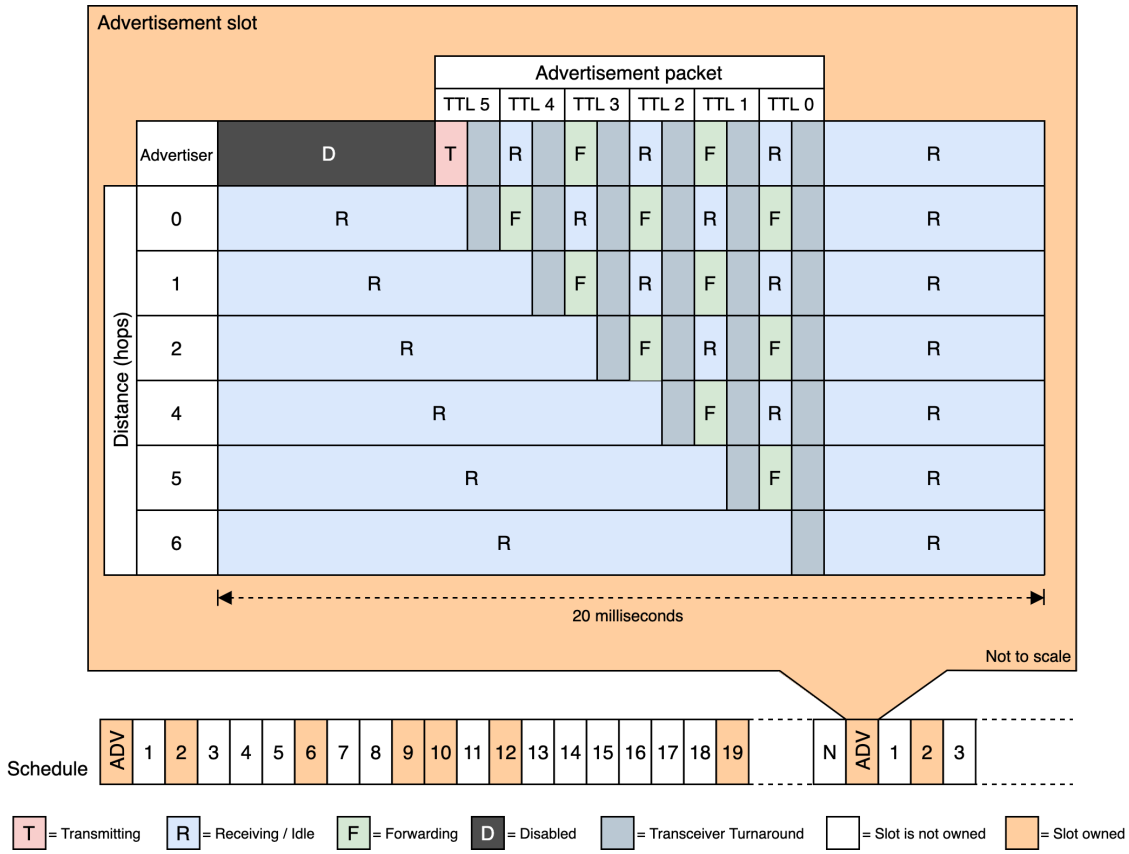

Fig. 6.2 Transceiver activity for a Droplet advertisement slot at different distances (hops) away from the Advertiser. For context, the current schedule is shown across the bottom.

collision during packet propagation, especially if devices are powered on simultaneously. Droplet mitigates advertising collisions in two ways. The first mitigation strategy makes use of a random back off between advertisement slots, where devices use a random number generator to select one of the next five upcoming advertisement slots. The second mitigation strategy makes use of a random transmission back off within an advertisement slot. By distributing advertisements across time both between and within advertisement slots, the opportunity for advertisement collisions is substantially reduced.

Figure 6.2 shows an expanded view of transceiver activity during an advertisement slot at different hops in a droplet network. In the figure, there are four possible roles for a device that correspond to transceiver state. Transmitting (‘T’) means that a device has its transceiver in transmit mode and is initiating a flood. Receiving / idle (‘R’) means that a device has its transceiver configured to receive and it is either waiting to receive or is actively receiving. Forwarding (‘F’) means that a device has its transceiver configured to transmit and it is

repeating (or forwarding) a received packet. Disabled ('D') means that the transceiver is disabled and no packets are sent or received. This terminology is also used in later figures.

The figure shows an advertiser initiating a flood to express ownership over a slot. After the initial transmission, the advertisement packet is concurrently forwarded five times until the TTL reaches zero. The figure also shows the transceiver state of devices at various distances (hops) in the network. The transceiver of the advertiser remains disabled until the random back off time as been met. All other devices leave their transceivers in receive mode when not forwarding a packet.

6.2.2 Standard slots

In a standard slot, a single device (the owner) is given absolute control over the wireless channel. Singular ownership removes issues of channel contention and allows owners to perform multiple back-to-back floods without interruption (as long as schedule slot duration is not exceeded). Each transmission packet that initiates a flood is assigned a *frame_id* that is unique within the scope of the current slot. Receivers keep track of each packet and record frame_ids as they arrive to prevent duplicate reception.

Owners lose ownership of a slot if no packet is sent for five consecutive windows. Each device maintains a counter for each slot that is reset to zero upon successful packet reception. Upon the counter reaching five, devices locally mark the slot as unused, allowing another device to express ownership of the slot in any future advertising window. If an owner has no data to send but wants to retain ownership of a slot, it may flood the network with a zero length packet with the 'KEEP_ALIVE' flag set.

There is a well established link between transceiver activity and energy consumption, and the only time energy is not consumed by the transceiver is when it is disabled. Droplet therefore pays more attention to transceiver state during a standard slot in order to achieve energy efficiency gains over other flooding-based protocols.

Energy efficiency gains are the result of two observations made during the design of Droplet. The first observation is that setting the TTL to a value beyond the current network diameter leads to unnecessary excess repetitions. Excess repetitions reduces network throughput and also leads to greater energy consumption due to increased transceiver activity. Many prior flooding protocols have recognised this problem and addressed it through techniques like probabilistic flooding. Droplet, however, addresses wasteful energy consumption by simply *honing the TTL to the current diameter of the Droplet network*.

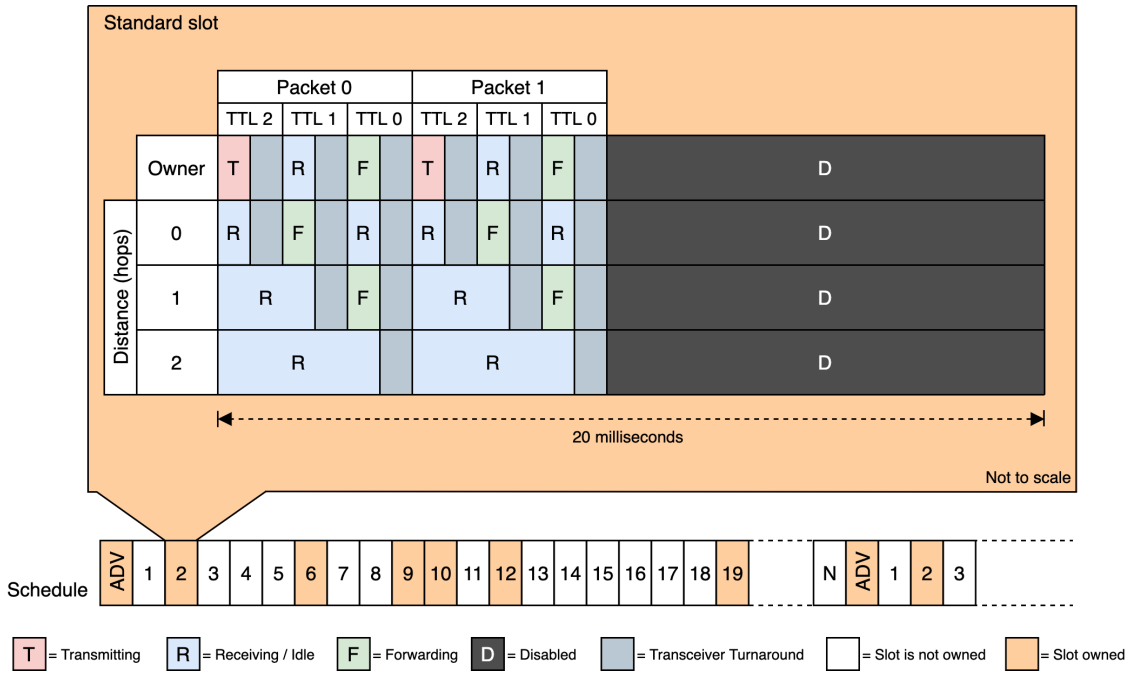

Fig. 6.3 Transceiver activity for a standard Droplet slot at different distances (hops) away from the slot owner. For context, the current schedule is shown across the bottom.

The current network diameter is calculated and set at the beginning of each window using metadata received in packets from the previous window. The necessary metadata is contained inside a droplet header and is a simple case of subtracting the received *ttl* from the *initial_ttl*. Subtracting one from the other gives an indication of the number of hops a packet has travelled and this information is stored in the schedule for each slot for later processing. A max function executed over the schedule yields the current network diameter. It is important to note that his technique is also applied to advertisement slots so that Droplet networks can adapt to nodes added at the extremes of the current network diameter.

The second observation is that transceivers only need to be active for as long as the slot owner has something to send. By marking the final flood in a slot with the 'DONE' flag, an owner can put an entire Droplet network to sleep. This leads to more optimal energy usage as the network adapts to the throughput demand of individual devices. Combined, both optimisations mean that *energy consumption intuitively scales with respect to application throughput and scale*.

Figure 6.3 shows transceiver activity for a standard slot at different distances in a Droplet network. The terminology used in this diagram was described previously in Section 6.2.1. The droplet network in this example has a network diameter of one hop, and the schedule

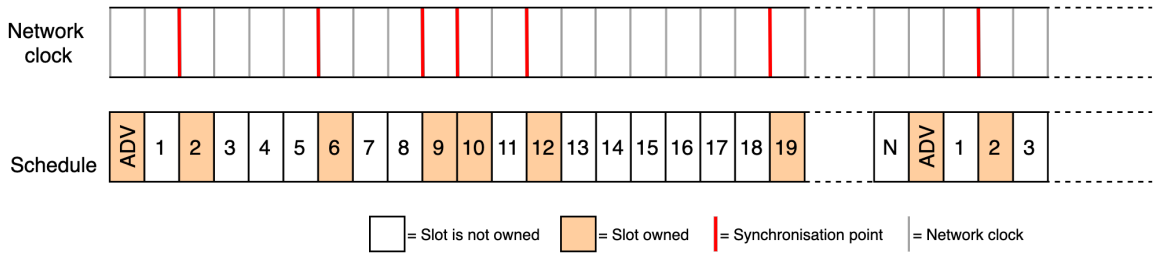

Fig. 6.4 The network clock in context with the Droplet schedule. Grey lines show the regular network clock and red lines indicate points where clock synchronisation occurs. Clock synchronisation only occurs in slots with an owner.

correctly identifies that a TTL of two will ensure reasonable propagation for a network of this size. Floods are initiated back-to-back allowing for maximal throughput with respect to network diameter. The owner has no more data to send after 'Packet 1', and this final transmission is marked with the 'DONE' flag. All nodes in the network respect this flag and disable their transceivers for the remainder of the slot upon flood completion.

6.3 Distributed network clock

The Droplet schedule is closely bound to the time domain and requires each device wake at the appropriate time to receive and forward packets. A network clock is therefore required to synchronise nodes to a common time base. Unlike other flooding-based protocols, Droplet does not require a dedicated device to maintain the network clock. The clock is instead *maintained among devices in the Droplet network*. This means that Droplet networks do not require any supporting infrastructure to operate.

Devices must maintain a local time that is synchronised to the network clock to drive the scheduler and place the radio transceiver into transmit or receive mode appropriately. Each device therefore requires a dedicated hardware timer. Hardware timers must firstly be *synchronised* to the network clock at power on, so the local time is in step with the network clock, and once synchronised, continuous *correction* must be made to account for local drift from the network clock. Synchronisation and correction can only take place in a standard slot that has an owner and Figure 6.4 displays a network clock alongside a schedule to show synchronisation points.

6.3.1 Synchronisation

In Droplet, network clock synchronisation is achieved *without adding any maintenance overhead*. All the information a device needs to perform network clock synchronisation can be extracted from the very first packet sent by an owner in a standard slot. Careful observation of Figure 6.3 reveals that in a standard slot, owners must always begin transmission at the commencement of a slot (i.e where *time* = 0 within the slot). By computing packet transmission time (including any network hops) and subtracting that number from reception time, receivers can calculate the precise time a packet was sent. This calculation is presented in Equation 6.1 and using this equation, all nodes can compute the start time of the slot regardless of proximity. As all slots have a fixed duration in a Droplet network, it is a simple case of addition to calculate when the next slot begins.

$$\begin{aligned} hops &= ttl_initial - ttl_actual \\ t_{tx} &= (packet_{length} + preamble_{length} + crc_{length} + address_{length}) \times t_{symbol} \\ t_{arrive} &= t_{tx} + (hops \times (t_{tx} + t_{radio\ turnaround}) \\ t_{start} &= t_{end} - t_{arrive} \end{aligned} \tag{6.1}$$

6.3.2 Correction

Even though microcontrollers that support BLE have highly-precise 32-bit timers, these timers are still subject to drift. This drift can cause timers to lose up to a millisecond of time every second, and without accounting for this drift, Droplet networks would quickly fall apart. Rapidly correcting for clock drift however would make the problem even worse as devices may incorrectly align to devices that are out of step with the network clock.

Droplet devices therefore compensate clock drift by *averaging relative drift over time*. This causes clocks to converge slowly, leading to a highly accurate, yet distributed, network clock. To account for the worst case clock drift (in the case of an empty schedule), devices wake 500 microseconds before an owned slot. Anecdotally, however, we have observed that this 500 microsecond period is often unnecessary and local clocks *drift by only an average of 15 microseconds*.

6.4 Dynamic network creation and discovery

Like in any ad-hoc network protocol, network creation is dynamic and occurs without user intervention. The network creation and discovery approach adopted by Droplet is designed

for flexibility. When powered on, all Droplet devices first enter *initialisation mode* which configures transceivers as receivers to detect packets from any existing network. Upon the reception of a packet from an established network, the device synchronises to the network clock and enters *discovery mode*.

Discovery mode lets devices discover the existing schedule of the network and leaves transceivers in receive mode for the duration of each slot. In this mode, a device can only forward packets and does not contend for the ownership of any slots. As each packet contains the slot identifier (*slot_identifier*) for the current slot, it is easy for a device to synthesise the entire network schedule locally. After listening for a complete window the schedule is considered synchronised and a device can then contend for one or more slots.

If after two seconds a device remains in initialisation mode, a device establishes a new network by transmitting an advertisement packet with a slot identifier of 0. Devices must implement a random back off to guard against multiple simultaneous network creation.

6.5 Error detection

Without any feedback on network connectivity it is hard to create a reliable and robust network—especially for novices. Droplet therefore detects packet propagation issues throughout its operation giving users real time feedback on link connectivity without adding any overhead to the protocol. This makes device deployment simple, responsive, and efficient.

Every network flood provides a means to analyse connection quality to other nodes, but more importantly a means for a device to gauge the success of its own packet propagation. After initiating a flood, an owner can roughly approximate transmission success by the number of flooding rounds it participates in (obviously with respect to TTL). If an owner sees its packet repeated/forwarded multiple times, then it can assume with a degree of certainty that the packet was propagated successfully.

Measuring packet propagation is not suitable for detecting all errors however. Packet loss can lead to incomplete schedules and occasions where more than one device owns the same slot. Droplet therefore has other mechanisms to detect these conditions.

6.5.1 Incomplete schedules

Occasionally packets may be missed due to environmental interference or other factors and this may cause a device to miss an advertisement if this happens during an advertisement

slot. This can lead to an incomplete schedule which can be detrimental to an application—especially if the device in question bridges a partition in the network.

To prevent an incomplete schedule, devices occasionally enter *listen* mode. Listen mode is similar to discovery mode and places the transceiver into receive mode for the beginning of every slot in a window. The frequency a device enters listen mode can be tailored to the application, but defaults to 1 window in 10.

6.5.2 Slot collisions

It is possible for devices to contend for the same slot simultaneously and therefore end up owning the same slot (i.e. slot collision). If more than one device owns the same slot, packet propagation will be hampered for all owners of that slot.

Other devices in the network provide the means to detect slot collision. If for a particular slot devices consistently see a high density of CRC errors, in the next available advertisement window, devices will send a packet containing the identifier of the slot and the 'ERROR' flag set in the flags field. Upon reception an error packet, all devices must remove the given slot from their schedules and the owners of that slot must re-contend for a new one.

6.6 Implementation

This section describes an implementation of Droplet for the Nordic NRF51822. It is is built using CODAL and is later used for evaluating Droplet in Section 6.7. The implementation has two main components: the *scheduler* that maintains and oversees the schedule; and the *protocol driver* that manages the hardware.

6.6.1 Hardware

As a demonstration of the real-world applicability of Droplet, we implement the protocol on the BBC micro:bit. The main processor of the micro:bit is a 16 MHz Nordic NRF51822 microcontroller with 16 kB of RAM and 256 kB of flash. Though the microcontroller has little memory resource, it is based on the extremely capable ARM cortex-m0 processor architecture. The microcontroller can operate as a BLE peripheral through the inclusion of a software stack called Nordic SoftDevice. Without SoftDevice however, applications are free to use the low-level radio hardware interface to transmit packets at 2.4 GHz and we use this radio interface for our implementation.

Table 6.2 Droplet slot representation in memory.

Field Size (bits)	Field Name
64	device_identifier
8	slot_identifier
8	expiration
4	distance
4	flags
8	errors

Each peripheral (i.e. UART, I2C, SPI) on the NRF51822 has a set of tasks and events. Tasks represent an operation (i.e. power on, ready the receiver) and events are emitted as a result of peripheral activity. Software developers make use of events during interrupts to determine what specific activity has occurred. Events can be connected to tasks to trigger peripheral operation *without processor intervention*. Module 'short cuts' enable this functionality within the same peripheral. For instance, upon receiving a UART byte (RX_END event), a short cut can be used to trigger the UART module's TX_START task for automatic local echo. The Programmable Peripheral Interconnect (PPI) module extends this idea to allow *any* peripheral event to be connected *any* peripheral task. This means that a timer event can be used to trigger a UART transmission without processor intervention. We make use of short cuts and PPI throughout our implementation of Droplet.

6.6.2 Scheduling

All Droplet devices maintain an individual schedule that is synthesised during discovery. In memory, the schedule is represented as an array of slots and contains the fields shown in Table 6.2. The schedule is continuously updated during operation and every packet is passed to the scheduler for meta data extraction.

Each time a packet is passed to the scheduler, the following process is observed. First, if the packet has an incorrect CRC, the 'errors' field is incremented and no further action is taken. Otherwise, the 'slot_index' in the Droplet packet is used to retrieve the relevant slot stored in the schedule. The scheduler then performs a basic validity check against the device identifier received in the packet and the one stored in the schedule. If the stored slot is initialised and the device identifiers do not match the 'errors' field is incremented and no further action is taken.

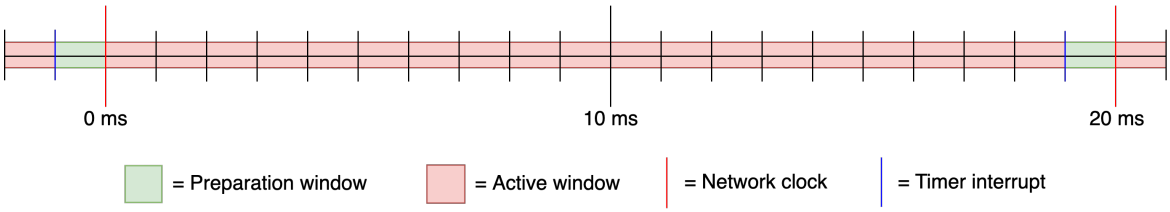

Fig. 6.5 The scheduling quantum of the timer and the network over a 20 millisecond slot.

Next, the 'expiration' field for the slot is reset to its initial value. The expiration field is a counter that is used to determine if a device has left the network. Once a second an asynchronous callback decrements slot expiration counters and when they reach zero, the FREE flag is set in the 'flags' field. The FREE flag indicates to the scheduler that this slot can be used for any future allocation. This process is lightweight and simple to implement even on a resource constrained microcontroller like the NRF51822.

Finally, device 'distance' is also updated. The protocol driver always computes the number of hops a packet has travelled upon successful reception. When a packet is passed to the scheduler, the number of hops is also passed as a parameter and this is stored in the schedule.

The scheduler provides many APIs to inspect and query the schedule. For instance, the scheduler separately maintains the current slot index and ensures it is synchronised with the network using packet meta data. The scheduler also maintains the individual distance to each node and this distance is used to optimise flooding. The protocol driver regularly queries the scheduler to determine whether it is an owner or participant in an upcoming slot and the optimal distance for flooding. This division between the schedule and protocol driver decouples individual concerns.

6.6.3 Keeping time

The scheduler is tightly coupled to the time domain using a dedicated hardware timer. This timer drives the scheduler after it synchronises with the scheduling quantum of the network clock (i.e. 20 milliseconds).

Timer interrupts are not in phase with the network clock however and are configured to occur 1000 microseconds before the next slot begins. This 1000 microsecond period is known as the *preparation window* and its relationship with the network clock is depicted in Figure 6.5. The preparation window is used to move the scheduler forward onto the next slot and to determine the hardware configuration for this slot. A preparation window of

1000 microseconds means that the effective usable portion of a 20 millisecond slot is 19 milliseconds. This bigger portion of a slot is known as the *active window*.

In the timer interrupt, slot metadata stored in the schedule is used to determine the role of a device for the upcoming slot. The device role dictates the configuration of the radio hardware. If the slot is free or uninitialised, then the radio hardware is disabled (if not already) and the timer schedules itself to wake in one scheduling quantum. If the slot metadata indicates a device is the owner of the upcoming slot, the radio hardware is configured begin transmission in exactly 1000 microseconds (at 0 milliseconds in the next slot). Finally, if a device is just a participant for the upcoming slot, the radio hardware is configured for reception in 500 milliseconds. The extra 500 milliseconds is used to account for network timing variations.

PPI is used to connect timer events to radio transceiver tasks, allowing timer to place the transceiver into receive or transmit mode after a fixed time offset. Radio module short cuts are also used to trigger transmission and reception after the transceiver has been enabled in either mode. This means that transmission and reception can occur at a fixed point in time *without processor intervention*, especially important when spare clock cycles are limited and timing is critical.

6.6.4 Managing radio state

There are two kinds of radio state to manage in Droplet: high level state and low level state. High level state represents the higher level operation of the hardware, and low level state represents the physical state of the radio hardware. A high level state is therefore composed of many low level states.

High level states

Droplet effectively has three high level states for the radio hardware:

1. *Transmitting*: A device is considered the slot ‘owner’ and is beginning a flood.
2. *Receiving*: A device is not the slot owner and is preparing to receive a packet.
3. *Forwarding*: A device has received a packet and is preparing to forward it.

These three higher level states are combined to produce a flood. There are only two valid start states: receiving and transmitting. The preparation window is used to determine which high level state is required. A slot owner will start in the ‘transmitting’ state, swap to ‘receiving’ after transmission, and switch repeatedly between ‘forwarding’ and ‘receiving’

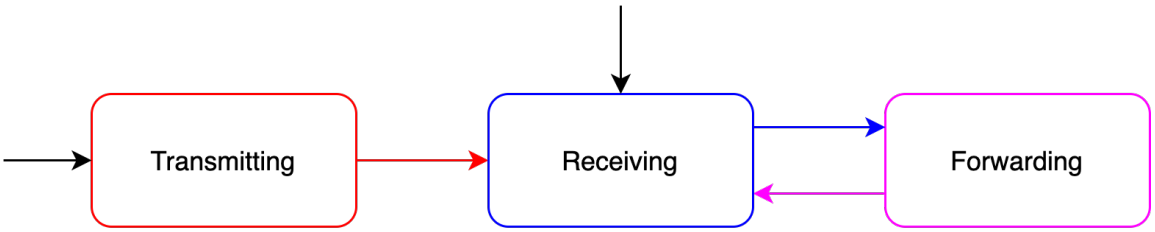

Fig. 6.6 Possible transitions between the higher level states of the radio module. Only the 'transmitting' and 'receiving' states can be used as starting points.

until the TTL reaches 0. Any other device will start in the 'receiving' state and swap between 'forwarding' and 'receiving' until the TTL reaches zero. Figure 6.6 summarises the permitted transitions between these states.

Low level states

The radio hardware transitions through many states for each high level state. Figure 6.7 shows the low level state machine of the radio hardware and the different paths taken when in different high level states. Solid arrows indicate that state transition is manual, whereas a dashed arrow indicates that state transition occurs without processor intervention (i.e. using short cuts).

One path through the state machine (Figure 6.7) is of particular note. The 'forwarding' path consists of many manual states compared with the entirely automatic 'receiving' and 'transmitting' paths. This is because state transmission is sometimes temporally variable which is challenging for concurrent transmissions that require timing accurate to within 100 nanoseconds. We achieve highly accurate (to within 50 nanoseconds) by using a fixed duration spin loop to manually swap between states for the 'forwarding' path. This spin loop is consistent between devices and happens in interrupt context at the highest priority and so cannot be pre-empted by any other task.

As can be seen in Figure 6.7, both reception and transmission are bound to an interrupt routine. This routine is invoked after receiving, transmitting, or forwarding a packet and is used to transition the hardware into the next permissible state (as per Figure 6.6). By the time the interrupt routine is invoked, the hardware has already reached disable mode, resetting the state machine for the next high level state.

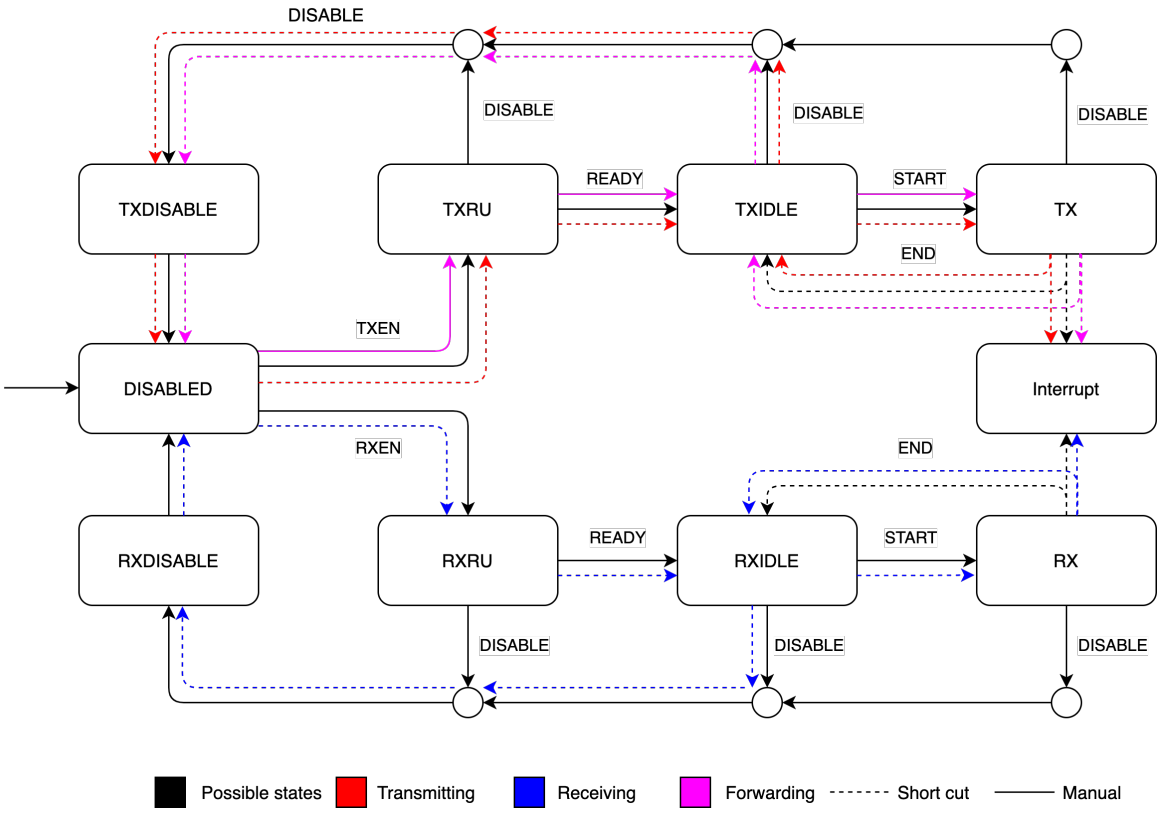

Fig. 6.7 The state machine diagram for Droplet on the NRF51822. Black connections outline standard radio paths, whereas other colours outline paths for the different high level states.

6.6.5 Reception and transmission

The protocol driver uses a fixed length pool of buffers for both transmission and reception. Buffers are moved from the pool to transmission and reception queues as required. Pre-allocated buffers are a necessity as we found that allocating buffers on demand was a costly process, especially given the time critical nature of the Droplet protocol.

When receiving a packet, the hardware stores data into a fixed allocation reception buffer, and when reception is complete, it generates a software interrupt. This software interrupt copies data from the fixed reception buffer to another buffer taken from the pool. This buffer is subsequently moved to the reception queue, which is drained through an asynchronous callback via the CODAL event bus.

At any time applications can call an API to transmit a packet. This API copies data from a provided buffer into a buffer from the pool. The pool buffer is subsequently added to the transmission queue and at the next owned slot, the timer copies the buffer from this queue into the fixed buffer for transmission. The packet is removed from the transmission queue by a software interrupt that is generated when transmission is complete.

The frequency at which packets are added to the transmission queue governs how many slots a device contends for. Transmission frequency analysis occurs every window, and a device will contend for a new slot if required and the schedule permits. Of course, it takes time for transmission frequency to reach equilibrium after a new slot has been obtained and Droplet accounts for this by preventing new allocations for 5 consecutive windows after an allocation.

Currently Droplet does not define the payload of packets, allowing higher level protocols to be easily layered on top. The current Droplet implementation places a 16-bit identifier in the first 2 bytes of the payload and maps this number straight through to higher level drivers. In the future, we plan to integrate the service model of JACDAC from Chapter 5.

6.6.6 Error detection

In Droplet, the network can be used to gauge the success of a flood. After a slot owner initiates a flood, any future reception during the flood is likely to be a repetition of the original transmission. Repetitions are therefore also passed to the scheduler for meta data analysis. If the scheduler notices a difference in device identifiers or a CRC error is detected, the corresponding error counter for the slot is incremented.

Periodically, an asynchronous call back analyses the schedule and takes note of any slots with a high number of errors. The callback averages error counters across the entire schedule

to guard against the case where a device is placed in an area of high interference. Any slots that exceed the average are reported during the next advertisement window and if a device owns any of the reported slots it contends for a new slot.

6.7 Evaluation

In this section we evaluate various aspects of Droplet. We begin by outlining our methodology (Section 6.7.1) for all experiments and then profile the performance of Droplet at close range (Section 6.7.2). We then increase the range of our experiment and deploy Droplet devices across a large space (Section 6.7.3). Finally, we provide other performance statistics of Droplet and compare it to other state of the art concurrent transmission protocols (Section 6.7.7).

6.7.1 Methodology

We use the Droplet implementation (described in Section 6.6) and the BBC micro:bit to perform all our evaluation. For each experiment, we created a network of micro:bits to test the reliability of Droplet in different conditions. Each micro:bit was powered from battery (except where otherwise stated) and was assigned one of four roles, forming part of a distributed application:

Collector One micro:bit was programmed with the collector role in each experiment. The collector operated a different radio protocol on a different frequency band to Droplet. The collector was connected to a personal computer and used USB serial to report results received over the radio to a serial terminal. The serial terminal output was stored for later analysis.

Transmitter Only one micro:bit was programmed to be a transmitter. The transmitter could fully participate in a Droplet network, and could both transmit and forward packets. The transmitter, upon a press of button 'A', sent sequence numbers from 0 to 1000 and recorded the number of repetitions seen for each sequence number. Upon a button 'B' press, the transmitter swapped to the same frequency as the collector and broadcast its results.

Repeater Repeaters could only forward packets, and there were multiple micro:bits with the repeater role in each experiment. Each repeater recorded the number of packets it saw for each sequence number, and, like the transmitter, broadcast these results to the collector upon a button 'B' press.

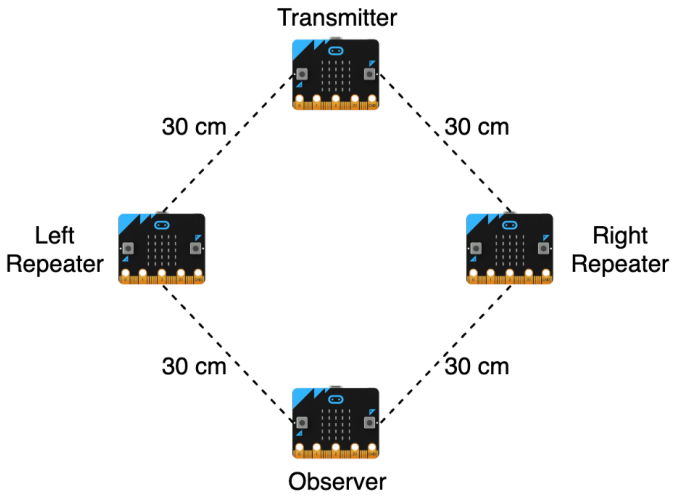

Fig. 6.8 Desk configuration used to test required timing accuracy for successful concurrent transmissions.

Observer Observers were passive, and could only receive packets. There were multiple micro:bits with the observer role in each experiment. Each observer recorded the number of packets it saw for each sequence number, and like the transmitter, broadcast these results to the collector upon a button 'B' press.

Each data point in the upcoming experiments is the average of three runs and between each run, each micro:bit was power cycled. Data points are generated using results reported from each device via the collector and a python script was used to process results and generate graphs.

6.7.2 Behaviour at close range

The first set of experiments we carried out were designed to profile the effect of concurrent transmissions at close range. For this, we removed the scheduler from the Droplet implementation, leaving only software to manage concurrent transmissions. In each experiment, the transmitter was configured to send sequence number every 20 milliseconds, with a TTL of one.

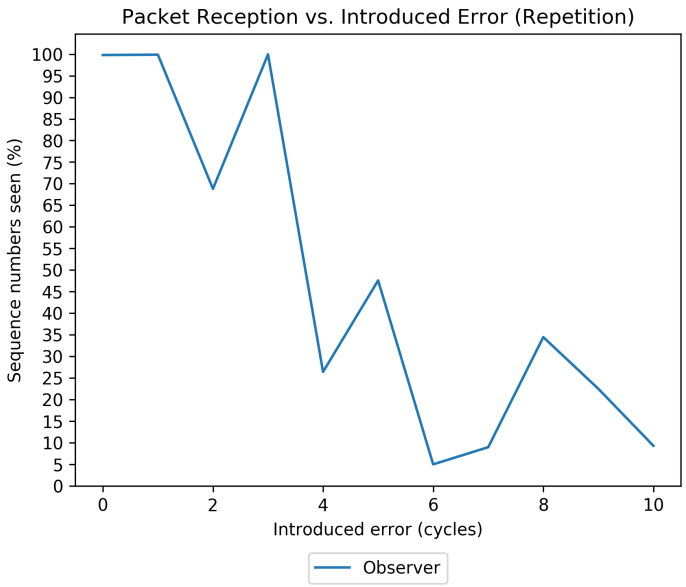

Fig. 6.9 Sequence numbers seen by the observer for repeated packets. As the introduced delay of the right repeater increases, the percentage of sequence numbers seen by the observer decreases

Tolerance to timing variance

This experiment was designed to observe how critical timing is to the success of concurrent transmissions. To find out, we placed four micro:bits equidistant from one another on a desk to form a diamond shape, as shown in Figure 6.8. The top most micro:bit was the transmitter, the two middle micro:bits were repeaters, and the bottom micro:bit was an observer.

We customised the firmware for the right repeater to introduce a small delay to the 'forwarding' path through the radio state machine. Each test added a 'nop' instruction to the path, increasing the time difference by one processor cycle (62.5 nanoseconds) each time. A 'nop' instruction equates to one processor cycle and we added 10 'nop' instructions in total. Each measurement was performed three times and each device was powered cycled before each run. The averaged results from the observer are displayed in Figure 6.9.

The figure shows a clear trend: as the introduced delay of the right repeater increases, the percentage of repetitions seen by the observer decreases. This result shows that for the micro:bit, implementations can only drift by exactly one processor cycle (62.5 nanoseconds) before having a detrimental affect on concurrent transmissions. On a faster processor, where instruction duration is shorter, implementations can tolerate more processor cycle drift (as long as they stay within the 62.5 nanosecond limit).

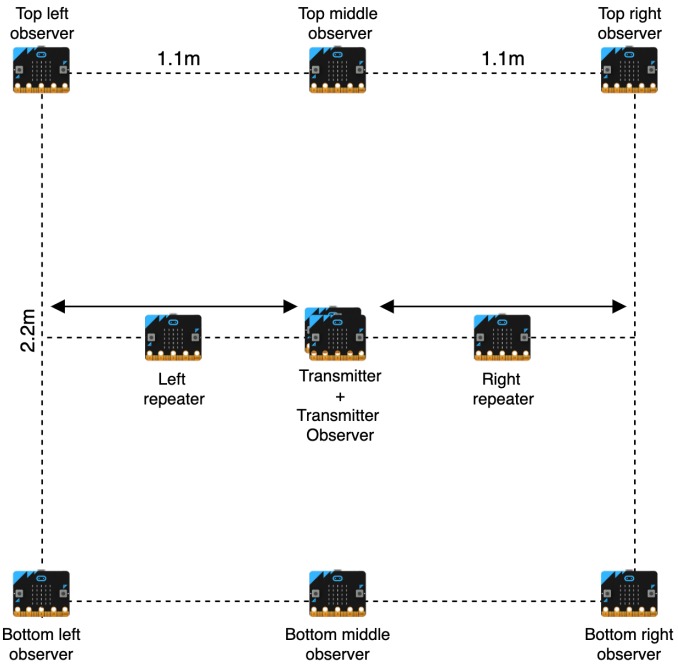

Fig. 6.10 The frame configuration used to profile the effect of proximity on concurrent transmissions.

Tolerance to proximity

This experiment was designed to test the effect of proximity on concurrent transmissions. We constructed a free standing frame made of pine wood visualised in Figure 6.10, right. The frame was composed of 3 horizontal beams and four vertical supports.

To the top and bottom beams we affixed three observers to the left, middle, and right. These micro:bits were mounted to the frame using nuts and bolts. On the middle beam we placed four micro:bits: a transmitter and observer were placed in the centre, and two repeaters were placed on either side. The repeaters could be freely positioned along the beam. All micro:bits were powered from a bench power supply set to consistently provide 3.3 volts.

Each test changed the relative distance between the repeaters and transmitter starting from .1 metres finishing at .9 metres. Each repeater was moved separately and incrementally. We repeated each test three times, and devices were power cycled before each run. The results of the tests are displayed in Figure 6.11.

Each point along the x-axis in Figure 6.11 is a tuple containing the relative distance of each repeater from the transmitter. The left repeater is on the left side of the tuple, and the right repeater on the right. Starting equidistant, each repeater was moved in step, becoming

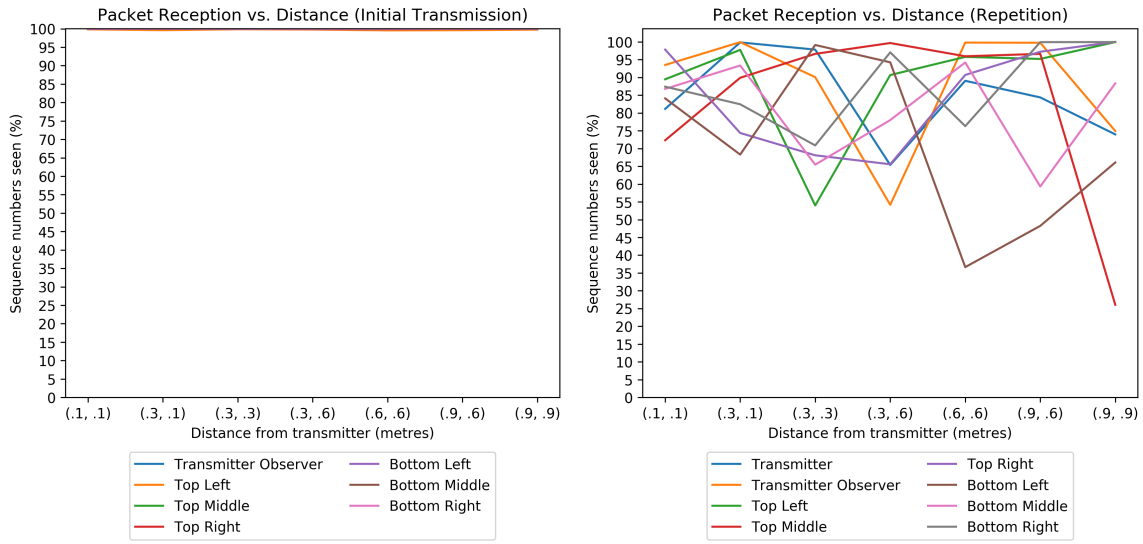

Fig. 6.11 Sequence numbers seen from all observers from the original transmission (left) and the repetition (right), as distance increases.

equidistant at various points during the test. The y-axis displays the percentage of sequence numbers seen, with the left graph displaying the sequence numbers seen from the initial transmission, and the right graph displaying the sequence numbers seen from the repetition.

From the results shown in Figure 6.11, we can only conclude that there is no clear correlation between proximity and the success of concurrent transmissions. In fact, the results were so sporadic that we re-ran the experiment with only one repeater to verify that concurrent transmissions were indeed the cause.

We suspect that there are many factors affecting the success of our experiment. Through further isolation of our test set up however we could not identify any single cause. The authors of Blueflood [225] attribute reception variability to the phase of carrier signals, which is beyond the control of software. An important observation to make however, is that even with variable reception of repetitions, all devices saw each sequence number at least once.

6.7.3 Behaviour at longer ranges

This set of experiments were designed to profile Droplet at longer ranges. We first provide a metric of base reliability, then demonstrate the impact of increasing device density and the number of repetitions. For all of these experiments we used a full Droplet implementation with a schedule slot duration of 20 milliseconds.

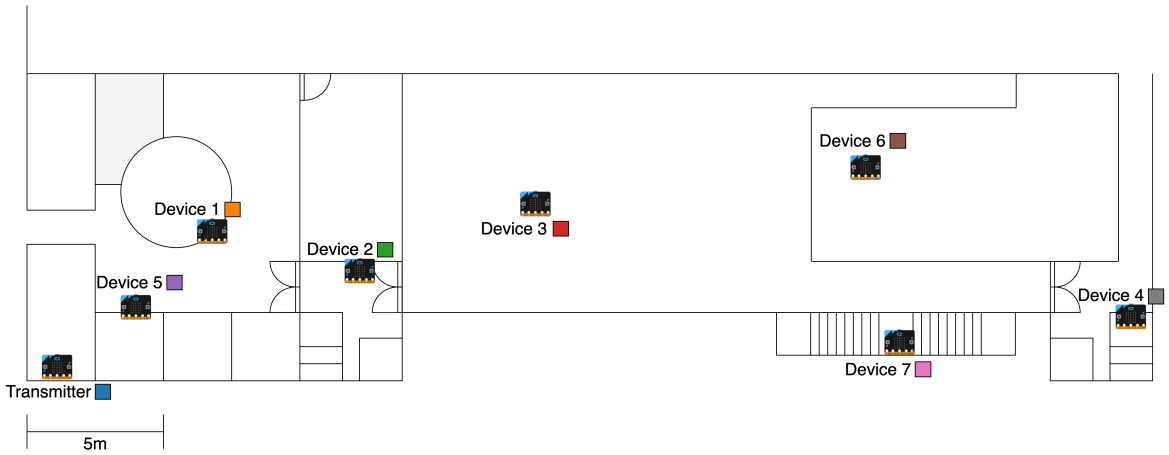

Fig. 6.12 The deployment floor plan, containing placement of all devices used in the upcoming experiments, and spanning over 40 metres in distance. Coloured boxes beside each device label correspond to colours used in graphs.

Base reliability

This experiment was designed to test the reliability of Droplet at longer ranges. We therefore deployed 5 micro:bits across 40 metres in a technologically and constructionally dense environment. We measured the standard communication range of the micro:bit in this environment as between 5 and 10 metres.

A floor plan showing the placement of micro:bits is displayed in Figure 6.12. There was no particular care taken to deployment of devices, each having different heights and orientation, placed whenever and wherever possible. The environment itself has different properties, from the technologically dense academic offices on the left, through to a cavernous glass-encased cafe in the middle, finishing in the equally technologically dense business offices on the right.

Like prior experiments, there was only one micro:bit with the transmitter role (labelled Transmitter in the figure), the four other devices in the experiment were repeaters (labelled Device 1–4 in the figure). All devices were running the full Droplet protocol, and for repeatability of results, we modified the schedule on each device to have 12 slots statically allocated to the transmitter, and one slot allocated to each repeater. The schedule had a slot duration of 20 milliseconds and there were 50 slots in a window (1 second window). Devices were added to the Droplet network incrementally and at each stage we recorded the number of sequence numbers seen by each device. Each increment was repeated three times and averaged.

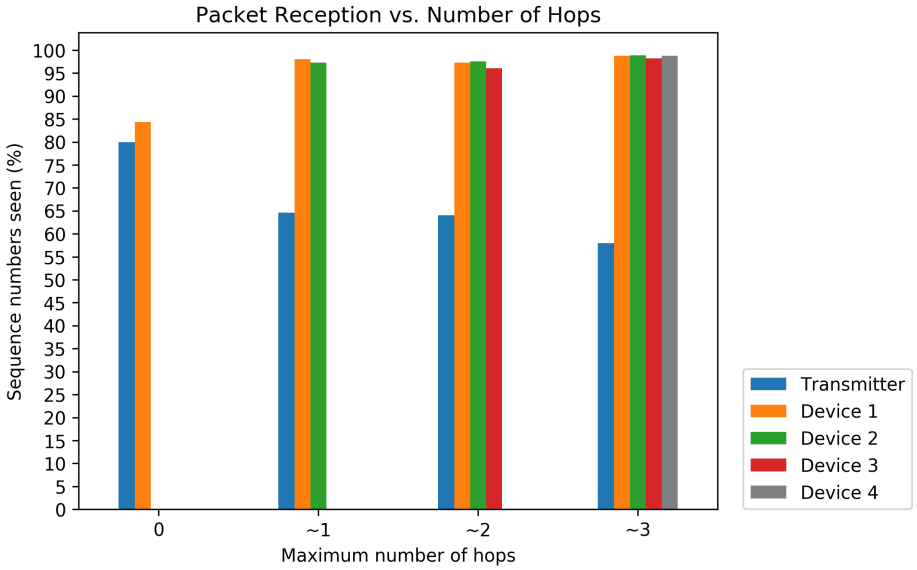

Fig. 6.13 Sequence numbers seen by devices. The reliability of the network increases as nodes are added.

Figure 6.13 shows the percentage of sequence numbers seen by the transmitter and devices 1 to 4. As the number of devices participating in the network increases, the reliability of the network improves. The transmitter is the only device which sees reception degradation as more devices are added. This is because the transmitter effectively sees only repetitions from other devices. The close proximity of devices 1 and 2 may also cause the capture effect, adversely impacting repetition reception rate.

Increasing device density

This experiment was designed to see how increasing node density affected reliability. We therefore added three more devices to the network, labelled devices 5, 6, and 7. These devices also ran the full Droplet implementation. Devices were added to the final phase of the prior experiment, and we repeated this experiment three times.

Figure 6.14, left, shows the effect of adding more devices. As can be seen, increasing node density only has a negative impact on device 4. We believe that this degradation due to the capture effect caused by the equidistant positioning of devices 6 and 7. The addition of device 5 significantly improved the reception rate of the transmitter (to over 90%), lending more weight to the capture effect theory.

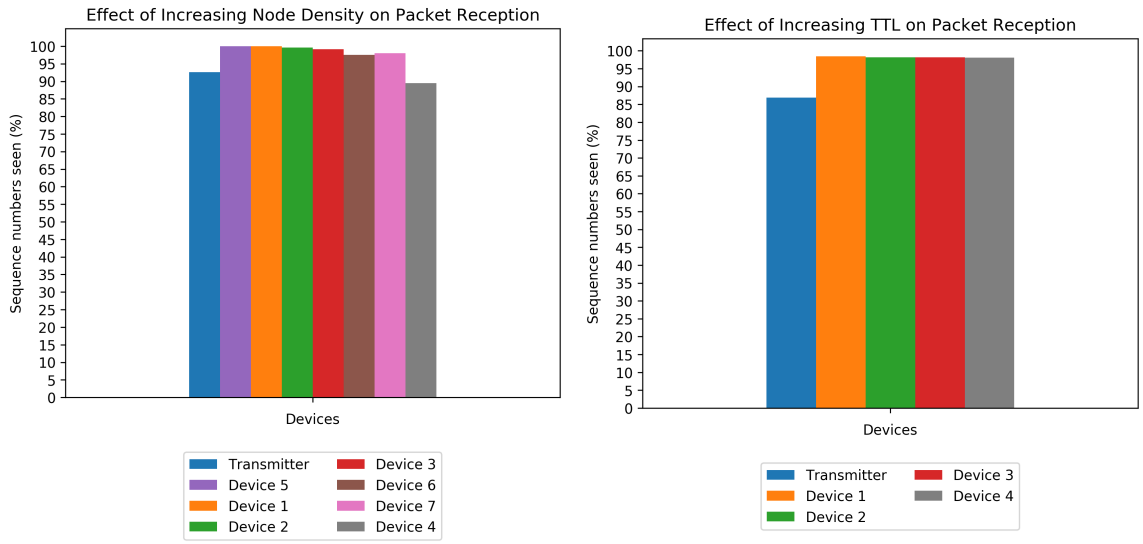

Fig. 6.14 The effect of adding nodes to the network (left) and the effect of increasing the default TTL to 4 (right) has on the percentage of packet numbers seen.

Increasing repetitions

Adding more devices to a network comes at financial cost, and so we designed an experiment to see if we could increase reception success without adding any devices. We therefore removed devices 5, 6, and 7, and increased the TTL of each packet sent by the transmitter to four. Figure 6.14, right, shows the effect of this modification. By increasing the TTL, the reception rate for all devices improves.

6.7.4 Transmission time and latency

We used two micro:bits to profile the transmission time and latency of Droplet. One micro:bit was programmed as a transmitter and the other as a repeater. The firmware for both micro:bits was also programmed to toggle a GPIO on the micro:bit's edge connector when a packet was sent or received. We then attached a logic analyser to the edge connector of each micro:bit to visualise the produced waveforms.

The total transmission time for a 20 byte packet was 238 microseconds. A keen reader might observe that a 20 byte transmission should take just 160 microseconds. We found that BLE hardware adds additional metadata to each packet including a: 1 byte pre-amble, 5 byte address, 1 byte prefix, 1 byte length, and a two byte CRC. We also determined that it

takes four microseconds between the end of a transmission before reception is propagated to software.

6.7.5 Scalability

The scalability of Droplet depends on its configuration. For prior experiments, each schedule was configured to have fifty 20 millisecond long slots, leading to a total window duration of 1 second. In this default configuration there can only ever be a maximum of 50 devices, and a maximum of 15 packets per slot for a network diameter of 3 hops.

The scheduler can be of course be adapted to applications if required. For instance, an application that requires frequent transmissions across a large number of devices could reduce slot duration to 10 milliseconds and increase the number of slots to a hundred. The scheduler can be similarly adapted to small data rate applications too. We believe however that the default configuration of Droplet should be suitable for most applications.

6.7.6 Memory and peripheral usage

In embedded development hardware peripherals, flash, and RAM are finite. Droplet consumes one timer and one radio peripheral on the NRF51822. In terms of memory, Droplet consumes 5 kB of flash memory and a total of 3.14 kB of RAM. Of this 3.14 kB, the schedule statically consumes 900 bytes (for 50 slots) and 2.24 kB is assigned to the packet buffer pool. As a reference point, Bluetooth Low Energy (BLE) on the same processor consumes the same two hardware peripherals, 128 kB of flash, and 10 kB of RAM.

6.7.7 Energy consumption

Whilst performing long-range testing we also profiled the energy consumption of the transmitter as nodes were added to the network. Measurements were taken through attaching an ammeter in series at the transmitter. We categorise our measurements by two states: *idle* and *active*. We define the idle state as devices maintaining the network, sending only keep alive packets and an active state as the transmitter actively sending sequence numbers from 0 to 1000.

Figure 6.15 (left) shows a linear increase in energy consumption as the number of hops increases. This is expected as the radio is on for longer. This concept is intuitive and easy to explain to novices in two statements: (1) The larger the network, the more energy is consumed; *and* (2) the more packets transmitted, the more energy is consumed.

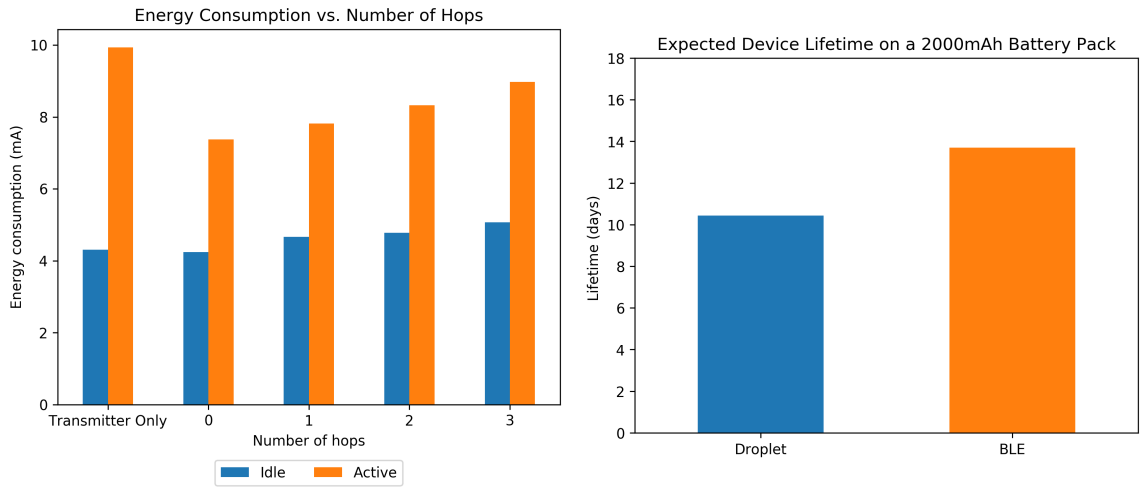

Fig. 6.15 Energy consumption of Droplet in the idle and active states (left); and expected lifetime on a 2000 mAh battery using Droplet compared to BLE (right).

Comparison with BLE

We also compare the energy consumption of Droplet to BLE. We programmed a single micro:bit to appear as a BLE enabled accelerometer. We then connected to the micro:bit from a smart phone application using BLE and the application then polled accelerometer values every 100 milliseconds (10 packets per second). We measured energy consumption using an ammeter placed in series at the micro:bit, and computed the energy consumption for each packet. For a fair comparison, we multiplied per packet energy consumption in BLE by the number of transmissions per second in the Droplet active state. We did *not* include repetitions as part of this computation and measurements for Droplet were taken from the previous experiment of a network with 3 hops in the active state. Figure 6.15 (right) shows the results.

The figure shows that Droplet is less efficient than a point-to-point BLE connection, but the difference is less than one might have thought. Droplet offers many other benefits such as no configuration, no infrastructure, ad-hoc networking, and we therefore believe these benefits far outweigh the loss of some energy efficiency.

Comparison with other concurrent flooding protocols

Finally, we compare JACDAC with other state of the art concurrent flooding protocols. A direct comparison can not be made due to hardware incompatibility—prior work uses the 802.15.4 physical layer. These protocols do however provide a device agnostic means of

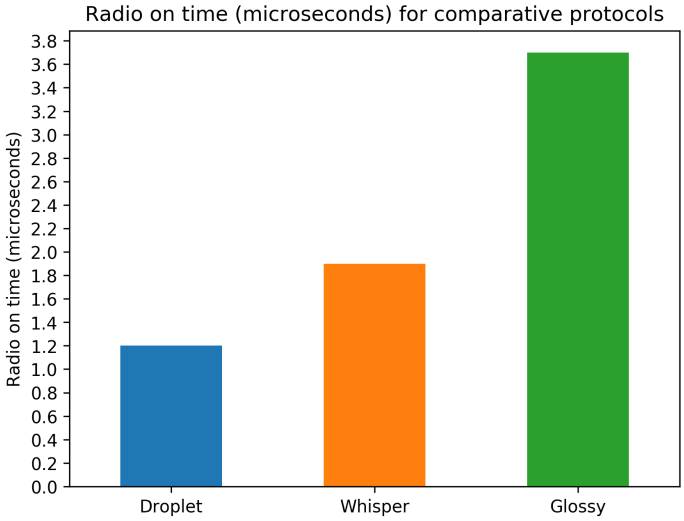

Fig. 6.16 Radio on time of Droplet compared to other protocols.

measuring energy efficiency: *radio on time*. We computed radio on time using the values measured whilst implementing Droplet on the NRF51822. We sum the total active time for a 20 byte packet for a network diameter of 3 hops.

Figure 6.16 (right) displays the radio on time of Droplet compared to Glossy [148] and Whisper [119]. Glossy has the longest radio on time of 3.7 milliseconds, followed by Whisper at 1.9 milliseconds, and Droplet with the shortest radio on time of 1.2 milliseconds. Droplet's shorter radio on time can be attributed to its faster, 1 megabit data rate.

6.8 Applications of Droplet

Reducing carbon emissions for the benefit of ourselves and future generations is a well-recognised global challenge. Schools are estimated to be the second largest consumer of non-domestic energy, making them a lucrative candidate for energy savings [29]. However, in organisations such as schools consisting of many stakeholders with differing agency, it can be hard to enact change. It is easy to see why—to reduce energy consumption, behavioural change is required, which means all stakeholders (senior leadership, teachers, pupils) in a school must be made aware of energy consumption and prompted to actively reduce it.

We took part in the Energy in Schools (EiS) project which aimed to reduce the energy consumption of 30 schools by prompting behaviour change. The project took a two pronged approach to engage all stakeholders within the school. The first aimed to provide real time

Fig. 6.17 Students creating an IoT application in the classroom

energy consumption statistics to all stakeholders. Senior management were able to intuitively access real time energy consumption statistics for their schools from a HTTP web portal and energy consumption and temperature data was reported to this portal from sensors installed in each school building. This allowed senior management to immediately and easily identify wasteful usages of energy. To make students and educators more aware of energy consumption, displays showing real time energy consumption were placed in prominent locations in each school. Displayed energy consumption was also gamified to encourage healthy competition between schools.

The other key part of the EiS project was its educational offering. Lesson plans, consumer IoT devices (e.g. smart bulbs and switches), and micro:bits were provided as part of the project. These resources were used to create engaging lessons where students could wirelessly control IoT devices using the micro:bit and MakeCode. IoT devices were connected to the EiS web portal and controlled using commands sent from the micro:bit to the EiS web portal. The micro:bit however is an extremely capable device itself, it has a variety of built in sensors and can even be embedded into custom creations. We therefore made it possible for micro:bits to retrieve and supply data to the web portal for senior management to see. This allowed teachers and students to conduct their own scientific IoT experiments and create their own visualisations for energy consumption data.

We used Droplet to make wireless communication with the web portal intuitive to technically inexperienced teachers and students. micro:bits could be programmed using

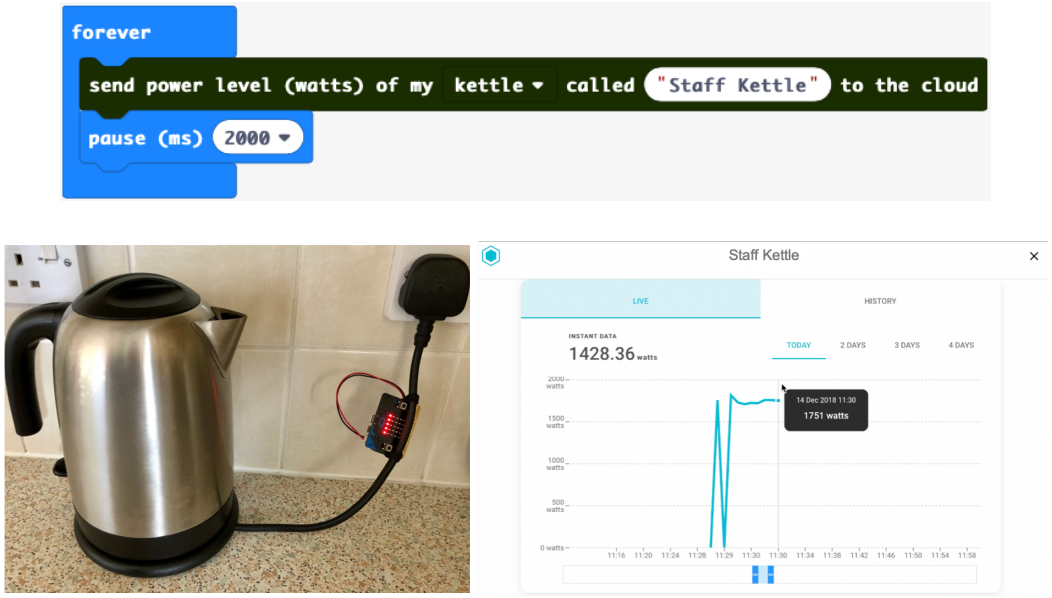

Fig. 6.18 A MakeCode application (top) to measure the induced current of a kettle (bottom left) using the micro:bits magnetometer every two seconds. Results are reported to the EiS web server and visualised (bottom right).

Blocks from MakeCode to send messages to the EiS web portal via Droplet. One micro:bit (usually the teachers') acted as the Internet uplink for the Droplet network, forwarding packets onto the EiS web portal when required. The uplink micro:bit was connected over USB serial to a no installation web application running on the teachers computer. This web application made a secure connection to the portal, performed web requests, and returned responses to the Droplet network. The use of Droplet meant that no configuration and no infrastructure was required to network micro:bits in the classroom.

Intuitive classroom IoT

The combination of MakeCode and Droplet allowed students and teachers to flexibly and intuitively create their own IoT deployments. To evidence how intuitive MakeCode and Droplet made wireless networking and the IoT, we provide a sample of the advanced applications students and teachers created.

Citizen science

As the micro:bit could be used as a sensing device for the EiS portal, many applications incorporated elements of citizen science. One example application makes use of the mi-

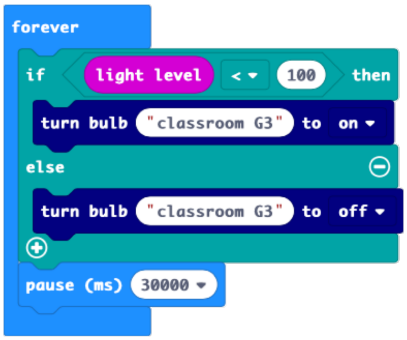

Fig. 6.19 A MakeCode application that turns on a smart bulb when the micro:bit detects the ambient light-level has gotten too dark.

cro:bit's magnetometer. The magnetometer can be used to estimate the energy consumption of applications by measuring the induced current down a wire. Combining this capability with Droplet allowed students and teachers could visualise energy consumption of appliances over time in the EiS web portal. Figure 6.18 (top) shows a MakeCode application that measures the energy consumption of a kettle (bottom left) and reports it to the EiS web portal (bottom right). On the web portal, students and teachers could visualise energy consumption of the kettle over time.

Other example applications make use of the micro:bits accelerometer keep track of door state in order to minimise energy wasted through heat loss. As well as providing a deep educational experience, these applications allow technically inexperienced students to become citizen scientists.

Smart actuation

Other projects turned the micro:bit into an actuator for consumer IoT devices. An example actuation project uses the micro:bit's light sensing capability to control the state of a smart light bulb. Figure 6.19 shows a MakeCode application that turns on a smart bulb when the micro:bit senses that the ambient light level in the classroom has gotten too dark. One again, as well as giving students valuable educational experience, the micro:bit, Droplet, and MakeCode allows technically inexperienced students to become citizen developers.

Cross-curricular visualisations

The final set of example applications obtained data from the portal via the micro:bit and used that data to create more engaging visualisations. Here, the micro:bits small form factor

Fig. 6.20 A MakeCode application that retrieves the current energy consumption of a school.

allowed it to be embedded into custom creations using classroom materials. Figure 6.20 shows a MakeCode application that allows students to retrieve the real-time energy consumption of their school from the EiS web server. Resulting creations could be deployed in the classroom, better engaging students in the goal of reducing energy consumption.

6.9 Summary

This chapter has presented *Droplet*, a configuration and infrastructure free ad-hoc protocol for wireless networking. Infrastructure and configuration free operation is enabled through the use of concurrent flooding and a distributed scheduler and network clock (WN1, WN2). Interactivity is supported by default because of Droplet's low latency propagation and high slot frequency that adapts to the needs of applications (WN3).

Droplet applications can be written from MakeCode using a Droplet implementation written in CODAL. This means that applications can be developed intuitively and without installation (GP1). Droplet also makes no assumption of the throughput demands of applications and automatically balances demand against energy efficiency (GP2). Droplet's responsive error detection also allows for the easy extension of Droplet networks (GP1, GP3).

Through an extensive evaluation we show that Droplet enables the features above whilst maintaining a high degree of energy efficiency (GP4). Finally, we show how citizen developers are already using the intuitive experience offered by Droplet to create custom IoT applications in schools across the UK (GP1, GP2).

Chapter 7

Conclusions

Across the globe citizens want to partake in technological innovation. These *citizen developers* have the drive to solve problems with technology, but not necessarily the technical expertise to do so. To benefit society, and promote inclusivity and diversity, it is therefore important that tools for technological innovation are made accessible to all.

There have been many efforts to democratise access to technological innovation with software. Intuitive higher level programming languages allow anyone to learn about machine learning and data science without any formal training. No configuration cloud technologies allow anyone to deploy applications across the globe without significant infrastructure investment. And no code development environments allow anyone to build and deploy web applications across the globe without writing a single line of code. These advances make it easier than ever before for citizen developers to partake in software innovation.

A similar trend of democratisation can be observed for hardware. Using *physical computing* technologies citizens can now build interactive devices and systems that can sense and respond to the real-world—*and they are doing just that*. Citizen makers and hobbyists are using physical computing for artistic creativity and problem solving. Citizen scientists and innovators are using physical computing to advance scientific understanding and automate day-to-day life. And citizen teachers are using physical computing to provide students with a fun and engaging way to learn computer science. Despite these uses however, hardware innovation via physical computing remains inaccessible to some.

The process of building a physical computing device is riddled with technical complexity. Embedded development boards equipped with low cost, resource constrained microcontrollers form the basis of prototyped physical computing devices. Through microcontroller *programming*, users can create applications that define the functionality of a physical com-

puting device. Resource constrained microcontrollers however require applications to be written in highly efficient but hard to use programming languages.

Embedded development boards also expose microcontroller General Purpose Input/Output (GPIO) for *hardware composition*. Hardware composition involves wiring external sensors to GPIO and this process is what enables physical computing devices to sense and respond to the real-world. Communication between microcontroller applications and external sensors is enabled by highly efficient but hard to use wired protocols.

Some physical computing application require devices network locally and to the Internet and *wireless networking protocols* are commonly used for this purpose. Wireless protocols offer great flexibility but consume energy in their operation and energy consumption therefore becomes an important concern for battery powered physical computing devices. Counterintuitively however, wireless protocols that are easy to use but less efficient are preferred to those that are hard to use but are energy efficient. The use of these less efficient protocols limits physical computing applications.

7.1 Research questions

Motivated by three research questions (RQ1—RQ3) this thesis has explored the trade off between efficiency and ease of use for existing technologies for physical computing. Each question corresponds to one of three areas (programming, hardware composition, and wireless networking) and we answer each question in turn in the upcoming sections.

7.1.1 Programming (RQ1)

RQ1 What are the capabilities, characteristics and limitations of event-based visual programming languages when applied to microcontrollers? How do these languages' capabilities and performance compare with the state-of-the-art in supporting citizen developers?

Chapters 2 and 3 provide an extensive exploration of programming languages. These chapters showed that citizen developers find higher level languages, particularly those that offer event-based (P2) visual programming (P1), more intuitive. These chapters also revealed an emergent trend towards web-based, installation-free (P3) programming. Lack of software installation makes programming accessible to citizen developers from any device with a web browser.

Visual programming languages for microcontrollers already exist, but none of these languages offer the event-based programming paradigms evidence to make programming more intuitive to citizen developers. Moreover, extensive examination of state of the art programming languages used by professional software developers revealed that microcontrollers require careful management of memory and processor utilisation. This need for processor and memory efficiency becomes even more prevalent for resource constrained, battery powered physical computing devices. Visual programming languages for microcontrollers however typically enable more intuitive programming at the cost of memory and processor efficiency.

This thesis therefore contributes CODAL, a runtime environment that enables installation-free event-based visual programming from Microsoft MakeCode. CODAL provides a number of primitives that allow higher level languages to efficiently map to the hardware. Through an extensive evaluation we show that MakeCode and CODAL are up to 50 times more processor efficient than other popular languages, and that the approach can be applied to resource constrained microcontrollers with as little as 2 kB of RAM. We also illustrate how intuitive MakeCode and CODAL make programming by reporting on the many advanced projects citizen developers create, and sharing statistics that show that over one million citizen developers now use MakeCode and CODAL every month.

7.1.2 Hardware composition (RQ2)

RQ2 Do single wire approaches to modular hardware composition simplify hardware integration for citizen developers, and what are the performance implications of these approaches?

Chapters 2 and 3 provide an extensive exploration of wired protocols. This exploration revealed that citizen developers find wired protocols that offer dynamic connectivity (HC1), dynamic device discovery (HC2), and hardware abstraction (HC1) make hardware composition more intuitive. Furthermore, protocols that enabled this feature set using a single cable also proved more intuitive.

Wired protocols that enable low infrastructure hardware composition already exist but are not widely adopted by low cost and resource constrained microcontrollers. Extensive examination of these protocols also revealed that those that had intuitive features did so at the expense of implementation complexity. As implementation complexity leads to an increase in cost, we found that protocols shown to be more intuitive to citizen developers are also not widely supported by low cost microcontrollers.

This thesis therefore contributes JACDAC, a single-wire protocol for intuitive hardware composition. JACDAC reuses the UART peripheral common to most microcontrollers and its software stack can be applied to implemented on high and low capability microcontrollers. JACDAC supports all of the intuitive features required by citizen developers at the expense of a small amount of efficiency when compared to I2C. JACDAC devices can be intuitively programmed from MakeCode and services developed and debugged from any device with a web browser and USB port. We show that JACDAC enables intuitive hardware composition by describing its use as part of a modular toolkit for building interactive garments. Using JACDAC citizen developers were able to intuitively compose hardware to construct interactive garments that were later shown at a high profile fashion show in Brooklyn, New York.

7.1.3 Wireless networking (RQ3)

RQ3 Do concurrent flooding approaches simplify ad-hoc wireless networking for citizen developers and what are the performance implications of these approaches?

Chapters 2 and 3 provide an exploration of wireless networking protocols relevant to the needs of the citizen developer. This exploration revealed that citizen developers require protocols that support interactive physical computing applications (WN3) without requiring configuration (WN1) or supporting infrastructure to operate (WN2). These chapters also revealed that citizen developers typically deploy networked physical computing devices on battery across environments. But instead of using protocols designed for distributed and energy efficient operation, they choose to use proximal and energy intensive protocols that offer the intuitive features above.

Ad-hoc networking protocols offer distributed and energy efficient operation, but we found that many of these protocols require configuration and supporting infrastructure to operate. Other ad-hoc networking protocols were also designed to sacrifice real time interactivity for energy efficient operation. Through examination of different types of ad-hoc networking protocols, we found that concurrent flooding approaches seemed to offer many intuitive features that citizen developers require. Existing approaches however required infrastructure to support their operation.

This thesis therefore contributes Droplet, a configuration and infrastructure free ad-hoc protocol for wireless networking. Droplet uses concurrent flooding to flexibly support distributed interactive applications that adapt to the throughput demands of applications automatically and without intervention. Using the BBC micro:bit, we show that Droplet is implementable on commercially available BLE radios. Using this implementation we

evaluate Droplet's performance and show that Droplet enables all of the intuitive features required by citizen developers at the expense of a small amount of energy efficiency. We show that Droplet makes wireless networking more intuitive by describing its use by citizen developers to create advanced IoT applications in 30 schools across the UK.

7.2 Concluding remarks

Working on the micro:bit project motivated many of the contributions and learnings of this thesis. Close work with teachers and students revealed an appetite for learning and creating with physical computing technologies, but similarly revealed the challenges and issues faced by many citizen developers.

It has been extremely humbling to learn that over 20 million children have now programmed a micro:bit using MakeCode and CODAL—that is 1% of all children *in the world*! It is important however to recognise that close research collaboration were critical to this success. Collaborators offered considerable help in defining, refining, and motivating ideas and concepts. Without these very different perspectives, it would have been difficult to sort interesting ideas from the inane. The results of this thesis are evidence that close research collaborations can change the world for good.

The technologies in this thesis were not only built, they were deployed and used by the people they were designed for. This process proved extremely valuable in directing future research. A need for more intuitive wireless networking emerged through working with educators, and their use of edge connector peripherals for the micro:bit evidenced a need for more intuitive hardware composition. End users therefore also play a critical role in defining and shaping research direction.

Each time researchers open Google Scholar they are greeted with the phrase "stand on the shoulders of giants". Sometimes, however, it is necessary to reach the shoulders by scaling up the body of the giant. This thesis has shown that by investigating the fundamental technologies of physical computing, rather than their application, physical computing can be made more intuitive.

7.3 Limitations

Though the technologies contributed in this thesis have made physical computing more intuitive, it is important to acknowledge their limitations. This section briefly discusses the limitations of each technology in turn.

CODAL

CODAL is designed to allow higher level languages run efficiently on microcontrollers. Many higher level primitives like events and managed types are supported in CODAL. This not only makes it easier for higher level languages to map efficiently to C++, it makes easier for developers to write applications in C++.

These features are designed to give users a safe platform on which to build applications, but each feature increases processor and memory utilisation compared to a plain C application. This means that CODAL cannot be used for real-time applications that require sub-microsecond accuracy or for super resource constrained microcontrollers.

JACDAC

There are many different types of peripherals available for microcontrollers to use. Many of these peripherals are sensors that have the capability to occasionally emit events (e.g. an accelerometer shake event) or report data when requested. Other types of peripherals like microphones, speakers, and displays regularly stream large quantities of data to or from microcontrollers. JACDAC is ideal for low data rate sensors, but not for supporting high data rate streaming-based interfaces like displays.

Droplet

Droplet was designed for teachers and educators to use. It is dynamic, flexible, and ad-hoc and uses a time-division multiple access approach to give each device a dedicated time to transmit. The number of slots available is fixed which means that if more devices are added than can be supported, those devices will get no air time.

Like JACDAC, Droplet is best suited to applications that do not require high data rate streams. For example, streaming live video is likely not possible with Droplet. There are ways around this limitation in both cases however. If the end goal is image recognition, that recognition could happen on the camera module itself rather than processed elsewhere on the network. Moreover, the data could be pre-processed so to reduce the amount of data sent across a network.

7.4 Future work

JACDAC has the potential to transform many applications areas beyond those discussed in this thesis. It brings a paradigm shift not seen before in the low-cost microcontroller market and removes the constraints of traditional communication protocols.

Flexible hardware production

It is often the case that electrical components selected for consumer devices may not be available throughout the expected manufacturing lifetime of a product. For example, the micro:bits produced today are different from those manufactured in 2015. The original micro:bit accelerometer, the MMA8653, was declared end of life in 2017. This meant that after a specific date, no new MMA8653 accelerometers would be produced and supplies of the part would dwindle.

Hearing this news, the micro:bit foundation were forced to choose a different model of accelerometer to fit to the micro:bit. They also decided to future proof themselves further by selecting a second accelerometer to act as a back up to the first. With two accelerometers to choose from, factories could make the decision of which accelerometer to fit based upon supply chain availability bringing agility and flexibility to manufacture.

Supply chain flexibility and agility is important for all parties involved in hardware production. If the new accelerometers required a significant change in PCB layout, the micro:bit foundation would have incurred costs for Federal Communications Commission (FCC) recertification. Similarly, factories configured to produce micro:bits, but lack the required components, do not make any money.

If JACDAC was natively supported on every sensor, component selection would become arbitrary. In the case of the micro:bit, it would not have mattered what accelerometer was available in the supply chain, *any* accelerometer could have been selected as a replacement.

Sustainable hardware production

Sustainability should be high on the agenda for anyone involved in producing electronic products. Vital planetary resources are consumed each time a component is produced, and factory emissions from production processes contribute significantly to rising CO2 levels.

One way we could reduce the impact of hardware production on the environment is by modularising the design of PCBs. Modularisation at the PCB level would allow disposed and disused electronic products to be broken down into individual discrete modules that could be

re-purposed and reused in new electronics products. JACDAC's hardware abstraction and support for modularity would be a key enabler to this approach.

Democratising hardware production

It is often the goal of the prototyping process to produce a small number of artifacts. But if those artifacts are successful, the next step may be commercialisation. Prototyping as a production process may scale to tens of products, but it cannot scale much beyond that. When selling to the mass market, there are certain quality expectations that must be met. And to meet these expectation more refined production processes are required which typically require huge investment from the creator.

This is true of many areas of manufacturing and especially so for electronic products—building a single low quality prototype is relatively easy, but producing high quality products at high volumes (e.g. 100,000+) can only be achieved with strong financial backing. High quality products produced in smaller quantities (100–100,000) often come with a hefty price tag. A quick glance at Kickstarter [23] however will evidence demand for niche products aimed at these smaller quantities, but their quality is typically less than that of a mass manufactured electronic product.

This niche area is known as the long tail of hardware [174] and making it easier for people to produce hardware in smaller quantities has now be declared an open research challenge. There are many research areas encapsulated in this challenge, but one area which JACDAC could have a pivotal role is in democratising the manufacture of electronics.

There is a significant amount of skill required to produce a custom PCB. Complexity lies in circuit design—drawing the schematic, designing the PCB layout, and routing electrical signals between components. Using completely modularised JACDAC components would vastly simplify the PCB design process. Only three wires would be required to connect modules together, and designs could even be automatically generated due to topological simplicity. Reusing modular JACDAC components would also address questions of scalability and sustainability—the prolific use of JACDAC modules in the wild would reduce the cost of creating hardware for everyone. I envisage a world where someone can create a piece of hardware for their application, without prior electronics or programming experience.

A new connector

Though the audio connector proved intuitive for fashion designers to use, the use of the connector itself introduced some complexity. The first are of complexity concerned power

delivery. It became clear that shorts occurred as the 3.5mm jack was inserted into the barrel and it was also common for users to partially insert the jack. As power occupied one of the 3 pins of the audio jack, 5 Volts would occasionally be shorted to ground and sometimes for an extended period of time. Hardware components were added to account for this short circuit condition, but these components quickly became hot.

The second area of concern was the huge wealth of 3.5mm compatible audio accessories already in existence. It was highly likely that children, or even adults, would connect their MakeCode Arcade devices to expensive audio systems anticipating 8-bit game sounds. As audio systems are not design to accommodate the 5 Volt and 3.3 Volt data signals of JACDAC, what would most likely happen is that those expensive audio devices would break.

Because of these concerns, the use of the 3.5mm jack become untenable. JACDAC will be using a new connector in the future and it is ongoing work to define its look and feel. The reversibility and the low cost of the 3.5mm jack will be used as design aspirations for a new connector.

Intuitive power delivery

Understanding electronic power delivery and distribution requires a basic understanding of electronics. Many citizen developers do not have such understanding and it is difficult for them to calculate power consumption and identify issues that may arise from insufficient power. Making power more intuitive and easy to understand is an open research challenge that is vital to the success of JACDAC and any other tool or protocol that could be used for hardware prototyping.

Acronyms

.NET MF .NET Micro Framework. 82, 100

2D Two Dimensional. 5, 29

3D Three Dimensional. 30, 44, 86, 105

ABI Application Binary Interface. 78

ABIs Application Binary Interfaces. 78

ADC Analog-to-Digital Converter. xv, 56, 58

ANT Adaptive Network Topology. 66

AODV Ad-hoc On Demand Vector. 118

APC Asynchronous Procedure Call. 132, 133, 144

API Application Programming Interface. xi, 34, 47, 50, 59, 78, 79, 81, 88, 129, 138, 149, 152, 179, 194, 198

APIs Application Programming Interfaces. 1, 21, 44, 64, 78, 79, 81, 82, 84, 86, 95, 99, 105

ARM Advanced RISC Machines. 5, 49, 50, 84, 137, 192

BBC British Broadcasting Coorporation. 40

BLE Bluetooth Low Energy. xvi, xix, 8, 12, 29, 37, 38, 40, 66, 67, 80, 85, 102, 108, 112, 113, 115, 114, 116, 117, 119, 149, 182, 183, 190, 192, 206, 207, 208, 217

BMN BLE Mesh Network. 117

BPSK Binary Phase-Shift Keying. 113

CAN Control Area Network. 92, 93, 97, 106

CAP Contention Access Period. 114

CDMA Code Division Multiple Access. 68

CFP Contention Free Period. 114

CLR Common Language Runtime. 82

CoAP Constrained Application Protocol. 118

CODAL Component Oriented Device Abstraction Layer. xi, xvi, xvii, xx, 11, 88, 128, 129, 130, 131, 136, 137, 138, 139, 138, 139, 141, 147, 148, 149, 150, 156, 158, 170, 179, 180, 192, 198, 213, 216, 218, 219

CPU Computer Processing Unit. 133

CPX Circuit Playground Express. xvi, xvii, 4, 39, 40, 129, 130, 132, 131, 136, 138, 140, 142, 144, 147, 148, 149, 170, 171

CRC Cyclic Redundancy Check. 92, 112, 164, 166, 169, 174, 192, 193, 198, 206

CSMA-CA Carrier-sense Multiple Access with Collision Avoidance. 113, 114

DAG Directed Acyclic Graph. 117

DIY Do It Yourself. xiv, 23, 24, 110, 111

DMA Direct Memory Access. 172

EiS Energy in Schools. xix, 209, 210, 211, 210, 211, 212

EMC Electromagnetic Compatibility. xviii, 166, 172

FCC Federal Communications Commission. 220

FFD Full Functionality Device. 114, 118

Gbps Gigabits per second. 93, 97

GFSK Gaussian Frequency-Shift Keying. 112

GHz Gigahertz. xvi, 63, 112, 113, 192

GP Guiding Principles. 12, 69, 71, 72, 149, 150, 180, 181, 213

GPIO General Purpose Input/Output. xvii, 1, 2, 3, 6, 14, 29, 34, 36, 39, 40, 47, 53, 54, 55, 56, 58, 59, 104, 147, 148, 167, 168, 171, 206, 215

GUI Graphical User Interface. 36, 43, 45, 47, 82

HAL Hardware Abstraction Layer. 49, 82, 84, 86, 105, 125

HALs Hardware Abstraction Layers. 74, 76, 87

HATs Hardware Attached on Top. 55

HTML Hyper-text Markup Language. 44, 84

HTTP Hyper-text Transfer Protocol. 154, 209

I2C Inter-Integrated Circuit. xvi, xx, 7, 11, 53, 59, 60, 70, 73, 76, 94, 95, 96, 97, 99, 100, 101, 104, 105, 106, 107, 125, 126, 146, 165, 170, 173, 174, 181, 216, 238, 247

I3C Improved Inter-Integrated Circuit. 97, 107

IDE Integrated Development Environment. xv, 1, 34, 35, 39, 43, 42, 43, 44, 45, 47, 49, 80, 84, 87, 105

IFTTT If This Then That. xiv, 21, 23

IMU Intertial Measurement Unit. 105

IO Input/Output. 44, 53, 99, 100

IoT Internet of Things. xiv, xix, 2, 4, 8, 9, 13, 18, 19, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 34, 36, 37, 38, 40, 41, 49, 50, 56, 61, 66, 67, 71, 76, 84, 100, 110, 111, 118, 209, 210, 211, 212, 213, 217

IP Internet Protocol. 21, 61, 62, 64, 67, 68

IQ Investigatory Question. xx, 73, 90, 107, 108, 124, 125, 126

IR Intermediate Representation. 77, 130

JACDAC Joint Asynchronous Communications; Device Agnostic Control. xi, xvii, xviii, xx, 11, 107, 151, 152, 153, 154, 155, 156, 157, 158, 160, 158, 160, 161, 163, 164, 165, 164, 165, 166, 167, 166, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 179, 180, 181, 198, 208, 216, 219, 220, 221, 222

JIT Just In Time. 82

JSON JavaScript Object Notation. xvi, 138, 154, 155

JVM Java Virtual Machine. 81, 82

kB Kilobytes. 34, 40, 65, 81, 82, 83, 87, 90, 150, 170, 172, 192, 207, 216

Kbps Kilobits per second. 92, 94, 97

LAN Local Area Network. xv, 61

LED Light Emitting Diode. xvi, 39, 40, 76, 79, 104, 147, 178

LLVM Lower-level Virtual Machine. 77

LTE Long-term Evolution. 20, 38, 61, 68

MAC Medium Access Control. 113

Mbps Megabits per second. 92, 93, 94

MCU Microcontroller Unit. 1

MHz Megahertz. 97, 113, 142, 170, 192

MIPI Mobile Industry Processor Interface. 96, 97

NFC Near Field Communication. 65

O-QPSK Offset Quadrature Phase-Shift Keying. 113

OSI Open Systems Interconnect. xv, 64, 65, 64

PC Personal Computers. 18

PC Personal Computer. 98, 99, 101, 130

PCB Printed Circuit Board. 3, 6, 7, 34, 36, 39, 52, 53, 54, 59, 60, 90, 94, 95, 99, 100, 102, 103, 220, 221

PCBs Printed Circuit Boards. 1, 109

PPI Programmable Peripheral Interconnect. 192, 195

PSR Proactive Source Routing. 117

RAM Random Access Memory. 20, 34, 40, 51, 74, 75, 77, 80, 81, 82, 83, 87, 88, 90, 109, 128, 144, 150, 170, 171, 172, 207, 216

REPL Read Eval Print Loop. 79, 80, 81

REST REpresentational State Transfer. 21, 152, 154, 155

RFD Reduced Functionality Device. 114, 118

RFID Radio Frequency Identification. 65

RGB Red Green Blue. 39, 104, 147, 178

RIP Routing Information Protocol. 118

RQ Research Question. 10, 215

RTOS Real Time Operating System. 49, 50, 75, 76, 77

RTOSs Real Time Operating Systems. 50, 74, 75, 76

SDF Semi-directional Flooding. 121

SIG Special Interest Group. 119

SPI Serial Peripheral Interface. xvi, 7, 53, 59, 60, 70, 73, 95, 96, 100, 104, 105, 106, 125, 126, 146, 165

STS Static TypeScript. 130, 136, 139, 140, 141, 142, 143

TDMA Time-Division Multiple Access. 183

TTL Time to Live. xix, 119, 121, 122, 184, 185, 187, 188, 191, 195, 200, 205

UART Universal Asynchronous Receiver Transmitter. xv, 53, 58, 60, 91, 100, 105, 153, 165, 166, 167, 168, 169, 172, 192, 216

UDID Unique Device Identifier. 160, 161, 164, 165

UDIDs Unique Device Identifiers. 161

UDP User Datagram Protocol. 118

USB Universal Serial Bus. 7, 24, 29, 32, 34, 39, 40, 47, 48, 54, 60, 61, 70, 74, 80, 85, 87, 93, 94, 98, 99, 100, 102, 106, 107, 130, 157, 180, 199, 210, 216

XML Extensible Markup Language. 154, 155

YAWN Yet Another Wearable Toolkit. 105

References

[1] 1-wire - wikipedia. https://en.wikipedia.org/wiki/1-Wire. (Accessed on 04/12/2020).

[2] Aws iot applications & solutions. https://aws.amazon.com/iot/. (Accessed on 10/29/2019).

[3] Adafruit pygamer for makecode arcade, circuitpython or arduino id: 4242 - $39.95 : Adafruit industries, unique & fun diy electronics and kits. https://www.adafruit.com/product/4242. (Accessed on 04/02/2020).

[4] Amazon.com: All-new echo (3rd gen) - smart speaker with alexa - twilight blue: Amazon devices. https://www.amazon.com/dp/B07R1CXKN7?ref=ods_surl_pl. (Accessed on 10/21/2019).

[5] An all-in-one iot kit built for the cloud. https://microsoft.github.io/azure-iot-developer-kit/. (Accessed on 04/14/2020).

[6] Arduboy. https://arduboy.com/, . (Accessed on 04/02/2020).

[7] Arduino - homepage. https://www.arduino.cc/en/IoT/HomePage, . (Accessed on 10/29/2019).

[8] Arm pelion | arm pelion iot platform. https://www.pelion.com/. (Accessed on 10/29/2019).

[9] Avant-garde fashion: A modern definition - barbara i gongini. https://barbaraigongini.com/universe/blog/avant-garde-fashion-a-modern-definition/. (Accessed on 11/08/2019).

[10] Brainpad – a mini coding computer. https://www.brainpad.com/. (Accessed on 04/14/2020).

[11] Circuitpython. https://circuitpython.org/. (Accessed on 10/10/2019).

[12] Crowdfund innovations & support entrepreneurs | indiegogo. https://www.indiegogo.com/. (Accessed on 13/05/2020).

[13] Cubetto: A toy robot teaching kids code & computer programming. https://www.primotoys.com/. (Accessed on 09/30/2019).

[14] Duktape. https://duktape.org/. (Accessed on 10/15/2019).

[15] Edge connector and pinout. https://tech.microbit.org/hardware/edgeconnector/. (Accessed on 06/25/2019).

[16] Engage every learner in steam with lego® education spike™ prime! https://education.lego.com/en-gb/meetspikeprime. (Accessed on 04/10/2020).

[17] Eric migicovsky on pebble's origin, smartwatch philosophy and what's wrong with the competition | engadget. https://www.engadget.com/2013/09/11/eric-migicovsky-pebble-ceo/?guce_referrer=aHR0cHM6Ly9kdWNrZHVja2dvLmNvbS8&guce_referrer_sig=AQAAAMHeo_nHMt84xH-KkMFzlqI3fYKflGDSaWxsD1_Ik-AYqxQkBmlzCKdY_lSnFwTCgqDLauK-aiC1caCd-gwxT9UT7fecrfQWiTg3ocqC60eP56AHF5LBZiwkXkVWkBLUHU_zKNBKIEgZY_OgVMAVMacUxM53VQa3f3DQ0NNIhv_d&_guc_consent_skip=1573318361. (Accessed on 11/09/2019).

[18] Feather specification | introducing adafruit feather | adafruit learning system. https://learn.adafruit.com/adafruit-feather/feather-specification. (Accessed on 06/25/2019).

[19] Google home – connected home assistant – google store. https://store.google.com/product/google_home. (Accessed on 10/21/2019).

[20] Grove system - seeed wiki. http://wiki.seeedstudio.com/Grove_System/. (Accessed on 06/25/2019).

[21] Ifttt: Every thing works better together. https://ifttt.com/. (Accessed on 03/25/2020).

[22] Introducing raspberry pi hats - raspberry pi. https://www.raspberrypi.org/blog/introducing-raspberry-pi-hats/. (Accessed on 06/25/2019).

[23] Kickstarter. https://www.kickstarter.com/. (Accessed on 13/05/2020).

[24] Kittenbot. https://www.kittenbot.cc/. (Accessed on 04/14/2020).

[25] Mindstorms ev3 – products – lego education. https://education.lego.com/en-gb/product/mindstorms-ev3. (Accessed on 04/10/2020).

[26] Mipi i3c host controller interface. https://mipi.org/specifications/i3c-hci. (Accessed on 04/12/2020).

[27] Maker faire | skoolie: Build your own rv from a school bus. https://makerfaire.com/maker/entry/69395/. (Accessed on 13/05/2020).

[28] Modular robotics | cubelets robot blocks. https://www.modrobotics.com/. (Accessed on 11/10/2019).

[29] Non-domestic smart energy management innovation competition - gov.uk. https://www.gov.uk/government/publications/non-domestic-smart-energy-management-innovation-competition. (Accessed on 03/05/2020).

[30] Pc/104 - wikipedia. https://en.wikipedia.org/wiki/PC/104. (Accessed on 05/29/2020).

[31] Pico-8 fantasy console. https://www.lexaloffle.com/pico-8.php. (Accessed on 04/02/2020).

[32] Particle - welcome to real iot. https://www.particle.io/, . (Accessed on 11/04/2019).

[33] Particle mesh update — a note from the ceo – particle blog. https://blog.particle.io/mesh-deprecation/, . (Accessed on 03/27/2020).

[34] Pebble time - awesome smartwatch, no compromises by pebble technology — kickstarter. https://www.kickstarter.com/projects/getpebble/pebble-time-awesome-smartwatch-no-compromises/description. (Accessed on 11/09/2019).

[35] Plant monitoring system - arduino project hub. https://create.arduino.cc/projecthub/ryanjgill2/plant-monitoring-system-88ed2b, . (Accessed on 14/05/2020).

[36] Playdate. https://play.date/, . (Accessed on 04/02/2020).

[37] Rust vs c gcc - which programs are fastest? | computer language benchmarks game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html. (Accessed on 10/10/2019).

[38] Smartthings. add a little smartness to your things. https://www.smartthings.com/gb/smart-home. (Accessed on 10/29/2019).

[39] Starbucks turns to technology to brew up a more personal connection with its customers | transform. https://news.microsoft.com/transform/starbucks-turns-to-technology-to-brew-up-a-more-personal-connection-with-its-customers/. (Accessed on 11/04/2019).

[40] Survey shows linux and freertos out front in embedded tech. http://linuxgizmos.com/survey-shows-linux-and-freertos-out-front-in-embedded-tech/. (Accessed on 17/05/2020).

[41] Tic-80 tiny computer. https://tic.computer/. (Accessed on 04/02/2020).

[42] The component object model - windows applications | microsoft docs. https://docs.microsoft.com/en-us/windows/win32/com/the-component-object-model. (Accessed on 10/10/2019).

[43] Wedo 2.0 – products – lego education. https://education.lego.com/en-gb/product/wedo-2. (Accessed on 04/10/2020).

[44] Webusb api. https://wicg.github.io/webusb/. (Accessed on 08/14/2019).

[45] Welcome to microsoft flow | power automate blog. https://flow.microsoft.com/en-us/blog/welcome-to-microsoft-flow/. (Accessed on 03/26/2020).

[46] Adafruit feather nrf52 bluefruit le [nrf52832] id: 3406 - $24.95 : Adafruit industries, unique & fun diy electronics and kits. https://www.adafruit.com/product/3406. (Accessed on 05/29/2020).

[47] Interface and application programming | week 14. http://archive.fabacademy.org/2018/labs/fablabcept/students/dhruv-saidava/14-interface-and-application-programming.html, . (Accessed on 05/29/2020).

[48] Shortcuts: A new vision for siri and ios automation - macstories. https://www.macstories.net/stories/shortcuts-a-new-vision-for-siri-and-ios-automation/, . (Accessed on 05/29/2020).

[49] Arduino nano 33 ble sense with headers | arduino official store. https://store.arduino.cc/arduino-nano-33-ble-sense-with-headers, . (Accessed on 05/29/2020).

[50] Arduino board - plugnmake news. http://plugnmake.com/news/blog/2015/07/06/makers-you-need-to-know-massimo-banzi/arduino/, . (Accessed on 05/29/2020).

[51] What are breadboards and their uses | breadboard | maker pro. https://maker.pro/breadboard/tutorial/an-introduction-to-breadboards-and-their-uses, . (Accessed on 05/29/2020).

[52] Arduino code | adafruit mma8451 accelerometer breakout | adafruit learning system. https://learn.adafruit.com/adafruit-mma8451-accelerometer-breakout/wiring-and-test, . (Accessed on 05/29/2020).

[53] First hand on the arduino uno board - tutorial45. https://tutorial45.com/first-hand-on-the-arduino-uno-board/, . (Accessed on 05/29/2020).

[54] Azure sphere get started | microsoft azure. https://azure.microsoft.com/en-us/services/azure-sphere/get-started/. (Accessed on 05/29/2020).

[55] Overview | cheerlights | adafruit learning system. https://learn.adafruit.com/cheerlights/overview. (Accessed on 14/05/2020).

[56] Overview | adafruit circuit playground express | adafruit learning system. https://learn.adafruit.com/adafruit-circuit-playground-express. (Accessed on 05/29/2020).

[57] Primo toys cubetto coding playset | officeworks. https://www.officeworks.com.au/shop/officeworks/p/primo-toys-cubetto-coding-playset-primo001b. (Accessed on 05/29/2020).

[58] Rk education rkub1 bbc microbit breakout adapter board micro bit self solder: Amazon.co.uk: Computers & accessories. https://www.amazon.co.uk/Rk-Education-MicroBit-Breakout-Adapter/dp/B07NP8Q7VC/ref=pd_sbs_147_5/275-6479219-3868602?_encoding=UTF8&pd_rd_i=B07NP8Q7VC&pd_rd_r=5963376f-1176-428c-9884-c1091b3f8e23&pd_rd_w=pAGEW&pd_rd_wg=yZjtf&pf_rd_p=2773aa8e-42c5-4dbe-bda8-5cdf226aa078&pf_rd_r=PRPP7PKF03RBWN2MBBYQ&psc=1&refRID=PRPP7PKF03RBWN2MBBYQ. (Accessed on 05/29/2020).

[59] Esp32 overview | espressif systems. https://www.espressif.com/en/products/socs/esp32/overview. (Accessed on 12/05/2020).

[60] Overview | adafruit feather m0 wifi with atwinc1500 | adafruit learning system. https://learn.adafruit.com/adafruit-feather-m0-wifi-atwinc1500. (Accessed on 05/29/2020).

[61] The world's leading software development platform · github. https://github.com/. (Accessed on 25/05/2020).

[62] Grove cape for beaglebone – unmanned tech uk fpv shop. https://www.unmannedtechshop.co.uk/product/grove-cape-for-beaglebone/. (Accessed on 05/29/2020).

[63] https://mipi.org/specifications/unipro-specifications. https://mipi.org/specifications/unipro-specifications. (Accessed on 04/12/2020).

[64] ios - home - apple (uk). https://www.apple.com/uk/ios/home/. (Accessed on 10/21/2019).

[65] With the official ifttt app, now available for iphone, you can add your apps to any recipe. https://genuine-lamps.com/ios/2743-official-ifttt-app-now-available-for-iphone-lets-you-add-your-apps-to-any-recipe.html. (Accessed on 05/29/2020).

[66] Prime day 2019: The best deals on smart lights from lutron, lifx, philips hue and more. https://www.msn.com/en-us/news/technology/prime-day-2019-the-best-deals-on-smart-lights-from-lutron-lifx-philips-hue-and-more/ar-AAEmHkl. (Accessed on 05/29/2020).

[67] The lego mindstorms graphical programming environment, used to create... | download scientific diagram. https://www.researchgate.net/figure/The-Lego-Mindstorms-graphical-programming-environment-used-to-create-simple-programs-for_fig2_225480818, . (Accessed on 05/29/2020).

[68] Lego mindstorms tank rover with ultrasonic sensors | cold solder. https://coldsolder.wordpress.com/2010/12/04/lego-mindstorms-tank-rover-with-ultrasonic-sensors/, . (Accessed on 05/29/2020).

[69] Lightblue bean australia. https://core-electronics.com.au/lightblue-bean.html. (Accessed on 05/29/2020).

[70] littlebits electronics base kit, science - amazon canada. https://www.amazon.ca/littleBits-Electronics-650-0119-Base-Kit/dp/B00ECWSL0I. (Accessed on 05/29/2020).

[71] Makey makey – makey shop. https://makeymakey.com/. (Accessed on 05/29/2020).

[72] microgame 10 is a tiny, python-programmable game console - hackster.io. https://www.hackster.io/news/game-10-is-a-tiny-python-programmable-game-console-4ea5a9288c5a. (Accessed on 04/02/2020).

[73] Robotické autíčko micro:maqueen – rozšíření pro váš micro:bit. https://www.microbiti.cz/2019/08/roboticke-auticko-micromaqueen.html. (Accessed on 05/29/2020).

[74] node-red/readme.md at master · node-red/node-red · github. https://github.com/node-red/node-red/blob/master/README.md. (Accessed on 05/29/2020).

[75] Xenon – particle retail. https://store.particle.io/products/xenon. (Accessed on 12/02/2020).

[76] Innovative applications of internet-of-things (iot). https://medium.com/@StudIoTeHub/innovative-applications-of-internet-of-things-iot-de1a6ec2c972. (Accessed on 05/29/2020).

[77] The official site of philips hue | meethue.com. https://www2.meethue.com/en-gb. (Accessed on 14/05/2020).

[78] Raspberry pi zero w - raspberry pi. https://www.raspberrypi.org/pi-zero-w/. (Accessed on 12/05/2020).

[79] Plant monitoring system - arduino project hub. https://create.arduino.cc/projecthub/ryanjgill2/plant-monitoring-system-88ed2b. (Accessed on 05/29/2020).

[80] Learn how unlimited tomorrow is modernizing prosthetics with the iot – particle blog. https://blog.particle.io/unlimited-tomorrow-prosthetic-hand/. (Accessed on 05/29/2020).

[81] Raspberry pi zero - accessories collection | the pi hut. https://thepihut.com/collections/raspberry-pi-zero. (Accessed on 05/29/2020).

[82] Raspberry pi sense hat | physicsopenlab. http://physicsopenlab.org/2020/01/21/raspberry-pi-sense-hat/. (Accessed on 05/29/2020).

[83] Jump into: Scratch – just for girls (ages 8-10) – museum of applied arts and sciences. https://maas.museum/event/jump-into-scratch-just-for-girls-ages-8-10/. (Accessed on 05/29/2020).

[84] Sphero mini programmable app-enabled robot ball - blue. https://www.mobilefun.co.uk/sphero-mini-programmable-app-enabled-robot-ball-blue-80821. (Accessed on 05/29/2020).

[85] Mozilla project things workshop - youtube. https://www.youtube.com/watch?v=uEHL3ZYS790. (Accessed on 05/29/2020).

[86] Selecting and using rs-232 , rs-422 , and rs-485 serial data standards. 2000.

[87] Ardublock | a graphical programming language for arduino. http://blog.ardublock.com/, 2012. (Accessed on 02/22/2018).

[88] Apache mynewt. https://mynewt.apache.org/, 2015. (Accessed on 11/16/2017).

[89] Home - zephyr project. https://www.zephyrproject.org/, 2017. (Accessed on 11/16/2017).

[90] Sam Aaron. Sonic Pi – performance in education, technology and art. *International Journal of Performance Arts and Digital Media*, 12(2):171–178, 2016. ISSN 1479-4713. doi: 10.1080/14794713.2016.1227593. URL https://www.tandfonline.com/doi/full/10.1080/14794713.2016.1227593.

[91] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[92] Ferran Adelantado, Xavier Vilajosana, Pere Tuset-Peiro, Borja Martinez, Joan Melia-Segui, and Thomas Watteyne. Understanding the limits of lorawan. *IEEE Communications magazine*, 55(9):34–40, 2017.

[93] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. Concurrent transmissions for multi-hop bluetooth 5. In *EWSN*, pages 130–141, 2019.

[94] MIPI Alliance. Mipi alliance launches new m-phy and unipro specifications for mobile device applications. *Jun*, 10:1, 2011.

[95] Apple. Apple reports record first quarter results - apple. https://www.apple.com/newsroom/2020/01/apple-reports-record-first-quarter-results/, January 2020. (Accessed on 04/02/2020).

[96] ARM. Gnu toolchain | gnu-rm downloads – arm developer. https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads. (Accessed on 01/01/2020).

[97] ARM. The arm mbed iot device platform. 2017. URL https://www.mbed.com/.

[98] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[99] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[100] Faisal Aslam, Ghufran Baig, Mubashir Adnan Qureshi, Zartash Afzal Uzmi, Luminous Fennell, Peter Thiemann, Christian Schindelhauer, and Elmar Haussmann. Rethinking java call stack design for tiny embedded devices. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 1–10, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1212-7. doi: 10.1145/2248418.2248420. URL http://doi.acm.org/10.1145/2248418.2248420.

[101] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli de Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale. The BBC micro:bit—from the UK to the World. *Communications of the ACM*, 2020.

[102] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli De Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale. The BBC micro:bit—from the UK to the world. *Communications of the ACM*, 63(3):62–69, 2020.

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

[104] Emmanuel Baccelli, Oliver Hahm, and M Günes. RIOT OS: Towards an OS for the Internet of Things. *Proc. of the 32nd IEEE ...*, pages 2453–2454, 2013. doi: 10.1109/INFCOMW.2013.6970748. URL http://www.riot-os.org/docs/riot-infocom2013-abstract.pdf.

[105] Carmen Badea, Alexandru Nicolau, and Alexander V. Veidenbaum. A simplified java bytecode compilation system for resource-constrained embedded processors. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 218–228, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-826-8. doi: 10.1145/1289881.1289920. URL http://doi.acm.org/10.1145/1289881.1289920.

[106] Mathias Baert, Jen Rossey, Adnan Shahid, and Jeroen Hoebeke. The bluetooth mesh standard: An overview and experimental evaluation. *Sensors*, 18(8):2409, 2018.

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

[108] Hernando Barragán. Wiring: Prototyping physical interaction design. *Interaction Design Institute, Ivrea, Italy*, 2004.

[109] Richard Barry et al. Freertos. *Internet, Oct*, 2008.

[110] Oliver Bates and Adrian Friday. Beyond data in the smart city: repurposing existing campus iot. *IEEE Pervasive Computing*, 16(2):54–60, 2017.

[111] Ayah Bdeir and Ted Ullrich. Electronics as material: littleBits. *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction - TEI '11*, page 341, 2011. doi: 10.1145/1935701.1935781. URL http://portal.acm.org/citation.cfm?doid=1935701.1935781.

[112] Andrew Beatty, Kevin Casey, David Gregg, and Andrew Nisbet. An optimised java interpreter for connected devices and embedded systems. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 692–697, New York, NY, USA, 2003. ACM. ISBN 1-58113-624-2. doi: 10.1145/952532.952667. URL http://doi.acm.org/10.1145/952532.952667.

[113] Chatschik Bisdikian et al. An overview of the bluetooth wireless technology. *IEEE Commun Mag*, 39(12):86–94, 2001.

[114] Paulo Blikstein. Gears of our childhood: constructionist toolkits, robotics, and physical computing, past and future. In *Proceedings of the 12th international conference on interaction design and children*, pages 173–182. ACM, 2013.

[115] Tracey Booth and Simone Stumpf. End-user experiences of visual and textual programming environments for arduino. In *International symposium on end user development*, pages 25–39. Springer, 2013.

[116] Tracey Booth, Simone Stumpf, Jon Bird, and Sara Jones. Crossed wires: Investigating the problems of end-user developers in a physical computing task. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3485–3497. ACM, 2016.

[117] Martin Bor, John Edward Vidler, and Utz Roedig. Lora for the internet of things. 2016.

[118] Carsten Bormann, Angelo P. Castellani, and Zach Shelby. CoAP: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012. ISSN 10897801. doi: 10.1109/MIC.2012.29.

[119] Martina Brachmann, Olaf Landsiedel, Diana Göhringer, and Silvia Santini. Whisper: Fast flooding for low-power wireless networks. *ACM Transactions on Sensor Networks (TOSN)*, 15(4):1–26, 2019.

[120] Neil Briscoe. Understanding the osi 7-layer model. *PC Network Advisor*, 120(2), 2000.

[121] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[122] Leah Buechley, Nwanua Elumeze, Camille Dodson, and Michael Eisenberg. Quilt snaps: A fabric based computational construction kit. In *IEEE International Workshop on Wireless and Mobile Technologies in Education (WMTE'05)*, pages 3–pp. IEEE, 2005.

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

[124] Christian Butterworth, Daniel Kershaw, James Devine, James Gallagher, and Jack Croft. Presenting the past: a case study of innovation opportunities for knowledge dissemination to the general public through pervasive technology. In *21st Annual Meeting of the European Association of Archaeologists*, pages 391–392, 2015.

[125] Lautaro Cabrera, John H. Maloney, and David Weintrop. Programs in the palm of your hand: How live programming shapes children's interactions with physical computing devices. *Proceedings of the 18th ACM International Conference on Interaction Design and Children, IDC 2019*, (June):227–236, 2019. doi: 10.1145/3311927.3323138.

[126] Martin C Carlisle, Terry A Wilson, Jeffrey W Humphries, and Steven M Hadfield. Raptor: a visual programming environment for teaching algorithmic problem solving. *Acm Sigcse Bulletin*, 37(1):176–180, 2005.

[127] James Caska and Martin Schoeberl. Java dust: How small can embedded java be? In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 125–129, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0731-4. doi: 10.1145/2043910.2043931. URL http://doi.acm.org/10.1145/2043910.2043931.

[128] Arduino CC. Arduino create. https://create.arduino.cc/. (Accessed on 01/04/2018).

[129] Ian D Chakeres and Elizabeth M Belding-Royer. Aodv routing protocol implementation design. In *24th International Conference on Distributed Computing Systems Workshops, 2004. Proceedings.*, pages 698–703. IEEE, 2004.

[130] Tej Bahadur Chandra, Pushpak Verma, and AK Dwivedi. Operating systems for internet of things: A comparative study. In *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, page 47. ACM, 2016.

[131] François Chollet et al. keras, 2015.

[132] Michael Clarke, Gordon S Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 160–178. Springer, 2001.

[133] CodeBug. Codebug – home. https://www.codebug.org.uk/. (Accessed on 01/03/2018).

[134] Beginner's Mind Collective and David Shaw. Makey makey: improvising tangible and nature-based user interfaces. In *Proceedings of the sixth international conference on tangible, embedded and embodied interaction*, pages 367–370, 2012.

[135] Software Freedom Conservancy. Git. https://git-scm.com/. (Accessed on 01/01/2020).

[136] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: a 3-d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5):107–116, 2000.

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

[138] Eva Dakich. Teachers' perceptions about the barriers and catalysts for effective practices with ict in primary schools. In *IFIP World Conference on Computers in Education*, pages 445–453. Springer, 2009.

[139] Artem Dementyev, Tomás Vega Gálvez, and Alex Olwal. Sensorsnaps: Integrating wireless sensor nodes into fabric snap fasteners for textile interfaces. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 17–28. ACM, 2019.

[140] Media & Sport Department for Digital, Culture. No longer optional: Employer demand for digital skills - no_longer_optional_employer_demand_for_digital_skills.pdf. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/807830/No_Longer_Optional_Employer_Demand_for_Digital_Skills.pdf. (Accessed on 10/24/2019).

[141] Giuseppe Desoli and Enrica Filippi. An outlook on the evolution of mobile terminals: from monolithic to modular multiradio, multiapplication platforms. *IEEE Circuits and Systems Magazine*, 6(2):17–29, 2006.

[142] James Devine, Joe Finney, Michał Moskal, Peli de Halleux, Thomas Ball, and Steve Hodges. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. 2018.

[143] James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. *Journal of Systems Architecture*, 2019.

[144] Manjunath Doddavenkatappa, Mun Choon Chan, and Ben Leong. Splash: Fast data dissemination with constructive interference in wireless sensor networks. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 269–282, 2013.

[145] Dale Dougherty. The maker movement. *Innovations: Technology, Governance, Globalization*, 7(3):11–14, 2012.

[146] Vadim Drabkin, Roy Friedman, Gabriel Kliot, and Marc Segal. Rapid: Reliable probabilistic dissemination in wireless ad-hoc networks. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 13–22. IEEE, 2007.

[147] Adam Dunkels. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors.

[148] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with glossy. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 73–84. IEEE, 2011.

[149] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power wireless bus. *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems - SenSys '12*, page 1, 2012. ISSN 9781450311694. doi: 10.1145/2426656.2426658. URL http://dl.acm.org/citation.cfm?doid=2426656.2426658.

[150] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85, 2000.

[151] Daniel K Fisher and Peter J Gould. Open-source hardware is a low-cost alternative for scientific instrumentation and research. *Modern instrumentation*, 1(02):8, 2012.

[152] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[153] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language: Everything You Need to Know*. " O'Reilly Media, Inc.", 2008.

[154] Rostislav Fojtika. The ozobot and education of programming. 2017.

[155] Warwick Institute for Employment Research. What_digital_skills_do_adults_need_to_succeed_in_the_workplace_now_and_in_the_next_10_years https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/807831/What_digital_skills_do_adults_need_to_succeed_in_the_workplace_now_and_in_the_next_10_years_.pdf. (Accessed on 10/24/2019).

[156] N Fraser et al. Blockly: A visual programming editor. *Published. Google, Place*, 2013.

[157] Neil Fraser. Ten things we've learned from Blockly. *Proceedings - 2015 IEEE Blocks and Beyond Workshop, Blocks and Beyond 2015*, pages 49–50, 2015. doi: 10.1109/BLOCKS.2015.7369000.

[158] Despo Galataki. Design & verification of unipro protocols for mobile phones. *Vrije Universiteit (VU), Amsterdam, The Netherlands*, 2009.

[159] Mikhail T Galeev. Catching the z-wave. *Embedded Systems Design*, 19(10):28, 2006.

[160] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 19–20. ACM, 2015.

[161] Damien George. Micropython-python for microcontrollers, 2014.

[162] Adele Goldberg. Smalltalk-80-the interactive programming environment. 1984.

[163] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9): 11734–11753, 2012.

[164] UK Government. National curriculum in england: computing programmes of study - gov.uk. https://www.gov.uk/government/publications/national-curriculum-in-england -computing-programmes-of-study/national-curriculum-in-england-computing-progr ammes-of-study. (Accessed on 10/24/2019).

[165] David N Gray, John Hotchkiss, Seth LaForge, Andrew Shalit, and Toby Weinberg. Modern languages and microsoft's component object model: Programming com made simple. *Communications of the ACM*, 41(5):55–66, 1998.

[166] Saul Greenberg and Chester Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218. ACM, 2001.

[167] Chris Griffith. *Mobile App Development with Ionic, Revised Edition: Cross-Platform Apps with Ionic, Angular, and Cordova*. " O'Reilly Media, Inc.", 2017.

[168] John Gruber, Aaron Swartz, et al. Markdown, 2004.

[169] Zonglin Guo, Ian G Harris, Lih-feng Tsaur, and Xianbo Chen. An on-demand scatternet formation and multi-hop routing protocol for ble-based wireless sensor networks. In *2015 IEEE wireless communications and networking conference (WCNC)*, pages 1590–1595. IEEE, 2015.

[170] Zygmunt J Haas, Joseph Y Halpern, and Li Li. Gossip-based ad hoc routing. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1707–1716. IEEE, 2002.

[171] Stephan Hankammer, Ruth Jiang, Robin Kleer, and Martin Schymanietz. From phonebloks to google project ara. a case study of the application of sustainable mass customization. *Procedia CIRP*, 51:72–78, 2016.

[172] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# programming language*. Adobe Press, 2006.

[173] R Hinden and S Deering. Rfc 2373: Ip version 6 addressing architecture. *IETF, July*, 1998.

[174] Steve Hodges and Nicholas Chen. Long tail hardware: Turning device concepts into viable low volume products. *IEEE Computer Architecture Letters*, 18(04):51–59, 2019.

[175] Steve Hodges, Stuart Taylor, Nicolas Villar, James Scott, Dominik Bial, and Patrick Tobias Fischer. Prototyping connected devices for the internet of things. *Computer*, 46 (2):26–34, 2013. ISSN 00189162. doi: 10.1109/MC.2012.394.

[176] Steve Hodges, Nicolas Villar, Nicholas Chen, Tushar Chugh, Jie Qi, Diana Nowacka, and Yoshihiro Kawahara. Circuit stickers: peel-and-stick construction of interactive electronic prototypes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1743–1746. ACM, 2014.

[177] Steve Hodges, Sue Sentance, Joe Finney, and Thomas Ball. Physical computing: A key element of modern computer science education. *IEEE Computer*, 2019.

[178] IBM. Ibm watson internet of things (iot) - united kingdom | ibm. https://www.ibm.com/uk-en/internet-of-things. (Accessed on 10/29/2019).

[179] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6): 635–652, 1996.

[180] Beate Jost, Markus Ketterl, Reinhard Budde, and Thorsten Leimbach. Graphical programming environments for educational robots: Open roberta-yet another one? In *2014 IEEE International Symposium on Multimedia*, pages 381–386. IEEE, 2014.

[181] Yoshiharu Kato. Splish: A visual programming environment for arduino to accelerate physical computing experiences. *8th International Conference on Creating, Connecting and Collaborating through Computing, C5 2010*, pages 3–10, 2010. doi: 10.1109/C5.2010.20.

[182] Eva-Sophie Katterfeldt, Nadine Dittert, and Heidi Schelhowe. Eduwear: smart textiles as ways of relating computing technology to everyday life. In *Proceedings of the 8th International Conference on Interaction Design and Children*, pages 9–17. ACM, 2009.

[183] KM Kensek. Integration of environmental sensors with bim: case studies using arduino, dynamo, and the revit api. 2014.

[184] Sabri Khssibi, Hanen Idoudi, Adrien Van den Bossche, Thierry Val, and Leila Azzouz Saidane. Presentation and analysis of a new technology for low-power wireless sensor network. 2013.

[185] Hyun-Soo Kim, JungYub Lee, and Ju Wook Jang. Blemesh: A wireless mesh network protocol for bluetooth low energy devices. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 558–563. IEEE, 2015.

[186] Seong Hoon Kim, Poh Kit Chong, and Daeyoung Kim. A location-free semi-directional-flooding technique for on-demand routing in low-rate wireless mesh networks. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3066–3075, 2014.

[187] Patrick Kinney et al. Zigbee technology: Wireless control that simply works. In *Communications design conference*, volume 2, pages 1–7, 2003.

[188] Kitware. Cmake. https://cmake.org/. (Accessed on 01/01/2020).

[189] Frank Klassner and Scott D Anderson. Lego mindstorms: Not just for k-12 anymore. *IEEE Robotics & Automation Magazine*, 10(2):12–18, 2003.

[190] Bran Knowles, Joe Finney, Sophie Beck, and James Devine. What children's imagined uses of the BBC micro:bit tells us about designing for their iot privacy, security and safety. *In Living in the Internet of Things: Cybersecurity of the IoT*, 2018.

[191] Bran Knowles, Sophie Beck, Joe Finney, James Devine, and Joseph Lindley. A scenario-based methodology for exploring risks: Children and programmable IoT. In *Proceedings of the 2019 on Designing Interactive Systems Conference*, pages 751–761. ACM, 2019.

[192] Bran Knowles Knowles, Sophie Beck, Georgia Newmarch, Joe Finney, and James Devine. IoT4Kids: Strategies for mitigating against risks of IoT for children. *In proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[193] Alfred Kobsa, Rahim Sonawalla, Gene Tsudik, Ersin Uzun, and Yang Wang. Serial hook-ups: a comparative usability study of secure device pairing methods. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, page 10. ACM, 2009.

[194] Stephan Korsholm. Flash memory in embedded java programs. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 116–124, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0731-4. doi: 10.1145/2043910.2043930. URL http://doi.acm.org/10.1145/2043910.2043930.

[195] Dong-Won Kum, Anh-Ngoc Le, You-Ze Cho, Chai Keong Toh, and In-Soo Lee. An efficient on-demand routing approach with directional flooding for wireless mesh networks. *Journal of Communications and Networks*, 12(1):67–73, 2010.

[196] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[197] MIPI Physical Layer, Peter Lefkin, and Rick Wietfeldt. Understanding mipi alliance interface specifications.

[198] J.C. Lee, D. Avrahami, S.E. Hudson, J. Forlizzi, P. Dietz, and D. Leigh. The calder toolkit: wired and wireless components for rapidly prototyping physical computing devices. *Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques*, 04:167–175, 2004. doi: http://doi.acm.org/10.1145/1013115.1013139.

[199] Frédéric Leens. An introduction to i2c and spi protocols. *IEEE Instrumentation & Measurement Magazine*, 12(1):8–13, 2009.

[200] Krijn Leentvaar and Jan Flint. The capture effect in fm receivers. *IEEE Transactions on Communications*, 24(5):531–539, 1976.

[201] Julio León, Abel Dueñas, Yuzo Iano, Cibele Abreu Makluf, and Guillermo Kemper. A bluetooth low energy mesh network auto-configuring proactive source routing protocol. In *2017 IEEE International Conference on Consumer Electronics (ICCE)*, pages 348–349. IEEE, 2017.

[202] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 54, 2014.

[203] Tim Leung. Beginning powerapps. *The Non-Developers Guide to Building Business Mobile Applications. Berkeley, CA: Apress*.

[204] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005. ISSN 01681699. doi: 10.1007/3-540-27139-2_7.

[205] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[206] Silvia Lindtner, Garnet D Hertz, and Paul Dourish. Emerging sites of hci innovation: hackerspaces, hardware startups & incubators. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 439–448. ACM, 2014.

[207] Joshua Lowe. Edublocks. https://edublocks.org/. (Accessed on 04/09/2020).

[208] Matthew B MacLaurin. The design of kodu: A tiny visual programming language for children on the xbox 360. In *ACM Sigplan Notices*, volume 46, pages 241–246. ACM, 2011.

[209] Anusha Mahale and BS Kariyappa. Architecture analysis and verification of i3c protocol. In *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, pages 930–935. IEEE, 2019.

[210] Bishnu Kumar Maharjan, Ulf Witkowski, and Reza Zandian. Tree network based on bluetooth 4.0 for wireless sensor network applications. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pages 172–176. IEEE, 2014.

[211] John Maloney, Kylie Peppler, Yasmin B. Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: urban youth learning programming with scratch. *SIGCSE '08 Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 367–371, 2008. ISSN 0097-8418. doi: 10.1145/1352135.1352260. URL http://dl.acm.org/citation.cfm?id=1352260.

[212] John Maloney, Mitchel Resnick, and Natalie Rusk. The Scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):1–15, 2010. ISSN 1946-6226. doi: 10.1145/1868358.1868363.http.

URL http://dl.acm.org/citation.cfm?id=1868363{%}5Cnhttps://blogs.kent.ac.uk/mik/2010/12/scratch-alice-greenfoot/.

[213] John H Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 367–371, 2008.

[214] Dave Mason and Kruti Dave. Block-based versus flow-based programming for naive programmers. In *2017 IEEE Blocks and Beyond Workshop (B&B)*, pages 25–28. IEEE, 2017.

[215] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[216] Timothy S McNerney. From turtles to tangible programming bricks: explorations in physical language design. *Personal and Ubiquitous Computing*, 8(5):326–337, 2004.

[217] Erik Meijer and John Gough. Technical overview of the common language runtime. *language*, 29:7, 2001.

[218] Robert M Metcalfe and David R Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[219] Microsoft. Azure iot | microsoft azure. https://azure.microsoft.com/en-us/overview/iot/. (Accessed on 10/29/2019).

[220] Amon Millner and Edward Baafi. Modkit: blending and extending approachable platforms for creating computer programs and interactive objects. *Proceedings of the 10th International Conference on Interaction Design and Children*, pages 250–253, 2011. doi: 10.1145/1999030.1999074. URL http://dl.acm.org/citation.cfm?id=1999074.

[221] Cecily Morrison, Nicolas Villar, Anja Thieme, Zahra Ashktorab, Eloise Taysom, Oscar Salandin, Daniel Cletheroe, Greg Saul, Alan F Blackwell, Darren Edge, et al. Torino: A tangible programming language inclusive of children with visual disabilities. *Human–Computer Interaction*, pages 1–49, 2018.

[222] J Paul Morrison. Flow-based programming. In *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 25–29, 1994.

[223] Mozilla. Mozilla iot. https://iot.mozilla.org/. (Accessed on 10/29/2019).

[224] Geoff Mulligan. The 6LoWPAN architecture. *Proceedings of the 4th workshop on Embedded networked sensors - EmNets '07*, page 78, 2007. doi: 10.1145/1278972.1278992. URL http://portal.acm.org/citation.cfm?doid=1278972.1278992.

[225] Beshr Al Nahas, Antonio Escobar-Molero, Jirka Klaue, Simon Duquennoy, and Olaf Landsiedel. Blueflood: Concurrent transmissions for multi-hop bluetooth 5–modeling and evaluation. *arXiv preprint arXiv:2002.12906*, 2020.

[226] Rick Nelson. Mipi alliance debuts i3c, disco specifications. *EE-Evaluation Engineering*, 56(3):2–3, 2017.

[227] Kathy New, James Devine, Taylor Woodcock, Sophie Beck, Joe Finney, Mike Hazas, Nick Banks, Karen Smith, and Tim Bailey. Energy in Schools: Promoting global change through social technical deployments. *In Living in the Internet of Things: Harnessing Economic Value*, 2019.

[228] Grace Ngai, Stephen CF Chan, Vincent TY Ng, Joey CY Cheung, Sam SS Choy, Winnie WY Lau, and Jason TP Tse. 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*, pages 443–452. ACM, 2010.

[229] Ofcom. Data analysis - connected nations 2017: The internet of things - connected-nations-internet-things-2017.pdf. chrome-extension: //oemmndcbldboiebfnladdacbdfmadadm/https://www.ofcom.org.uk/__data/assets/pdf_file/0020/108515/connected-nations-internet-things-2017.pdf. (Accessed on 11/06/2019).

[230] Nick O'Leary and Dave Conway-Jones. Node-red. https://nodered.org/. (Accessed on 03/27/2020).

[231] OzoBlockly. Welcome to ozoblockly. http://ozoblockly.com/. (Accessed on 01/03/2018).

[232] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.

[233] Seymour Papert. Situating Constructionism. *Constructionism*, pages 1–11, 1991. doi: 10.1111/1467-9752.00269. URL http://namodemello.com.br/pdf/tendencias/situatingconstrutivism.pdf{%}5Cnhttp://psycnet.apa.org/psycinfo/1991-99006-000.

[234] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[235] Gaetano Patti, Luca Leonardi, and Lucia Lo Bello. A bluetooth low energy real-time protocol for industrial wireless mesh networks. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*, pages 4627–4632. IEEE, 2016.

[236] Joshua M Pearce. Building research equipment with free, open-source hardware. *Science*, 337(6100):1303–1304, 2012.

[237] Simon Peyton Jones. Computer science as a school subject. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 159–160. ACM, 2013. doi: 10.1145/2500365.2500609. URL http://doi.acm.org/10.1145/2500365.2500609.

[238] Stefan Pleisch, Mahesh Balakrishnan, Ken Birman, and Robbert Van Renesse. Mistral: efficient flooding in mobile ad-hoc networks. In *Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*, pages 1–12, 2006.

[239] Jon Postel et al. User datagram protocol. 1980.

[240] Ivan Poupyrev, Nan-Wei Gong, Shiho Fukuhara, Mustafa Emre Karagozler, Carsten Schwesig, and Karen E Robinson. Project jacquard: interactive digital textiles at scale. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 4216–4227. ACM, 2016.

[241] Ramjee Prasad. *CDMA for wireless personal communications*. Artech House, Inc., 1996.

[242] Yaswanth Kumar Reddy, Praneeth Juturu, Hari Prabhat Gupta, Pramod Reddy Serikar, Shruti Sirur, Sulekha Barak, and Bonggon Kim. A connection oriented mesh network for mobile devices using bluetooth low energy. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 453–454, 2015.

[243] Microsoft Research. Fashion forward: Researchers, designers debut new tech on new york city runway - microsoft research. https://www.microsoft.com/en-us/research/blog/fashion-forward-researchers-designers-debut-new-tech-on-new-york-city-runway/, May 2019. (Accessed on 08/15/2019).

[244] Mitchel Resnick and Brian Silverman. Some reflections on designing construction kits for kids. *Proceeding of the 2005 conference on Interaction design and children - IDC '05*, pages 117–122, 2005. ISSN 1595930965. doi: 10.1145/1109540.1109556. URL http://portal.acm.org/citation.cfm?doid=1109540.1109556.

[245] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, J a Y Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Communications of the ACM*, 52:60–67, 2009. ISSN 00010782. doi: 10.1145/1592761.1592779. URL http://search.ebscohost.com/login.aspx?direct=true{&}db=bth{&}AN=45021156{&}site=eds-live{%}5Cnfiles/130/RESNICKetal.-2009-ScratchProgrammingforAll..pdf.

[246] Open Roberta. Open roberta lab. https://lab.open-roberta.org//. (Accessed on 01/03/2018).

[247] Vex Robotics. V5 classroom starter kit - v5 robotics kits - products - vex v5 - vex robotics. https://www.vexrobotics.com/276-7110.html. (Accessed on 03/29/2020).

[248] Guido Rossum. Python tutorial. 1995.

[249] José Rufino. An overview of the controller area network. In *Proceedings of the CiA Forum CAN for Newcomers*, 1997.

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

[251] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6): 52–59, 1997.

[252] Jeffrey Schiller, Franklyn Turbak, Hal Abelson, José Dominguez, Andrew McKinney, Johanna Okerlund, and Mark Friedman. Live programming of mobile apps in app inventor. In *Proceedings of the 2nd Workshop on Programming for Mobile & Touch*, pages 1–8. ACM, 2014.

[253] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. Compiling java for low-end embedded systems. *SIGPLAN Not.*, 38(7): 42–50, June 2003. ISSN 0362-1340. doi: 10.1145/780731.780739. URL http://doi.acm.org/10.1145/780731.780739.

[254] Dallas Semiconductor. Fundamentals of rs-232 serial communications. *Dostopno na: http://www. pacontrol. com/download/RS232. pdf.[4. 3. 2016]*, 1998.

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

[256] Stefania Sesia, Issam Toufik, and Matthew Baker. *LTE-the UMTS long term evolution: from theory to practice*. John Wiley & Sons, 2011.

[257] Teddy Seyed and Anthony Tang. Mannequette: Understanding and enabling collaboration and creativity on avant-garde fashion-tech runways. In *Proceedings of the 2019 on Designing Interactive Systems Conference*, pages 317–329. ACM, 2019.

[258] Teddy Seyed, Peli de Halleux, Michal Moskal, James Devine, Joe Finney, Steve Hodges, and Thomas Ball. Makerarcade: Using gaming and physical computing for playful making, learning, and creativity. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, page LBW0174. ACM, 2019.

[259] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java™ on the bare metal. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–151, 2005.

[260] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM, 2006.

[261] Kiran Jot Singh and Divneet Singh Kapoor. Create your own internet of things: A survey of iot platforms. *IEEE Consumer Electronics Magazine*, 6(2):57–68, 2017.

[262] Shruthi Sirur, Praneeth Juturu, Hari Prabhat Gupta, Pramod Reddy Serikar, Yaswanth Kumar Reddy, Sulekha Barak, and Bonggon Kim. A mesh network for mobile devices using bluetooth low energy. In *2015 IEEE SENSORS*, pages 1–4. IEEE, 2015.

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

[264] Universal Serial Bus Specification. Revision 2.0, 2000.

[265] Sphero. Mini robot ball | programmable robot ball | sphero mini. https://www.sphero.com/sphero-mini. (Accessed on 03/29/2020).

[266] ISO Standard. 11898: Road vehicles—interchange of digital information—controller area network (can) for high-speed communication. *International Standards Organization, Switzerland*, 1993.

[267] Hans-christoph Steiner. Firmata: Towards making microcontrollers act like extensions of the computer. *New Interfaces for Musical Expression*, pages 125–130, 2009. URL http://archive.notam02.no/arkiv/proceedings/NIME2009/nime2009/pdf/author/nm090182.pdf.

[268] Doug Stiles. The hardware security behind azure sphere. *IEEE Micro*, 39(2):20–28, 2019.

[269] Petteri Teikari, Raymond P Najjar, Hemi Malkki, Kenneth Knoblauch, Dominique Dumortier, Claude Gronfier, and Howard M Cooper. An inexpensive arduino-based led stimulator system for vision research. *Journal of neuroscience methods*, 211(2): 227–236, 2012.

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

[271] Donald Thompson and Rob Miles. *Embedded programming with the microsoft. net micro framework*. Microsoft Press, 2007.

[272] Donald Thompson and Colin Miller. Introducing the .net micro framework. *Microsoft Corporation*, 2007.

[273] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[274] TIOBE. index | tiobe - the software quality company. https://www.tiobe.com/tiobe-index/, 05 2020. (Accessed on 08/05/2020).

[275] Franklyn Turbak, Mark Sherman, Fred Martin, David Wolber, and Shaileen Crawford Pokress. Events-first programming in app inventor. *Journal of Computing Sciences in Colleges*, 29(6):81–89, 2014.

[276] Franklyn Turbak, David Wolber, Mark Sherman, Fred Martin, and Shaileen Crawford Pokress. Events-First Programming in App Inventor. *Journal of Computing Sciences in Colleges*, 29(6):81–89, 2014. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=2602739.

[277] Antonis Tzounis, Nikolaos Katsoulas, Thomas Bartzanas, and Constantinos Kittas. Internet of things in agriculture, recent advances and future challenges. *Biosystems Engineering*, 164:31–48, 2017.

[278] Ishaq Unwala, Zafar Taqvi, and Jiang Lu. Thread: An iot protocol. In *2018 IEEE Green Technologies Conference (GreenTech)*, pages 161–167. IEEE, 2018.

[279] Eben Upton and Gareth Halfacree. *Raspberry Pi user guide*. John Wiley & Sons, 2014.

[280] Kristof Van Laerhoven, Nicolas Villar, Albrecht Schmidt, H-W Gellersen, Maria Hakansson, and Lars Erik Holmquist. Pin&play: the surface as network medium. *IEEE Communications Magazine*, 41(4):90–95, 2003.

[281] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[282] Ankush Varma and Shuvra S Bhattacharyya. Java-through-c compilation: An enabling technology for java in embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 3, pages 161–166. IEEE, 2004.

[283] Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. Programming microcontrollers in ocaml: the ocapic project. In *International Symposium on Practical Aspects of Declarative Languages*, pages 132–148. Springer, 2015.

[284] Nicolas Villar, Kiel Gilleade, Devina Ramdunyellis, and Hans Gellersen. The voodooio gaming kit: a real-time adaptable gaming controller. *Computers in Entertainment (CIE)*, 5(3):7, 2007.

[285] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. NET gadgeteer: A platform for custom devices. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7319 LNCS:216–233, 2012. ISSN 03029743. doi: 10.1007/978-3-642-31205-2_14.

[286] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. . net gadgeteer: A platform for custom devices. In *International Conference on Pervasive Computing*, pages 216–233. Springer, 2012.

[287] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, pages 1–12, 2020.

[288] Roy Want. An introduction to rfid technology. *IEEE pervasive computing*, 5(1):25–33, 2006.

[289] Gordon Williams. *Making Things Smart: Easy Embedded JavaScript Programming for Making Everyday Objects into Intelligent Machines*. Maker Media, 2017. ISBN 978-1680451894.

[290] Amanda Wilson and David C Moffat. Evaluating Scratch to introduce younger schoolchildren to programming. *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group*, pages 64–75, 2010.

[291] Wonder Workshop. Cue robot - wonder workshop - us. https://www.makewonder.com/robots/cue/. (Accessed on 03/29/2020).

[292] Dingwen Yuan and Matthias Hollick. Ripple: High-throughput, reliable and energy-efficient network flooding in wireless sensor networks. In *2015 IEEE 16th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–9. IEEE, 2015.

[293] Philipp Zenker, Silvia Krug, Michael Binhack, and Jochen Seitz. Evaluation of ble mesh capabilities: A case study based on csrmesh. In *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 790–795. IEEE, 2016.

[294] Jim Zyren and Wes McCoy. Overview of the 3gpp long term evolution physical layer. *Freescale Semiconductor, Inc., white paper*, 7:2–22, 2007.