Kids, Coding, and Connections:
Extending the ScratchJr Programming Environment
to Support Wireless Physical Devices

by Timothy R. Mickel

S.B. Computer Science and Engineering
Massachusetts Institute of Technology, 2014

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 2015

© 2015 Massachusetts Institute of Technology. All rights reserved.
Kids, Coding, and Connections:  
Extending the ScratchJr Programming Environment  
to Support Wireless Physical Devices

by Timothy R. Mickel

Submitted to the Department of Electrical Engineering and Computer Science

August 18, 2015

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in  
Electrical Engineering and Computer Science

ABSTRACT

This thesis presents a system and design for the connection of the ScratchJr programming  
environment to wireless physical devices. Using ScratchJr, young kids can create  
interactive stories, animations, and games on mobile tablet devices. With the extension  
system developed in this thesis, young kids can program wireless lights, motors, sensors,  
toy robots, and other Bluetooth Smart products and devices. Developers can use the  
framework to connect ScratchJr to more devices, and a streamlined programming  
interface is provided for this purpose. The thesis discusses the implementation of several  
example extensions, and it compares the design and functionality to similar mobile  
programming environments.

This thesis is based on research supported by the National Science Foundation under  
grant number 1118682. Any opinions, findings, and conclusions, or recommendations  
expressed in this document are those of the author and do not necessarily reflect the  
views of the National Science Foundation.
Acknowledgements

If you asked me some time ago (possibly only weeks ago), I would have never thought completing this thesis would be possible. Many people have contributed to this work, so I can only begin to express my thanks.

I would first like to thank my advisor, Mitchel Resnick, for his vision, ideas, and enthusiasm throughout this project, from conception to completion. I feel lucky to have had such a supportive advisor - thanks, Mitch.

To the entire Lifelong Kindergarten group and the Scratch Team: thank you for your unique excitement, friendship, inspiring attitudes, and collaborative spirit. Every one of you has been a pleasure to work with and spend time with! Thank you to Amos Blanton and John Maloney for supporting my work throughout my time at MIT and encouraging me to continue with the Scratch project this year.

I'd also like to thank the extended ScratchJr team including the DevTech Group at Tufts and the Playful Invention Company. Special thanks to Paula Bontá and Brian Silverman for their sharp suggestions and feedback. Thanks to Champika Fernando and Chris Garrity for working with me on ScratchJr.

Thank you to the creators of the technologies and platforms upon which this work builds.

Finally, I'd like to sincerely thank my friends and family. Thank you for your endless encouragement, sacrifice, and inspiration.
Contents

1. Introduction
   1.1. Introduction to ScratchJr and Scratch
   1.2. Related work: physical-world programming and construction kits
   1.3. Related work: Scratch extensions for the physical world
      1.3.1 Scratch Sensor Board (PicoBoard)
      1.3.2. LEGO WeDo Construction Kit
      1.3.3. Makey Makey
      1.3.4. ScratchX
   1.4. Introduction to the "Internet of Things"

2. Technical and platform background
   2.1. Overview of Bluetooth Smart
      2.1.1. Advertisement and connection
      2.1.2. The Bluetooth Generic Attribute Profile (GATT)
      2.1.3. Security considerations
      2.1.4. iOS abstraction: Core Bluetooth
   2.2. Overview of ScratchJr implementation

3. User experience design and discussion
   3.1. Introduction to devices-as-characters: "all the world's a stage"
   3.2. Connection design
   3.3. Palette and blocks design
   3.4. Connectivity indicators
   3.5. Messaging between devices
   3.6. Programming block considerations
      3.6.1. "Threshold" and "edge" trigger blocks
      3.6.2. Internal timing management and synchronous execution
      3.6.4. Conflicting block behavior

4. Extension system and implementation
   4.1. System design
   4.2. Structure of code for an extension
   4.3. Discovery and connection flow implementation
   4.4. Reconnection mechanism and silent discovery implementation
      4.4.1. Heuristics for deciding the device
      4.4.2. Compatible devices
   4.5. Bare-bones Bluetooth Smart API (BBT)
   4.6. Block implementations

5. Extension examples
   5.1. Dash and Dot
5.2. Sphero and Ollie
5.3. Light Play (and AdaFruit BlueFruit LE)
5.4. Intel Edison (using Bleno)
5.5. KIBO (extension created by Tufts DevTech Group)
5.6. Proprietary prototypes

6. Comparisons to other mobile physical-world programming
   6.1. Tickle
   6.2. Dash and Dot Blockly
   6.3. Sphero and Ollie orbBasic, MicroLab, SPRK

7. Future work
   7.1. Nearby device disambiguation and alternate connections
   7.2. Multi-platform support
   7.3. Mobile form factors
   7.4. Extension loading and sandboxing
   7.5. Additional on-screen experiences and representations

9. References
List of Figures

1-1: Screenshot of ScratchJr default editor running an example project.
1-2: Official photograph of the PicoBoard.
1-3: LEGO WeDo alligator that can be programmed to bite when objects are brought near.
1-4: Official photograph of a Makey Makey.
1-5: Screenshot of ScratchX main interface with blocks from an Arduino extension.
2-1: Bluetooth Smart Central/Peripheral model.
2-2: Illustration of Bluetooth Smart advertising, scanning, and connection.
2-3: Simplified illustration of Bluetooth Generic Attribute Profile.
2-4: ScratchJr technical organization. Thanks to Mark Roth for the original diagram.
3-1: Device discovery and connection design in ScratchJr.
3-2: Custom block palettes for extension characters.
3-3: Icons labelling physical extension blocks.
3-4: ScratchJr connectivity indicators and blocks when devices are disconnected.
3-5: Illustration of messaging in ScratchJr.
3-6: In ScratchJr, users choose the "color" of a message. Sending a red message, for example, causes all red trigger blocks to run.
4-1: High-level extension system design.
4-2: Code structure of a basic ScratchJr extension.
4-3: Illustration of device discovery and connection from the perspective of an extension.
4-4: Illustration of automatic device reconnection mechanism.
4-5: Example code for a custom extension block definition.
5-1: Official photograph of Wonder Workshop's Dash and Dot robots.
5-2: ScratchJr interface with blocks to drive the Dash robot.
5-3: ScratchJr blocks to set lights on Dash and Dot and control Dash head positions.
5-4: A set of example scripts for Dash. Dash will drive forward when the top button is pressed and drive backward when it senses an object "near" it in the front. A final version would use only icons.
5-5: Blocks to play sounds on Dash and Dot. A final version might use sound icons.
5-6: Official photographs of Ollie (left) and Sphero (right).
5-7: A Sphero script which sets the light and drives.
5-8: Example Light Play setup, including controllable turntable and lights projecting onto screen.
5-9: Light Play prototype Arduino shield by Robbie Berg and Adafruit BlueFruit LE.
5-10: Official photograph of Intel Edison.
5-11: Official photograph of KIBO robot.
5-12: A sampling of ScratchJr KIBO blocks.
6-1: Screenshot of Tickle interface with Dash blocks.
6-2: Screenshot of Dash and Dot Blockly app.
6-3: Screenshots of Orbotix MicroLab and orbBasic.
6-4: Screenshot of Orbotix SPRK environment.
7-1: Screenshot of Pocket Code application for mobile block-based programming.
7-2: Screenshot of loading extensions on ScratchX from a file or URL.
1. Introduction

With ScratchJr, young kids can create on-screen interactive stories, animations, games, and other media on tablet devices in a graphical programming environment. At the same time, wireless physical devices, including a variety of controllable toys, remotely readable sensors, and individually programmable motors and actuators are becoming cheaper and more widely available with the growing "Internet of Things" movement. Many of these devices can communicate with tablets using Bluetooth Smart. ScratchJr was originally designed for on-screen creation, but now there is an opportunity to extend the same programming environment beyond the screen.

In this thesis project, I have created an extensions framework and user interface for connecting ScratchJr to these wireless devices. In doing so, I hope to have demonstrated some of the interesting applications of connecting tablet-based programming environments to tangible physical devices. The physical devices can be manipulated by programming, and sensors from these devices can also be used to affect the virtual characters on-screen. Adding multiple devices to the common, generic programming interface also opens doors to connecting previously unrelated and incompatible physical objects together in creative projects.

In this chapter, I will describe related work and the foundations upon which this work is built.
1.1. Introduction to ScratchJr and Scratch

ScratchJr [1] is a block-based graphical programming language developed specifically for young kids, ages 5 to 7. It was first released in 2014 as an iOS application only, with an Android version added in 2015.

The programming language aims to make digital creativity accessible to these young audiences, and in doing so helps develop high-level thinking skills including literacy, numeracy, sequencing, estimation, prediction, and so on.

![Screenshot of ScratchJr default editor running an example project.](image)

Kids who use ScratchJr can tell stories by adding characters (including animals, people, props, and user-created graphics and photos) to a virtual page (sometimes known as a stage). A ScratchJr project contains up to four pages, each with a particular background image, a set of characters, and sometimes text. The characters can then be programmed
using a palette of drag-and-drop blocks in the scripting pane at the bottom of the screen. Blocks can be snapped together into "strips" which run in sequence.

ScratchJr blocks are organized into five categories:

- **Triggers**: blocks that run on a particular event, such as when the "green flag" at the top of the screen is tapped, when the character is tapped, when the character bumps another character, or when a message is received. Triggers are especially important in ScratchJr, since there is no explicit conditional execution ("if", "else", etc.) as in many programming languages. Instead, conditional execution can be implicit in a trigger ("trigger these blocks when this event occurs").

- **Motion**: blocks that manipulate the position and orientation of the character, including "move left," "turn right," "hop," and so on.

- **Looks**: blocks that change the size and visibility of the character, or prompt the character to "say" a message on-screen.

- **Sounds**: blocks that play sound on the tablet, including a "pop" sound and user-recorded sounds.

- **Control**: blocks that affect the control flow of the strip, including repeating blocks and changing the speed at which other certain blocks run.

- **Ends**: blocks that specify the behavior at the end of the strip (stop, repeat forever, or change the page).

Many of the design concepts used in ScratchJr are based on Scratch. Scratch is a block-based programming language designed for people ages 8 and up. It is also
integrated with a large online creative community. The Scratch project seeks to help people learn to "think creatively, reason systematically, and work collaboratively."

One of the goals in the development of the Scratch programming language and environment is to support self-directed learning through tinkering and peer-to-peer collaboration. The environment is media-rich and attempts to be more tinkerable, more meaningful, and more social than traditional programming environments [2]. To accomplish these goals, the Scratch programming environment was constructed intentionally with specific features:

- A single-window user interface for easy navigation with important components visible at all times.
- Liveness and tinkerability: no compilation step or edit/run distinction; clicking on code makes it run immediately, and scripts can be actively modified while the code is running. These features help users discover the functionality of the programming blocks.
- Visible execution: visual feedback on what code is running and for how long.
- No error messages: blocks only fit in ways that make sense, out-of-range values are replaced with reasonable limits, and blocks almost always do something even when scripts are constructed with errors.
- Concrete data: typically abstract ideas like variables and lists are displayed on the screen.
- Minimized command set: blocks are grouped and only appear when relevant to avoid overwhelming beginners [3].
ScratchJr reproduces many of these design ideas in principle but is retooled to capture the same ideas in a way that is appropriate for younger children. The development of ScratchJr was driven by noticing some significant struggles young children encountered in trying to use Scratch. To program using Scratch, users must have the ability to read text, understand relatively large numbers, non-integers, negative numbers, and so on. Programming blocks in Scratch can also have effects that lack visible results on the screen (or make instantaneous, unnoticeable changes) which proved difficult. In ScratchJr, blocks always take a measurable amount of time to execute, and if possible, a transition occurs on the screen (e.g., characters glide along the page). By replacing text-based blocks with icons and limiting default numerical values to whole numbers, the core ideas of Scratch are made accessible without the need for advanced literacy and mathematical ability. Young kids might also be unfamiliar with the use of a mouse and keyboard, making a touch-based environment more accessible.

ScratchJr is designed to be "developmentally appropriate" for children aged 5-7. The principles driving its design are:

- Low floor and appropriately high ceiling: easy to get started with and capable of complexity, but manageable for the target age.
- Wide walls: allowing many pathways and styles of learning and exploration and creation.
- Tinkerability (as in Scratch): enabling experimentation with features.
- Conviviality: a playful and inviting environment.
- Classroom support [4].
ScratchJr also has many features of Scratch, including liveness, visible code execution, no error messages, and an (even further) minimized command set. Data in ScratchJr is minimized, but where used it can be made visible (e.g., the position of characters on the screen can be displayed on a grid). The ScratchJr interface is mostly contained in a single window, with some dialogs where necessary (e.g., to add characters or share a project).

In Chapter 3, I discuss how the principles and designs of Scratch and ScratchJr have influenced and are embedded in the design of the physical world extensions presented by this thesis.

1.2. Related work: physical-world programming and construction kits

Scratch builds on the ideas of constructionism and ultimately has roots in the Logo programming language. Returning to these roots, there is a long history of projects that explore programming with physical, tangible devices. Robotic "floor turtles," programmed with Logo and studied by Seymour Papert, were demonstrating some of the potentials of this project more than forty years ago. In Papert's explorations, children wrote programs to move the physical turtle on the floor and draw pictures on paper. These turtles provided children with what Papert called "objects-to-think-with": objects "in which there is an intersection of cultural presence, embedded knowledge, and the possibility for personal identification" and which enable the discovery of "powerful ideas" [5]. By providing kids with the opportunity to program devices that are tangible and relevant in the world around them, we also hope to create the right environment to have these types of learning experiences.
The LEGO/Logo project by Resnick et al. brought motors, sensors, and Logo programming to physical LEGO projects in the late 1980s [6]. The Programmable Brick [7] embedded a computer directly into the LEGO brick, allowing for diverse and autonomous creations; the popular LEGO Mindstorms is based on this concept, and smaller and simpler versions were also developed (such as the Cricket). These extensions of programming to the physical world partially flip the equation from "manipulating worlds in the computer" to "manipulating computers in the world," and several of the lessons learned in these projects influence this thesis. Some of the relevant implications of these projects include:

- Designing for low floor (ease of entry) and wide walls (engaging kids in their interests and passions which have a diversity of possible outcomes).
- Creating the simplest possible user experience (which can foster creativity).
- Choosing "black boxes" carefully: the "building blocks" of the experience should exist at a level intentionally chosen and hide lower-level pieces that would be distracting to building and creation.
- "A little bit of programming goes a long way:" even simple programs can be engaging and enrich a creative project [8].

As mentioned, ScratchJr's interface attempts to be very low-floor to create accessibility for young kids. By adding the ability to program wireless devices, I hope to "widen the walls" of ScratchJr: to widen the space for creative outcomes and engaging kids with a variety of physical connections. In our example extensions, the black boxes
were chosen to enable the control of physical devices with minimal configuration: these choices are discussed more in Chapter 3 and Chapter 5.

1.3. Related work: Scratch extensions for the physical world

Several projects have connected Scratch to the physical world. I will give a few examples that also serve as inspiration and models for my extensions to ScratchJr.

1.3.1 Scratch Sensor Board (PicoBoard)

Figure 1-2: Official photograph of the PicoBoard.

Amon Millner's Scratch Sensor Board (now known as the PicoBoard) connects Scratch projects to the physical world to provide sensing information about the world and react in interesting ways [9]. Scratch Sensor Boards have buttons, sliders, light-intensity sensors, sound-level detectors, and alligator clips for attaching any kind of sensor.

Millner's research focused on three types of connections to Scratch: "basic" (everyday objects with sensors), "repurposed" (reuse of an existing electronic object to
control Scratch projects), and "fabricated" (created with personal fabrication tools like mills and laser cutters). Millner also cited three design styles that this system enabled for its users: "building to think" (making ideas tangible and concrete to enable thought), "learning faster by failing early (and often)" (enabling iteration), and finally, "permission to explore new behaviors with new materials" (e.g., reusing engaging objects from childhood).

One hope of this project is to make these modes of creation more accessible to more people. Adding programmability to wireless devices may help kids become more engaged in creative exploration with everyday objects, and the mobile nature of this project may enable a greater ability for creators to iterate and experiment away from a traditional computer.

1.3.2. LEGO WeDo Construction Kit

![LEGO WeDo Construction Kit](image)

*Figure 1-3: LEGO WeDo alligator that can be programmed to bite when objects are brought near.*

The LEGO WeDo Construction Kit [10] includes motors, lights, tilt sensors, distance sensors, and hubs to connect to computers. An extension for Scratch connects the WeDo
to Scratch projects. The current WeDo is wired together (and must be wired to the computer to work with Scratch). By bringing wireless programming to devices like an imagined future WeDo, our project could untether devices to encourage greater play and tinkering, while still maintaining instant and readily modifiable programming.

1.3.3. Makey Makey

![Makey Makey](image)

Figure 1-4: Official photograph of a Makey Makey.

Makey Makey [11], when connected to a computer, simulates keypresses upon completion of a user-created electric circuit. Banana keyboards, custom game controllers, and an Operation game powered by Scratch are some of Makey Makey’s usage examples. Scratch integration works the same as integration with other computer programs: since Scratch can detect key presses, any completion of an electrical circuit can trigger an event in Scratch.

1.3.4. ScratchX

ScratchX [12], publicly released in 2015, is an open platform for the creation of Scratch extensions. Extensions in ScratchX are implemented in JavaScript and can create any
number of programming blocks to interact with physical devices, web services, online data, etc.

One popular extension is the Arduino extension [13], with which people can read and control Arduino pins over USB. Other extensions include text-to-speech blocks and "Mesh" which allows programmers to send broadcast messages across multiple projects.

![Figure 1-5: Screenshot of ScratchX main interface with blocks from an Arduino extension.](image)

The ScratchX project and this thesis project share a lot in common, including the generic interface for adding new extensions. Our project considers physical devices as distinct "characters" to be programmed (see Section 3.1) which is different from ScratchX. We also focus on wireless devices in particular and provide a more-simplified set of programming blocks (with typically fewer parameters and immediately obvious physical effects).
1.4. Introduction to the "Internet of Things"

A rising trend in consumer technology is the creation of "Internet of Things" (IoT) devices. The term describes the product trajectory of more and more everyday technologies and objects becoming connected to the Internet.

In such a world where many objects are connected to the "Internet of Things," everyday objects like light bulbs, thermostats, kitchen appliances, laundry machines, security systems, and so on, could become controllable by computers. In addition, these devices could seamlessly notify their owners of changes in state (for example, a connected washing machine could push a notification to signal it's time to switch to the dryer).

The devices we choose to focus on in this project are suitable for use by young kids: toys, simple robotics, expressive lights and motors, and so on. Not all of these devices are intended to be connected in an "Internet of Things," but a common thread is the ability to wirelessly connect to a computer (in our case, a tablet computer).

Connection to ScratchJr raises the possibility of adding wireless programmability to IoT and wireless devices. Excitingly, it would also allow beginners and kids to program these devices. In this way, we hope to enable people to express themselves creatively and manipulate the world around them in interesting ways without having in-depth knowledge of programming syntax, APIs, wireless network protocols, etc.
2. Technical and platform background

In this chapter, I will give an overview of the technical platforms relevant to the implementation.

2.1. Overview of Bluetooth Smart

Bluetooth Smart, also known as Bluetooth Low Energy (LE) and Bluetooth 4.0, is specifically constructed for "Internet of Things" applications. The stack is designed to be low cost, low power, and easy to implement. While "classic" Bluetooth radios are designed primarily for mobile phones, voice, and audio transfer, Bluetooth Smart is instead intended for lower-bandwidth data transfer, including for use by wearables, home devices, and the devices used in this project. In this section, I will give an overview of some of the technical structures of Bluetooth Smart (at the application level). Most of our example ScratchJr extensions communicate with devices using Bluetooth Smart.

![Figure 2-1: Bluetooth Smart Central/Peripheral model.]

Bluetooth Smart devices can generally be organized into two relevant categories: "central" devices and "peripheral" devices. Peripheral devices serve small amounts of
data to central devices, while central devices explore and manipulate the data available on peripherals [14].

The most common central devices are smartphones, tablets, and computers (but any device could be created to be a Bluetooth central). Most peripherals are small, portable devices with sensors or actuators, like a wearable smart watch or heart rate monitor.

In our project, the central device is the iPad and the peripheral devices are the toys, motors, and sensors to which ScratchJr is connecting and with which ScratchJr is communicating.

2.1.1. Advertisement and connection

In Bluetooth Smart, a peripheral device that is powered on will broadcast periodic advertisements. Such an advertisement contains a minimal amount of data describing the device's capabilities so central devices can discover and choose to connect to it. Advertisements are broadcast regardless of the presence of nearby centrals.
A central device, when appropriate, can wirelessly scan for nearby peripheral devices to receive the advertisement data and signal strengths. If the central finds a device it wishes to connect to, it enters an "initiated" state and broadcasts to the peripheral the desire to connect. When the connection has been established, both devices are in the connected state. A peripheral generally connects to one central at a time and stops broadcasting advertisements once connected. As a result, other present central devices will not usually detect peripheral devices that are occupied with a connection.

2.1.2. The Bluetooth Generic Attribute Profile (GATT)

Bluetooth Smart devices are required to implement the Bluetooth Generic Attribute Profile (GATT) [15]. This is a generic format and hierarchy for describing how data is organized on the devices. GATT views devices as either servers (which have data) or clients (which browse server data), and in our project this aligns with the peripheral (has data) and central (browses and manipulates data) respectively.

![Figure 2-3: Simplified illustration of Bluetooth Generic Attribute Profile.](image)

Servers (in our application, these are generally Bluetooth Smart peripherals) can provide multiple services. A service can provide both data and behavior functions and is given a unique identifier so other devices can recognize it. If a service describes the primary
purpose of a device, the identifier is often included in the peripheral's advertisement so central devices can immediately recognize compatible devices.

Services can include a collection of characteristics. A characteristic defines a value, information about a value, and methods for interacting with it. For example, a characteristic could represent a heart rate monitor's "beats per time" value. In this case, the value might be readable but not writeable (this is described in the characteristic properties) and might include a descriptor to specify the unit of time (for example, "minutes").

This illustration of the Bluetooth GATT (and Bluetooth Smart) is simplified and does not describe the full extent of the system. However, this describes most of the knowledge needed for understanding and implementing ScratchJr Bluetooth extensions.

2.1.3. Security considerations

Security is not a critical consideration in this project since the intention is to use Bluetooth devices for local learning and play devices. For applications like smart watches, health data peripherals, and connected-home Internet of Things devices, Bluetooth Smart provides a pairing mechanism and connection encryption to provide authentication, trust, and privacy. Our current implementation is focused on the design and infrastructure for programming devices, but a future version might implement these Bluetooth security features to provide support for more sensitive IoT devices [16].

2.1.4. iOS abstraction: Core Bluetooth

Our implementation uses the Core Bluetooth library provided by Apple for iOS to provide the central Bluetooth role on iPads.
2.2. Overview of ScratchJr implementation

The user interface, block engine, interpreter, project display, and most other parts of the ScratchJr front-end are implemented using JavaScript and HTML. These are displayed in a UIWebView on the iOS app. To store data, access the camera and microphone, and perform other device functions, there is a thin layer of native code (written in Objective-C). This native layer and the JavaScript application can communicate via the JSContext API (and thus natively implemented code can call JavaScript and JavaScript can call natively implemented code).

Our extension system takes advantage of this structure to keep most of logic cross-platform, for a future implementation on Android and other mobile platforms.
3. User experience design and discussion

In this chapter, I will give an overview of ScratchJr extensions from the perspective of a ScratchJr user and discuss the design choices that have influenced the interface.

3.1. Introduction to devices-as-characters: "all the world's a stage"

In ScratchJr, agents that can be programmed are represented as characters. The characters live on a virtual "stage." Our extensions provide additional characters which represent physical objects in the world, and instead of just the on-screen stage, "all the world's a stage." For example, we might consider a playful robot to be another character in a story being told; it is represented this way in the user interface.

This physical device might communicate with on-screen characters (and other physical devices) in the same ways ScratchJr characters communicate with each other (see Section 3.5). The user interacts with and programs the physical device using blocks just as they might program characters.

This design was chosen intentionally both to fit well with the existing ScratchJr metaphor and because it allows the user to think clearly and intentionally about which physical object they are programming. The user can select a physical character and immediately see the corresponding physical device (described in Section 3.4), all the blocks which are affecting the behavior of the physical world object, and when the blocks are running. Device characters are present and when they are available and detected in the physical world.
This model means projects with physical-world connections can be readily shared with other people, and it is immediately obvious what kinds of devices the project uses (without inspecting the programming blocks). Lack of a physical device should not break the project except some physical programming blocks might not have effects.

In general, one character is created for each wireless connection (typically each physically-connected component). Each component that would be considered a single wireless "device" is represented as one character. In some cases, multiple sensors and actuators are a part of one device. But because they are all physically connected, they would usually be considered one character in our system. This is both for technical convenience and because it ensures a consistent user experience: for example, if the device is disconnected, all of its blocks are disabled in only one character.

By treating each device as its own character, mutually compatible physical devices (e.g., two of the same type of sensor or robot) might be interchanged. Multiple devices of the same type can coexist in a project and this is visually clear and possible within our implementation.

There are some disadvantages to this approach. In ScratchX, when devices are added to Scratch projects, all sprites are provided with a new set of blocks that provide interaction with the device. This alternative view of devices has its own advantages: sprites can be more tightly integrated with devices without using broadcasts and messages. Creating interactions between characters and devices can become slightly more difficult by requiring messaging, although this also allows the user to program the device without necessarily considering character interaction.
Presently, code for physical devices is run on the tablet and commands are streamed to the device. In the future, the device-as-character model of programming devices might help us imagine sending complete programs to the device itself where the program could continue to run autonomously without the programming device present. By separating code intended for the device itself (rather than integrating it with existing characters or sprites), the user is constructing code that describes what the device should do and how it should communicate feedback.

More implications, benefits, and downsides of the device-as-character idea are discussed throughout this chapter and thesis.

3.2. Connection design

Figure 3-1: Device discovery and connection design in ScratchJr.
Using the device-as-characters metaphor, users can add physical-world devices in the same way they usually add virtual characters. When the user taps on the + icon to add a new character (or physical device), the system opens a library that now includes not only on-screen characters but also wireless devices that are present and available. All of the devices are displayed with an icon, the name of the device, and a small bubble to indicate connection status. Tapping the device icon (as a user would tap a character) connects in the background and adds it to the project for immediate use.

This experience was minimized to need as little user intervention as possible. We could go further, perhaps, and automatically add nearby devices to users' projects. However, we decided against this considering the "classroom" case in which many unrelated devices could be in the same location as the user. Keeping the devices disconnected until needed allows for harmonious use in a setting with many devices or many programmers. In a case where there are many devices present, the devices are sorted on the library screen by connection strength.
3.3. Palette and blocks design

Extensions provide new programming blocks that affect or reflect the state of the physical device. In the example in Figure 3-2, the default ScratchJr blocks that affect how a character appears (in size, visibility, etc.) do not make sense given the physical constraints of the device. However, a new block is provided instead: one that can change the color of the lights on the robot.

Extensions always group blocks in the same categories as ScratchJr virtual characters, removing the need for users to learn a new set of categories. Extension developers can also choose to include default blocks in the set, such as the program flow.
control blocks which are just as applicable to extension programming as virtual programming (see Section 4.6). In our example extensions (Chapter 5) these logical categories have largely covered the abilities of the devices. In a version of this for older users, we might consider allowing extension developers to create new categories for blocks.

Custom block palettes for each physical character follows from Scratch's goal to minimize command sets when possible. In the alternate design choice in which extension blocks are added to existing characters (and devices are not represented by characters), users might have to juggle many blocks in the palette at the same time, all of which might have effects in different places in the world or on-screen. Adjusting the palette for each device hopefully ensures maximal utility with minimal confusion.

![Figure 3-3: Icons labelling physical extension blocks.](image)

Extension blocks are displayed with a small translucent icon hovering above them to help the user recognize that the blocks are special and different from those affecting the virtual world. Distinguishment is important if the icons are similar to those used in the virtual palette. For example, in Figure 3-3, the Dash robot's movement icons are the same as those that might be used to move the on-screen cat. This choice to reuse block icons is intentional (so users can immediately interpret what physical effects a block might have, given their knowledge of the virtual blocks). The device icon is a simple way to further
disambiguate the experience. However, the icons shown in this thesis are a preliminary prototype: in a final version, we might consider moving the icon from outside the block to inside it or otherwise integrating the two icons within the block. Another way to distinguish the blocks would be by showing a device watermark under the scripting area, as is done for characters on screen.

As in the rest of ScratchJr, blocks are highlighted when running and almost always create visible, tangible effect in the physical world over a noticeable time period. Extension developers determine the exact behavior of the blocks, and users can discover the behavior of the blocks by tinkering (tapping the blocks and noticing the effect on the physical device).

3.4. Connectivity indicators

Figure 3-4: ScratchJr connectivity indicators and blocks when devices are disconnected.
When devices are first added to ScratchJr projects, they are connected immediately. However, while the project is open or even while the blocks are running, physical issues can cause the device to become disconnected (for example, if the device runs out of battery or goes out of range). When this is detected, the connectivity indicator is updated and the blocks themselves change in visual appearance to show that they no longer have the desired effect. The blocks do not raise any direct errors that interrupt program execution, but instead visually show that they are deactivated. When the device is turned on again or comes back in range, the blocks are re-activated.

A similar pattern is followed when a project is first opened. Devices and extension blocks appear in the project, but are inactive until the device is rediscovered and reconnected to silently in the background by ScratchJr. Reopening the project and reconnecting to devices requires no intervention by the user (other than turning the device on and putting it in range).

Finally, users can discover which physical devices are connected to which icon on the screen by tapping the icon in the list of characters. When the icon is tapped, most physical devices will briefly flash white or otherwise make their connection visible.

The final visual design of the connectivity indicator (currently a green or red circle near the device icon) is not yet decided. We might also distinguish physical devices in the list of characters by a different visual design.
3.5. Messaging between devices

ScratchJr supports interaction of characters through "messaging" blocks. These blocks are included our physical devices' block palettes as well. This system provides lightweight messaging between on-screen characters and physical devices.

![Diagram of messaging in ScratchJr](image)

In Figure 3-5, we can see an example of how the message blocks can enable characters to affect devices. The reverse is also possible: actions or measurements in the physical world can trigger devices to send messages which change the appearance and behavior of virtual characters. Different messages can cause different actions, as determined by the user's choice of color (see Figure 3-6).
Figure 3-6: In ScratchJr, users choose the "color" of a message. Sending a red message, for example, causes all red trigger blocks to run.

Presently this is the only means of interaction between characters and devices, and this is an intentional choice. Messages are generic and using them as a common tool among all characters and devices ensures compatibility. These are a low-floor and tangible, illustrated way to see how objects can communicate. And once understood, messages are powerful and could lead to diverse creations.

3.6. Programming block considerations

3.6.1. "Threshold" and "edge" trigger blocks

Extensions sometimes include trigger blocks that react to physical conditions in the world. For example, if a button is pressed or a light sensor hits a certain threshold we might start running a strip of blocks.

For triggers, there is a design question of whether a strip of blocks should be triggered repeatedly while a value is above/below a threshold ("threshold triggers") or if it should trigger once when the transition is crossed ("edge trigger"). We left this decision up to the extension developer because in some physical devices one might be preferable over the other.
One thing worth noting is that these two block strips roughly produce equivalent results (assuming threshold triggers are not repeatedly restarted):

\[ \text{[Trigger repeatedly while above/below threshold]} \rightarrow \text{[Commands]} \]

should be equivalent to:

\[ \text{[On green flag]} \rightarrow \text{[Wait until above/below threshold]} \rightarrow \text{[Commands]} \rightarrow \text{[Repeat]} \]

While the second version is more verbose, it has the advantage of keeping the trigger about discrete "one-time" events or changes rather than reflecting an ongoing condition in the system. In our example implementations we exclusively used the edge triggers and we are adding "wait until" blocks which wait until a condition becomes true. However, we have not yet developed a final opinion on what combination of waits and trigger types is preferable in the ScratchJr language, so the implemented extension system is neutral to this.

In all triggers, we choose the condition/thresholds to be black-boxed. Since these connections are intended for young kids, figuring out a numerical threshold for a desired effect will be far less interesting and instructive than creating behaviors and effects that react to a reasonable threshold. For example, when dealing with devices that have distance sensors, we generally chose a value that meant "near" (5 cm, for example).

3.6.2. Internal timing management and synchronous execution

In all blocks, timing is managed by the block itself rather than the user. For example, Bluetooth devices can receive commands at very high rates so we could change the color of a light 20 times in a second or more. But a sensible wait is embedded in "color change" blocks so that the effect is observable.
In ScratchJr, blocks are generally executed synchronously (every action takes place in turn; blocks do not have effects after they have finished visibly executing). Extensions should be designed and constructed to take this into account. For actions that take some time in the physical world, like driving a robot forward, the programming block should never send a command and proceed to the next block. Instead, the block should wait a time and then send a new command to stop/slow the robot before proceeding to the next block. In most cases blocks run quickly enough so that sequential, synchronous blocks can still provide smooth motion.

3.6.3. Usable defaults and parameters

ScratchJr blocks are presently limited to one parameter per block. Extension blocks are designed so default parameters will result in a noticeable effect in the physical world (to ensure tinkerability). In some cases, the ScratchJr programming block does less than what the physical device is capable of, for the sake of ease-of-use and limiting overwhelming choices. Many of the devices our extensions connect to include RGB LEDs which could be programmed to display hundreds of colors. But for many creative applications, a menu of primary colors provides enough choices without an overwhelming user interface.

3.6.4. Conflicting block behavior

Sometimes, blocks that would have conflicting physical-world effects can be put into parallel ScratchJr block strips. A user might tell a device to go backward and forward at the same time. If the extension cannot resolve the desired behavior, among conflicting blocks the latest-reached will take precedence. If "set lights red" is run slightly after "set lights blue," for example, the lights will be red; if one sound is played at the same time as
another in a parallel script, both blocks are highlighted and proceed, but only one sound
is played. If two are started at nearly the same time (e.g. using the same trigger), which
block is run may be nondeterministic.
4. Extension system and implementation

From the perspective of an extension developer, implementing an extension typically involves writing a single JavaScript file to define the capabilities of a device, the programming blocks, and some of the visual experience. In this chapter, I will describe the extension system, including all of the abstractions provided to the extension developer, as well as how extensions are implemented within ScratchJr.

4.1. System design

![High-level extension system design diagram]

 Figure 4-1: High-level extension system design.

Figure 4-1 shows the overall top-level structure of the extension system and how individual extensions fit into the system. A particular extension defines blocks and the functions for device discovery and connection, making calls to either the provided
Bluetooth Smart API we implemented (description in Section 4.5) or to native code as provided by a native device API.

The extension system tracks extensions and calls the appropriate functions when the user takes certain actions. For example, when the user enters the character library, extensions are asked to start discovering physical devices. The extension then must either call native code to start discovering devices or forward the call to our Bluetooth Smart API. The extension system itself handles generating the user interface and manages the state of connections to devices.

To abstract a particular connection to a device, we track "extension device connections." There is a one-to-one correspondence between these and active physical connections to devices. These are managed and updated by the extension system and passed along to the extension itself at particular points in execution. Then the extension itself is connection-agnostic: the implementation of a programming block, for example, doesn't depend on having just one connection to one device. Instead, the implementation is generic for any connection to a compatible device. Multiple devices of the same type can be controlled individually as characters in ScratchJr. The extension device connection can also store information about the device that will be saved across project saves. For example, we can store the name of a particular device so that when reloading the project we are sure to connect to the correct device the blocks were programmed for.

4.2. Structure of code for an extension

Extensions are required to implement four functions to deal with device discovery and connection. They can optionally implement a few others which allow the extension
developer to control a variety of behaviors for the devices. Extensions are structured so that implementation is extremely convenient for Bluetooth Smart devices, but other kinds of connection could be (and have been) implemented as well.

```javascript
var anExtension = new Extension("Extension Name");
anExtension.shutdown = function() {}; // Optional

// Discovery functions
anExtension.startDiscovering = function() {};
anExtension.stopDiscovering = function() {};
anExtension.setupDeviceConnection = function(deviceConnection) {}; // Optional
anExtension.deviceConnectionsCompatible = function(devC1, devC2) {}; // Optional

// Connection functions
anExtension.onConnect = function(deviceConnection) {};
anExtension.onDisconnect = function(deviceConnection) {};

// Display functions
anExtension.getIconForDevice = function(deviceConnection) {}; // Optional
anExtension.getNameForDevice = function(deviceConnection) {}; // Optional
anExtension.flashDevice = function(deviceConnection) {}; // Optional

// Block functions and definitions
anExtension.blocks = [
  //... See Section 4.6
];
anExtension.blockListForCategory = function(deviceConnection, num) {}; // Optional

Extensions.addExtension(anExtension);
```

*Figure 4-2: Code structure of a basic ScratchJr extension.*

The functions that will be used by the extension system are:

- **shutdown**: called when the extension will be unloaded by the system.
- **startDiscovering**: called when the user opens the library; extension should start discovering devices and the native code or Bluetooth Smart API will insert them into its discoveredDevices.
- **stopDiscovering**: user has finished looking for new devices; extension should stop discovering.
• **setupDeviceConnection**: the discovered `deviceConnection` object is passed to the extension. Here the extension can set up data it wishes to be associated with the connection, for example the device type or cached device properties.

• **deviceConnectionsCompatible**: the extension should return `true` if the two connections are "compatible": they represent the same device or same type of device. This is called when devices are silently reconnected (Section 4.4).

• **onConnect**: user has selected a device connection and wishes to connect; extension should connect to the device.

• **onDisconnect**: the device has been removed or the user has exited; extension should disconnect from the device and clean up data.

• **getIconForDevice, getNameForDevice**: for a particular connection the extension should return a display name and a URL for an icon for the device. This can depend on connection properties: we might display a different icon based on the availability of a service, or certain robotic carry names given by the user.

• **flashDevice**: used to call attention to the device in the physical world. If possible, the extension should flash lights on the device, make a noise or a brief movement. This is triggered when the user taps the device icon on the screen.

• **blocks**: a list of implemented blocks and their properties (see Section 4.6).

• **blockListForCategory**: since the blocks available in a particular palette category may depend on properties of the device (e.g., if a Bluetooth Smart service is present), the extension can return a list of block names (defined in `blocks`) for a given device and category.
4.3. Discovery and connection flow implementation

Figure 4-3 shows the lifecycle of an extension device connection and the various pieces of the extension it will touch during device discovery, connection, and the use of programming blocks. This lifecycle is managed entirely by the extension system, so that developers only need to be concerned with the implementation of each of the pieces for a particular type of device.

4.4. Reconnection mechanism and silent discovery implementation

To simplify the experience for end users, upon opening a ScratchJr project previously saved with connected devices, the system attempts to discover and reestablish connections to compatible devices silently and automatically. This should be possible in most applications, except where there might be unlikely technical limitations (e.g., if a Bluetooth device never provides any of the same services, changes its name frequently, and so on).
In Figure 4-4, we show the system process that occurs when a user opens a project with device connections saved. The extension system attempts to reconnect to the closest physical device, as determined by connection strength. The same process is used when a device connection is lost (for example, if the device's battery dies). In this case the system will reconnect to a compatible device.

4.4.1. Heuristics for deciding the device

Which device is reconnected to is determined by "connection strength" which is provided by the instantiator of the extension device connections. If the extension is using our Bluetooth Smart API, the connection strength is determined by greatest RSSI (received signal strength indicator) as reported by Core Bluetooth. When possible, the example extensions using other native APIs were also implemented to consider connection strength. The imagined scenario in which this heuristic is useful occurs in a classroom: if many students are connecting to devices which are all compatible (e.g., the same kind of
device), each will be connected to the device they are physically closest to. In an application intended for older users, we might consider forgoing the heuristic and instead have the user choose the device manually.

4.4.2. Compatible devices

Each extension can determine which of its devices are "compatible." This is important because the extension might discover multiple devices in physical proximity, but it is possible not all of these devices would share the same programming blocks. For example, in Section 5.1, Dash robots and Dot robots have slightly different programming blocks (Dot cannot physically move) but share the same extension and connection structures. The extension itself can save information about a connection and determine if a saved connection is compatible with a discovered device in the deviceConnectionsCompatible method. The extension system creates a dummy device connection augmented with saved information (essentially applying setupDeviceConnection) so that the extension can do the comparison.

4.5. Bare-bones Bluetooth Smart API (BBT)

The extension system provides a thin JavaScript wrapper around Core Bluetooth that covers most functions an extension developer will want to use. This Bare-bones Bluetooth Smart API (BBT) makes connecting to Bluetooth Smart devices a matter of adding a few function calls.

To start, extension developers add these to the extension's startDiscovering and stopDiscovering functions respectively:
- `BBT.startScanning(advertisedServiceUUID, extensionObject);`
  - Triggers Core Bluetooth to handle discovery for devices which are advertising `advertisedServiceUUID`.
- `BBT.stopScanning();`
  - Core Bluetooth will stop searching for these devices

With this done, the BBT library will automatically call `setupDeviceConnection` with a `peripheral` object. The extension developer can then forward the extension's `onConnect` and `onDisconnect` to the peripheral object, and then the peripheral will be associated with the `deviceConnection` and fully ready for exploring services and characteristics.

The peripheral object is populated with this interface, which is designed to reflect the organization of GATT as closely as possible (see Figure 2-3 for a diagram):

- `peripheral.uuid`: unique identifier from Core Bluetooth
- `peripheral.name`: name provided by the Bluetooth peripheral
- `peripheral.connect(serviceUUIDs[], callback)`: connect and attempt to discover all of the services in the provided list
- `peripheral.disconnect()`: disconnect from the peripheral
- `peripheral.services[]`: a list of discovered and available services on the peripheral. Each service has:
  - `service.uuid`: unique identifier for the service
  - `service.characteristics[]`: list of auto-discovered characteristics on the service. Each characteristic has:
    - `characteristic.uuid`: unique identifier
characteristic.read(callback): read a value; callback is called with the latest read value as the first parameter.

characteristic.write(typedValueArray [,callback]): write a value. If Bluetooth write responses are possible for this characteristic, callback is called on a successful write.

characteristic.subscribe(callback): subscribe to notifications, providing a callback for when notifications are received. The latest value of the characteristic is passed to the callback as the first parameter.

characteristic.unsubscribe(): unsubscribe from notifications

With the connection boilerplate implemented, the extension developer only needs to use this peripheral object properties to interact with the device, including reading, writing, and receiving notifications from the device.

4.6. Block implementations

The final piece of an extension implementation is the definition of programming blocks. This typically involves two parts: an addition to the blocks property of the extension, and the implementation of a method which will be called when the block is run. The blocks property is an object; each named property {Events, Motion, Looks, Control, Ends} is a list which corresponds to blocks provided in a category. These can either be ScratchJr default blocks (provided in DefaultBlocks) or custom block definitions.
A custom block definition might look like:

```
{
  name: "extension_block_name",
  blockType: ExtensionBlock.blockTypes.Command, // Or Hat, End, ...
  icon: "URLToBlockIcon.png",
  argumentType: ExtensionBlock.argumentTypes.Number, // String, None, ImageMenu...
  menuOptions: [...], // menu icon URLs for image menus
  argumentDefaultMinMax: [defaultValue, minValue, maxValue],
  primitive: function(thread, sprite, deviceConnection, argumentValue) {},
  stopPrimitive: function(thread, sprite, deviceConnection) {}
}
```

All of these properties are optional. When the block is run, the `primitive` function will be called by the extension system. The function should either call `thread.nextBlock` when done running commands or `thread.yieldBlock` to keep control of the strip and run more JavaScript after time has passed. If the block provides a `stopPrimitive`, this is called when the block's execution needs to be stopped during a yield. This is useful for extensions that send asynchronous commands to devices, for example, a "set motors high" command which needs to be actively stopped by "set motors low."

With this organization, extension developers can implement blocks in a connection-agnostic way (the extension system passes the relevant connection to the primitive each time). User interface code for the blocks, thread management, etc., is fully handled by ScratchJr and the extension system.
5. Extension examples

In this chapter, I present a variety of example extensions constructed using this extension framework. Many of the extensions use off-the-shelf devices and components. These examples are preliminary prototypes, and several details are still under active consideration (such as the particular icons used in blocks).

Some devices with ScratchJr extensions use native device APIs to connect to and communicate with the devices (Dash and Dot, Sphero and Ollie). In these cases, this is necessary because device creators only publish an API (and not the protocol for talking to the device, e.g., descriptions of Bluetooth services and characteristics). Although we have implemented them here using the native API, the device creators could now create ScratchJr extensions with JavaScript.

5.1. Dash and Dot

Dash and Dot [17] are robots produced by the California-based startup Wonder Workshop. The robots, intended for ages five and up, are fully-assembled and available for purchase as a set for $230. Dash, the larger robot, is mobile, with two controllable wheels and a controllable head. Both Dash and Dot can play pre-loaded sounds, shine
multicolored lights in the "head" and "ears," and sense events in the form of button presses, audio events (e.g. clapping), distance to objects, and accelerometer events (e.g., collisions and shaking). The default applications meant for controlling Dash and Dot include "Go," for directly controlling the physical attributes, "Xylo," for playing songs on an attachable xylophone, "Path," for controlling Dash's movement in a drawn path, and "Blockly," a programming environment which we overview in Section 6.1.2.

The robots communicate with tablet devices using Bluetooth Smart. Our ScratchJr extension connects to Dash and Dot using their proprietary API. Any number of the robots can be detected and added on library screen, and a custom set of blocks is available for programming the robots.

Figure 5-2: ScratchJr interface with blocks to drive the Dash robot.
The blocks are chosen to make programming very low-floor: a user can immediately discover the command blocks' functionalities by observing the physical effects on the robots.

*Figure 5-3: ScratchJr blocks to set lights on Dash and Dot and control Dash head positions.*

No configuration or parameters are required for basic use, but the parameters are chosen so the default is small and observable but not physically overwhelming. For example, the forward/reverse motion blocks for Dash act in quarter-second intervals and the turn clockwise/counterclockwise blocks turn at the same rate as ScratchJr characters (twelve turns for a full rotation, like a clock). In ScratchJr the speed of motion is set to be slow by default (but can be increased or decreased with the default "set speed" block). Blocks like "set light" take a measurable amount of time to complete.

*Figure 5-4: A set of example scripts for Dash. Dash will drive forward when the top button is pressed and drive backward when it senses an object "near" it in the front. A final version would use only icons.*

Sensors are encapsulated in the trigger blocks (Figure 5-4). Usable defaults are chosen: for example, button presses are triggered when the button is released, and the "near" distance sensor triggers blocks to run when an object is about 5 cm away.
Our current implemented prototype blocks include:

- **Triggers**
  - Button presses: Dash and Dot both have four buttons on the top of their "head." When one of these is tapped, strips can be started in ScratchJr.
  - Distance sensors: On the front and back of the robots, there are distance sensors. When an object is detected at a fixed distance (5 cm), strips can be started with triggers.

- **Motion**
  - Driving: Dash has blocks for moving forward, backward, and turning clockwise or counterclockwise. The blocks are visibly executing and wait to proceed for the entire duration of physical movement.

- **Looks**
  - Lights: All the RGB lights on a particular device can be updated with a choice from a menu. The devices have multiple controllable lights, but we have merged them into one block for most utility with least complexity.
  - Head position: Dash's head can be adjusted up or down and rotated.

- **Sound**
  - Preset sounds: Dash and Dot are preloaded with sounds; we offer a selection of blocks which will play sounds and wait until they complete.
5.2. Sphero and Ollie

Sphero and Ollie [18], created by Orbotix, are robots that drive around, change color, and do various tricks. Ollie, controlled with Bluetooth Smart, is high-speed and intended for racing and stunts, while Sphero, controlled with classic Bluetooth, is waterproof and rugged. Both can now be detected, connected to, and programmed in ScratchJr to drive and change color.

Our Sphero and Ollie extension attempts to make the programmable aspects of the devices obvious. For example, Sphero, by default, gives no physical indication of what direction it will drive in. In the ScratchJr extension, the "tail" light is always enabled to show what Sphero sees as "forward." We also implemented multi-part actions as single blocks: for example, Sphero does not physically drive backwards, so a block is included to first turn around, then drive, then turn back to the original direction. Finally, the blocks include wait time and reasonably observable defaults just as our Dash and Dot extension.
The main interesting sensor in these devices is the accelerometer; to take advantage of this in an intuitive way, we've included a "collision" trigger block that activates block strips when the devices rapidly decelerate.

Our current implemented prototype blocks include:

- **Triggers**
  - Collision: Sphero and Ollie have motion-based collision detection, so ScratchJr block strips can be started when the devices run into objects or have other impacts.

- **Motion**
  - Driving: Both Sphero and Ollie have blocks to drive forward, backward, or to turn clockwise and counterclockwise. In Ollie, the device can drive backward; in Sphero, the "backward" block is multi-part as described above.

- **Looks**
  - Lights: the RGB light inside the device can be changed to any of a selection of seven rainbow colors.
5.3. Light Play (and AdaFruit BlueFruit LE)

In addition to extending ScratchJr to toy store products, we are actively investigating and developing new technologies that can enhance young kids' playful exploration of the physical world and objects. One of these efforts is the Light Play project, a collaboration between our Lifelong Kindergarten group, Robbie Berg (Wellesley), and the Tinkering Studio at the Exploratorium [19], with inspirations from the preschools and kindergartens of Reggio Emilia, Italy.

The project explores kids' experimentation and tinkering with light, color, and shadows. In our current prototype, objects with interesting light properties (mirrors, lenses, slinkies, etc.) can be placed on a slow-moving turntable in front of high-powered lights and behind a translucent screen. Beautiful and interesting results appear on the screen.
We are using a ScratchJr extension to enable programmability of the direction and speed of the turntable, as well as the color and intensity of the lights. In addition, we are experimenting with basic physical-world feedback in the form of a light-sensitive "shadow sensor" and clips that measure electrical resistance in a circuit. The prototype hardware was built by Robbie Berg, and includes an Arduino shield for connecting to and communicating with the high-powered lights and motors. To connect to ScratchJr, the Arduino is wired to an AdaFruit Bluefruit LE breakout board [20]. This board is wired to the Arduino's UART TX and RX. A Bluetooth Smart characteristic is provided for each of these: writing to the TX characteristic gives serial input to the Arduino and the Arduino can send Bluetooth notifications to our ScratchJr extension using its serial RX.

The Light Play hardware is under development. Our prototype blocks include:

- **Triggers**
  - Digital sensor trigger: the board has two ports for wiring in sensors; when a threshold is crossed, the provided trigger block ("A" or "B") is activated.
Motion

- Turn turntable on or off: the prototype Light Play board has four ports; these can be turned on and off in blocks.
- Reverse turntable direction.

Looks

- Set color of rainbow light: menu block which offers a few color options to control an RGB LED.
- Fade rainbow light from color to other color: fades the color light over a few seconds, also showing the intermediate hues.
- Fade light in or out: slowly fades the intensity of connected lights.
- Turn light on or off.

5.4. Intel Edison (using Bleno)

During the prototyping phase of the Light Play project, we developed a ScratchJr extension to connect to the Intel Edison [21]. The Edison is a full computer, including a processor, memory, WiFi, a Bluetooth Smart radio, and complete compatibility with Linux. The first ScratchJr extension to communicate with the Edison used its WiFi connection; since ScratchJr extensions are implemented in JavaScript, they are fully

Figure 5-10: Official photograph of Intel Edison.
compatible with WebSockets with no additional infrastructure. In a second prototype, we
used Node.js and the Bleno [22] library to use the Edison as a Bluetooth Smart
peripheral. Both of these connections were functional, but we switched to the AdaFruit
board for cost and reliability reasons.

5.5. KIBO (extension created by Tufts DevTech Group)

KIBO [23], a programmable robot for children 4-7 years old ($229), is produced by
KinderLab Robotics and Marina Bers (a collaborator on ScratchJr). The current version
of KIBO is programmed by scanning wooden blocks (with barcodes). Mitchell Katz, a
student at the Tufts DevTech research group which works on both KIBO and ScratchJr,
created a KIBO extension for ScratchJr using our framework.

A UART Bluetooth Smart chip was added to the KIBO to allow for wireless streaming of
commands and inputs. Katz developed a custom set of ScratchJr blocks to program the
robot without the use of the wooden barcode blocks. This allows users to connect the
KIBO to virtual projects and to the other physical devices for which we have developed
extensions.

5.6. Proprietary prototypes

Several devices we are working with are forthcoming advanced prototypes created by
companies. These devices frequently have several sensors and actuators which can be
programmed in ScratchJr. For marketing and disclosure reasons, specific details of
several devices and our implementations cannot be released at the time of this thesis
writing.

ScratchJr extensions usually treat these devices as single characters even when
multiple sensors and actuators are attached. Sensors are represented with trigger blocks
which are activated at fixed thresholds. High-level blocks are provided to control the
actuators and outputs.

We hope to release more details about these ScratchJr extensions once the
products are made public.
6. Comparisons to other mobile physical-world programming

There are a few existing examples of using tablets and mobile devices to program wireless devices, and I will present them here for comparison with our project.

6.1. Tickle

![Tickle interface screenshot with Dash blocks.](image)

Figure 6-1: Screenshot of Tickle interface with Dash blocks.

Tickle [24] is an iPad-based programming language inspired by Scratch. Like our project, Tickle supports programming physical-world Bluetooth devices. As of writing, the supported devices include Dash and Dot, Sphero and Ollie, Jumping Sumo, Parrot MiniDrone, Arduino Bean, and Philips Hue, with more being added frequently.

The Tickle project was released in April 2015 and the concepts contained were developed independently of the ScratchJr physical connections presented in this project.
However, some of the design decisions made in Tickle are similar to those made in our project. For example, Tickle treats physical devices as sprites in the project (similar to how our project considers physical devices as characters).

Tickle is intended for older audiences, so its blocks use text and a large variety of arguments and options (similar to Scratch). ScratchJr's simplified blocks for beginners means ScratchJr may be less appropriate for detailed programming, although in many cases the functionality of Tickle and ScratchJr extensions is the same.

Tickle currently only supports one connection per type of device at a time. When disconnected from a device, Tickle also requires the user to explicitly re-connect (unlike the auto-discovery/connection mechanism for ScratchJr extensions we discussed in Section 4.4). Devices are chosen at the project screen: Tickle includes project "templates" and always shows compatible devices rather than presenting devices upon discovery.

Tickle blocks largely choose to present the user with as many programming options as possible (for example, in Figure 6-1 the move command for Dash includes parameters for direction, speed, and time) whereas our ScratchJr extensions tend to choose a more streamlined but still tinkerable default and allow users to configure the device upon further exploration. In this particular example, the ScratchJr blocks for moving Dash include individual blocks for each direction; the time can be adjusted by changing the (non-named) numerical parameter, and speed can be changed in a separate block.

There are also many differences in terms of the virtual environment Tickle provides. For example, while ScratchJr never throws runtime errors for buggy code,
Tickle does frequently; ScratchJr blocks can be run immediately on touch to encourage tinkerability, while Tickle blocks must be run on events. There are a variety of other differences, but Tickle is the most similar comparison to our project in spirit.

6.2. Dash and Dot Blockly

Dash and Dot come with their own Blockly-based programming environment.

Blockly [25] was created by Google, and is inspired by App Inventor for Android and Scratch. Blockly provides a JavaScript-based standard framework for creating block-based programming environments. Dash and Dot take advantage of this by providing a block-programming application to specifically control their robots.

![Figure 6-2: Screenshot of Dash and Dot Blockly app.](image)

The Dash and Dot Blockly app presents the user with many built-in "puzzles" for the user to solve. These encourage the user to explore the functionality of the blocks and
discover potential uses for Blockly. The user can also program with blocks intended specifically for Dash and Dot and which include visual explanations for the textual blocks and parameters.

ScratchJr’s interface attempts to represent some of the illustrated/text-based explanations in the Dash and Dot interface with icons only. ScratchJr also provides on-screen virtual characters that can interact with the robots, and other companies' devices. Our environment focuses more on the creation of multiple projects and creative exploration and tinkering rather than learning and exploring the robot capabilities.

6.3. Sphero and Ollie orbBasic, MicroLab, SPRK

![Figure 6-3: Screenshots of Orbotix MicroLab and orbBasic.](image-url)
orbBasic is intended to be an educational, tablet-based programming language for programming the Sphero and Ollie and is one of the suite of the educational programs for these devices [26]. The language is text-based and programs are compiled and delivered (via Bluetooth) to be run autonomously on the device. Our project controls the devices instead by a stream of commands, which adds the ability to make code execution visible. We also add interaction with unrelated devices and on-screen characters, and an environment more suitable for beginners.

The Orbotix MicroLab represents Sphero and Ollie code as text-based macros which are very explicitly mirroring commands that are sent to the robot (e.g., wait commands must be specified as macros). ScratchJr chooses higher-level macros that capture high-level actions rather than streamed robot commands (e.g., "drive forward" in ScratchJr will wait in the block until complete).

![Figure 6-4: Screenshot of Orbotix SPRK environment.](image)
Finally, Orbotix provides a drag-and-drop programming environment (released July 2015). This environment is mobile phone compatible and visual. Blocks are displayed in a single organized stack and are specific to the devices chosen. ScratchJr has a more flexible scripting layout that tries to emulate a (potentially messy) "workspace" or desktop; blocks can be arranged in ScratchJr in ways other than an organized stack. Like the MacroLab environment, SPRK blocks are very similar to commands that are streamed to the robot. Finally, like Tickle and Blockly, these blocks cannot be run by tapping them, making experimentation with and discovery of what blocks do slightly more difficult.
7. Future work

To extend the potential audience for ScratchJr extensions, one future direction is making the environment more extensible and flexible for advanced users. Another direction is creating an implementation for more platforms (Android, Windows Mobile, and so on). Yet another direction is supporting additional form factors (for example, creating a minified user interface for mobile phones in addition to tablets, or supporting a desktop-based version of ScratchJr). The tangibility of physical extensions could potentially be heightened by creating a framework for on-screen representation of physical objects, or in the opposite direction, usefulness could be increased by removing the ScratchJr stage and providing more space for scripting on screen. Suggestions and ideas for some of these future directions are included in this chapter.

7.1. Nearby device disambiguation and alternate connections

As described in Section 4.4, the current mechanism for connecting to devices automatically chooses a device based on signal strength (approximating distance). For advanced users, ScratchJr might provide an option to switch what device it is connected to at any given time (provided there are multiple compatible devices available). This would also maintain the benefits of automatic reconnection while providing advanced users the ability to connect to far-away devices instead.

7.2. Multi-platform support

The ScratchJr extensions system is readily portable to Android devices (since most of the implementation is JavaScript) except for the backend of the Bare-bones Bluetooth Smart
API (which on iOS is implemented with Apple's Core Bluetooth). Android provides an equivalent interface to the Bluetooth radio, so this is a matter of implementation.

Supporting further operating systems requires porting ScratchJr, which will also involve porting the rest of the native platform.

7.3. Mobile form factors

ScratchJr blocks are small and compact, yet they can be extended and composed to support powerful interactions with the physical world. One direction forward for this project is using ScratchJr-type programming blocks for physical devices on smaller screens.

Figure 7-1: Screenshot of Pocket Code application for mobile block-based programming.
Experiments with using Scratch-like programming on phone-sized devices exist: for example, Catrobat's Pocket Code and Orbotix SPRK. But a programming environment based on the ScratchJr design could enable more complex scripts (thanks to the smaller blocks) and the ScratchJr physical extensions could motivate programming on the phone, since programming graphics on the phone can be somewhat cumbersome. A future direction of this project might be exploring and implementing this idea.

### 7.4. Extension loading and sandboxing

![Extension loading screenshot](image)

*Figure 7-2: Screenshot of loading extensions on ScratchX from a file or URL.*

ScratchX extensions are not tightly integrated with the programming environment: new extensions can be dynamically loaded in the browser from files and external URLs, allowing extension developers to rapidly build extensions without needing to recompile the editing environment, for example.

ScratchJr extensions do not yet have this flexibility, but such a system could be added. An ideal way of doing so seems to be allowing developers to load JavaScript extensions from URLs (just like ScratchX). However, before this is possible, ScratchJr
extensions might need to implement some amount of JavaScript sandboxing. The current
tablet OS-JavaScript bridges would allow developers potentially dangerous access to the
system itself, so careful effort must be invested into ensuring developer-provided
extensions cannot maliciously destroy user data or modify the application. Perhaps one
way to quickly implement a sandbox would be to run extension code in a separate, hidden
UIWebview (iOS)/WebView (Android) with an extremely limited bridge to ScratchJr and
the system. A small select number of commands could be passed from the extension
sandbox to the ScratchJr view and back, allowing full JavaScript flexibility to the
developer but no ability to adversely harm the system.

7.5. Additional on-screen experiences and representations

One design goal of Scratch has been "making data concrete" by showing and displaying
the value of variables and otherwise abstract data structures. ScratchJr's physical
extensions are somewhat lacking in this regard: internal data structures and known
properties about the physical objects (like what direction a device is currently facing or
what color the light is) are almost never shown on this screen. In some ways, this is not a
barrier: sometimes the physical device is clearly manifesting the properties in the world
and so the data is already concrete. But in other cases, it can be confusing to program a
device without the data visibility. A good example of this during development was
Sphero's direction: without a concrete indication of what direction the Sphero was facing,
it was nearly impossible to do any kind of meaningful programming of its motion without
many trials. The Sphero happened to have a light inside of it to indicate its direction, but
we could also imagine a solution where the direction is displayed on screen. Physical
sensor readings may also be useful to display. When trigger blocks are used, in the current implementation it can be somewhat opaque when the blocks will execute. By displaying the sensor data either visually or textually, this data could be made visible.
9. References


[14] Core Bluetooth Overview. From the iOS Developer Library.
   https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts/CoreBluetoothOverview/CoreBluetoothOverview.html
https://developer.bluetooth.org/TechnologyOverview/Pages/GATT.aspx


https://www.makewonder.com/robots/dashanddot


[22] Bleno: a node.js module for implementing BLE peripherals. 
https://github.com/sandeepmistry/bleno


