Writing and Connecting IoT and Mobile Applications in MIT App Inventor

by

Kathryn Elizabeth Hendrickson

B.S., Massachusetts Institute of Technology (2017)

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

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2018

Certified by .....
Harold Abelson
Class of 1922 Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Writing and Connecting IoT and Mobile Applications in MIT App Inventor

by

Kathryn Elizabeth Hendrickson

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As the "Internet of Things" (IoT) grows and becomes more prevalent in society, it is important that everyone is able to understand and take advantage of IoT technology. I present the IoT Embedded Companion, a system integrated with MIT App Inventor that allows users to design and program IoT applications alongside a mobile app. This system uses the same block-based programming language as MIT App Inventor and includes live development features that allow users to see changes to their application in real-time while it runs on the mobile device and IoT device. The resulting projects consist of a mobile application and autonomous IoT program that together create the IoT application. Both the mobile app and the IoT program share global variables that either system can read and write, allowing the components to act together as a single application. In addition to writing the IoT Embedded Companion, I designed a curriculum for a workshop to teach and test the IoT Embedded Companion targeting middle-school aged students and held two iterations of the workshop. My findings indicate that students as young as middle school level are able to understand the concepts of IoT and that learning about it expands their knowledge of computing capabilities.

Thesis Supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering

Acknowledgments

First, I want to thank my advisor, Hal Abelson, for providing me this opportunity to work on and create a system that will be used to educate and empower millions around the world.

I want to thank the entire MIT App Inventor team for their knowledge and support, especially Evan Patton for his guidance throughout this last year and a half and for devoting so much time and patience to my questions and bug-hunting.

I want to thank Katherine Prutz for helping me out with testing and getting my thesis ready for use this last semester; Samantha Briasco-Stewart for sticking by me during the MEng and helping me with my thesis during crunch times; and Graeme Campbell for supporting me and feeding me when I didn't have time to eat.

Finally, I want to thank my family who has always believed in me and cheered me on and who I would not be here without.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The "Internet of Things" (IoT) is a system of small lightweight devices typically connected through Wifi or Bluetooth and is one of the largest growing computing platforms in the world [9]. IoT connects the physical and digital worlds: people can gather information about the physical world through sensors or manipulate it by turning on motors or lighting up LEDs. Even though the field of robotics involves both the physical and digital world, robots are usually independent machines that act alone to perform a single task while IoT devices are interconnected and synchronize to perform a variety of small tasks. Everyone should be able to learn about and take advantage of new technologies such as IoT instead of just consuming them. In order to empower everyone, including young people and beginners, my thesis aims to create a single, online system for creating IoT applications in a visual programming language.

In my thesis, I present a system to create autonomous IoT applications that fits within MIT App Inventor, an online platform for making mobile applications in a visual block language [15], and a workshop to explore how effectively we can teach IoT technology and empower students to create their own IoT applications in order to affect or learn about the physical world. This system, which is called the IoT Embedded Companion, is generic and extendable enough to work with many different IoT platforms, though in my thesis I focus on testing the Embedded Companion on the Arduino 101.

The following chapters contain each step of my thesis in detail. The rest of this chapter introduces some background. Chapter 2 outlines related work on existing IoT application platforms and what one must do currently to implement a fully functioning autonomous IoT application with a mobile app. Chapter 3 introduces the design and implementation of the IoT Embedded Companion including background on MIT App Inventor itself. Chapter 4 discusses a methodology to run a workshop surrounding IoT and MIT App Inventor to help teach middle school level students about IoT and how they can use IoT technology in their own lives. Chapter 5 presents the results of a workshop implemented with the aforementioned curriculum as well as an analysis and discussion of those results. Chapter 6 contains a brief conclusion about the effectiveness and implications of the IoT Embedded Companion and teaching methodology. Lastly, Chapter 7 introduces possible future work to continue to improve and extend the IoT Embedded Companion and teaching materials.

1.1 Alex's Smart Light IoT Application

Let's meet Alex. Alex is a seventh grader who wants to have a smart light in her bedroom. She wants to be able to turn the light on or off automatically based on light levels in her room, in response to a physical button press, or remotely from any location. For example, if she sees her light on through her window as she's leaving the house, she wants to be able to turn the light off from her smartphone. Looking online, Alex finds a lot of mass-market IoT devices that claim to be smart lights, such as Philips Hue, which integrate with Alexa or Google Home. Even though Alex would have to rule out this option based on price alone, regardless she doesn't know how to customize the Philips Hue to do everything she wants [17].

Since Alex just learned how to write computer programs in Scratch, a visual block programming language [19], she decides to use this new knowledge to make her own smart light. Now attempting to find a do-it-yourself route, Alex looks up a variety of ways to make a smart light and, in the process, learns about Arduino, a type of inexpensive microcontroller that you can attach a light sensor, a button, and a

light bulb to [2]. She continues researching how to use an Arduino and discovers that you can program one only in C++, a programming language that she doesn't know. Eventually she finds some visual languages similar to Scratch which allow her to program the Arduino using blocks. It takes awhile, but she figures out how to compile one of these languages, Snap4Arduino, so that she can always have her program running on the Arduino, even when it's not connected to the computer [20]. Now her Arduino program turns the light on if the light sensor senses that it is too dark out and turns the light off when it is too light out. This works out until Alex realizes that she wants to be able to control the light from outside if she forgets to turn the light off before leaving the house.

Alex would like to control her light from anywhere in her room, not just from one specific location, so she decides that the best way to make sure that she can control the light from anywhere is by making a mobile application (app) which can turn the light on or off. Looking online for an easy way to create a mobile application, Alex finds MIT App Inventor and spends some time creating her mobile app and realizes that the app needs to communicate with her Arduino. She knows that her Arduino has Bluetooth capabilities, so she looks around until she finds App Inventor's Bluetooth components. Alex wants her mobile app to send her Arduino a Bluetooth message when she presses the button on her phone, but figuring out how to correctly establish and carry out this communication takes a really long time. She has to have App Inventor and Snap4Arduino open to modify her projects, have her Arduino connected to her computer, and have her phone ready to test. Whenever she makes a change in one system she has to remember to make a matching change in the other, and juggling the two systems becomes increasingly difficult as her project becomes more complex.

Alex's experience in creating an IoT application, her smart light, is not a unique one. Currently, ready-to-use IoT devices are expensive and making them out of relatively cheap components requires either learning new skills (e.g. C++) or juggling multiple systems (Snap4Arduino and MIT App Inventor). Therefore, in order to allow anyone to own and control IoT devices, App Inventor is incorporating IoT devices as

another platform on which users can easily develop applications.

App Inventor has recently created a Bluetooth Low Energy (BLE) component that allows mobile applications to connect to and interface with various BLE devices [12]. The greatest limitation of the current approach is that the IoT device is controlled by the app on the mobile device thus the IoT device must be within Bluetooth range in order for it to work. This means that Alex's smart light would not be able to turn itself on when it gets dark out if Alex's phone was not connected to her Arduino.

1.2 MIT App Inventor

MIT App Inventor is an online platform that anyone can use to create mobile applications through a visual drag-and-drop interface. This provides a high level abstraction that allows users to express what they want their app to look and act like without prior knowledge of traditional text-based programming languages and design aspects. This abstraction provides a lower barrier to making mobile applications, allowing anyone to take advantage of the technology provided to them by their mobile devices. By providing an accessible platform, App Inventor empowers and encourages people to think about how they can solve problems through the development of their own applications. Currently, MIT App Inventor has 6.8 million users from over 190 countries where over 1.1 million unique users are active monthly. People all over the world use App Inventor to solve real-world problems and have created over 24 million apps [15].

The development interface on App Inventor is split into two parts: the designer and the blocks editor. The designer shows the user the layout of the app, the components that it consists of, and the options available to customize component properties (Figure 1.2).

The blocks editor (Figure 1.2) provides a drag-and-drop interface to connect puzzle-like blocks that represent actions to create programs. This block language is built on Google Blockly [4] and is similar to Scratch [19]. The blocks replace traditional text-based programming languages and are colored to highlight the different

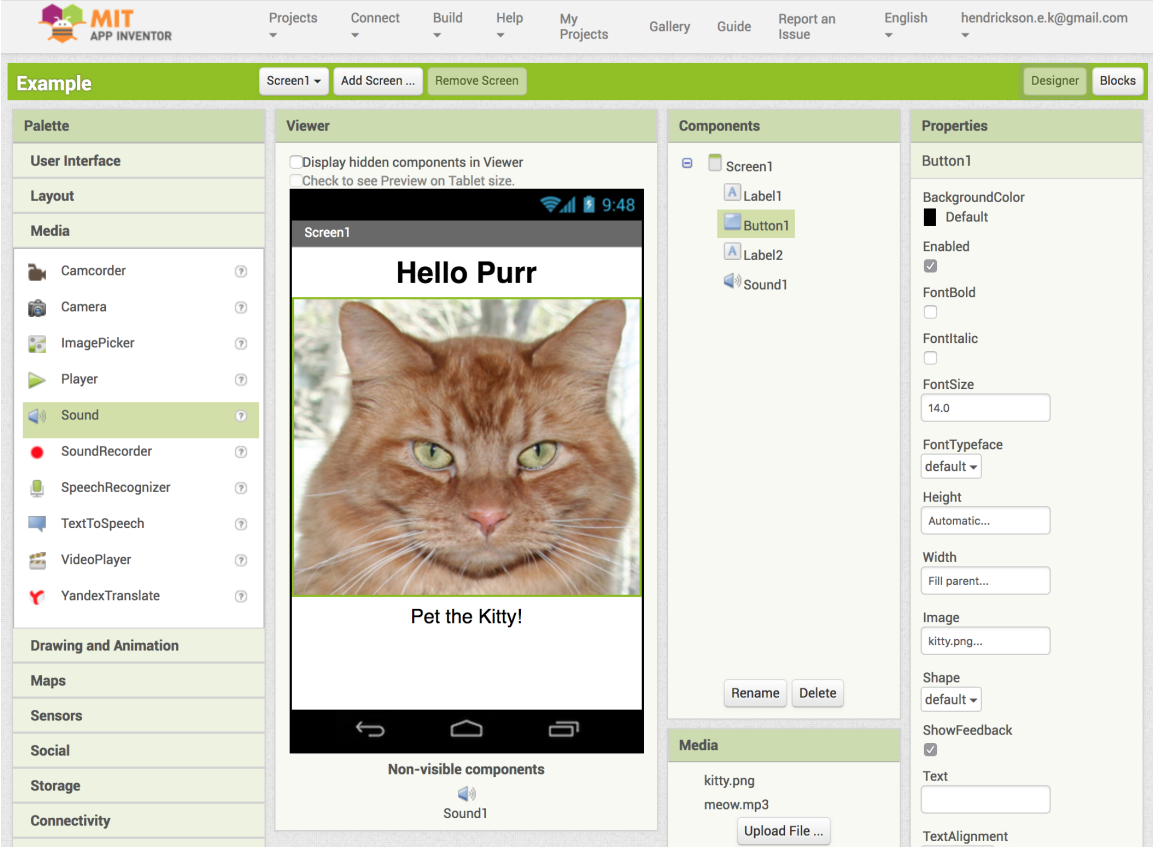


Figure 1-1: The designer view of MIT App Inventor. New components are found in the Palette (left) dragged onto the Viewer (center left). The components currently in the app are listed under Components (center right) and their properties can be edited in the Properties section (right).

types of logic or actions that they represent. This allows users to visually see their program laid out and organized in a manner that makes it easier to understand what their app is doing.

MIT App Inventor also provides a mobile application, the App Inventor Companion (or just Companion), that allows users to test their applications as they create them. The real-time feedback of the app allows users to see the exact consequence of the component they just added to the designer, the property they just changed, or the blocks that they just added. This feedback helps the user understand what is happening in their mobile app and allows the user to iterate over many versions of their app as they test and change it to reflect exactly what they desire. This creates a very interactive development experience which is extremely useful as a beginner.

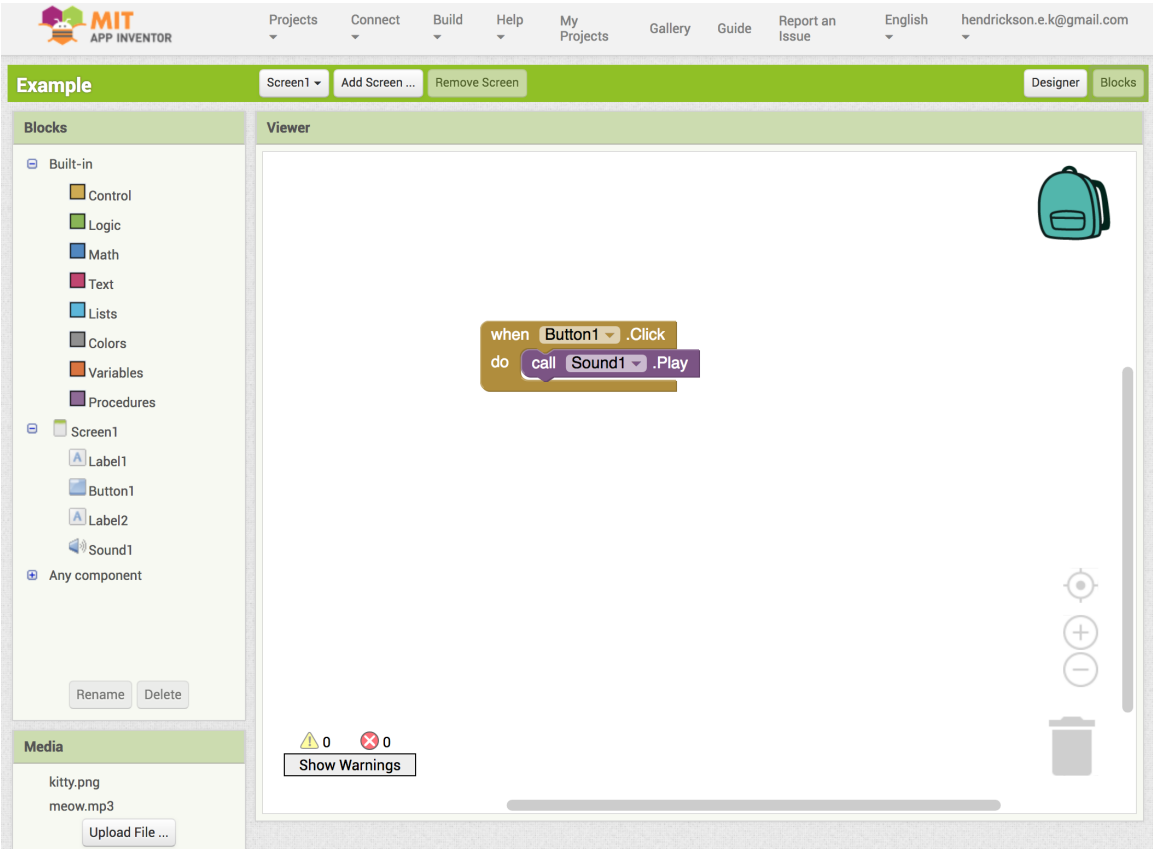


Figure 1-2: The blocks editor view of MIT App Inventor. Blocks appear based on what components are in the app in Blocks (left).

Chapter 2

Related Work

2.1 Arduino

Arduino is a platform that provides both physical hardware (Arduino boards) and software (the Arduino libraries, IDE, and web editor) for creating custom IoT programs [2]. Arduino provides abstractions for the physical and electrical components in the Arduino programming language, which exposes easy-to-use functions, such as `digitalRead` or `digitalWrite`, and associated constants such as `HIGH` or `LOW` constants. Advantages of using Arduino include both these abstractions and the ability to program many different types of Arduino devices using the same interface. However, knowledge of Arduino only allows users to program Arduino devices (e.g. it is no help programming a Raspberry Pi) and users also need to know how to write traditional, text-based programs.

If Alex created her Arduino program in the online Arduino web editor [1], she would first have to install the Arduino Plugin onto her computer in order to actually upload the Arduino sketch to her board. Then she would open a new sketch which would have empty methods for `setup()` and `loop()`. Alex would have to figure out how to describe what is connected to her Arduino in code as well as translate ideas such as "turn on the light if it is dark." Then she would press upload and the program would be downloaded onto her board, unless of course, there was a compilation error which she would have to fix first. When everything appears to work fine, she would

put the Arduino running the program onto her desk and walk away. Later, she might realize that she wants the light to turn on when her room is darker so she would have to get her computer out, plug her Arduino back in, and repeat the process.

2.2 Snap4Arduino

Snap4Arduino extends the Arduino platform to use a visual programming language similar to App Inventor's blocks, but includes more advanced programming paradigms such as first class procedures and continuations [20]. The visual programming language that Snap4Arduino uses is called Snap! which is an "extended reimplementation" of Scratch [20]. These advanced features imply that Snap4Arduino targets an audience with more technical background knowledge in programming than App Inventor. Snap4Arduino aims to provide a dynamic programming environment so that once users load the required firmware onto their Arduino they can create interactive programs that use both the Arduino (or multiple Arduinos) and the computer at the same time. The two platforms are intertwined and the goal is to create a live dynamic environment, not an isolated Arduino program. However, Snap4Arduino also provides a compiler which turns specific 'translatable projects' into Arduino sketches that users can then manually load onto their devices [21].

If Alex were to create her smart light with Snap4Arduino, she would first have to download the Arduino IDE, connect her Arduino to her computer, and upload the required firmware onto her device. Then she would open up Snap4Arduino to see familiar-looking programming blocks. She would build her program in blocks and hit run to test it on her connected Arduino. When satisfied, she would export her program as an Arduino sketch which she would then have to open in the Arduino IDE and upload to her device. Everytime she might want to make a change, she would have to fix it in Snap4Arduino, redownload it into an Arduino sketch, and upload it to her plugged in Arduino.

2.3 BlocklyDuino

BlocklyDuino is a platform, similar to Snap4Arduino, that uses a visual block language, Blockly, to program Arduino devices [10]. BlocklyDuino focuses solely on Arduino development and therefore can always create an Arduino sketch for a block program, unlike Snap4Arduino which requires the user to switch into a different mode in order to generate the Arduino code. Instead of having to download an Arduino sketch and using the Arduino IDE to upload it to the user's Arduino, BlocklyDuino provides a 'mini webserver' that does all of that for you. Even though this makes adjusting the permanent code on the Arduino easier, BlocklyDuino does not have live development so the user must remember to click the 'upload' button and wait for the code to be transferred to their Arduino before testing.

In addition to providing a platform with which users can visually program Arduinos, BlocklyDuino provides abstractions for a set of Arduino Grove devices and sensors. These abstractions remove the need for the user to understand what set up is needed for a given device and what they need to know about it. Also, BlocklyDuino allows users to create their own blocks which they can then incorporate into their programs. This gives users more flexibility in the scope of potential programs that they can create.

2.4 Raspberry Pi

A Raspberry Pi is a small, inexpensive, full-fledged computer made by the Raspberry Pi Foundation [18]. Since the Raspberry Pi is small and inexpensive and a computer that can do everything a microcontroller can do, the Raspberry Pi makes a good IoT platform. Since it is a computer, there are many languages and programs with which you can use to create an IoT application and you have a vast array of hardware components that are compatible with it. However, sometimes having extra options is not the best for a beginner and can be overwhelming. In order to use a Raspberry Pi, Alex will have to make sure she has the right set of cables and devices to just

connect to and run her Raspberry Pi (e.g. monitor, keyboard, mouse, SD card) and then she has to set it up with the set of software she needs to run her smart light program (e.g. an operating system, the language she decides to program in, libraries that will connect to her light, light sensor, and button). All of this needs to happen before Alex even starts creating her smart light. This creates a high barrier for Alex, a hurdle she is unlikely to clear if she is just trying to make her small smart light project.

2.5 BlockyTalky

BlockyTalky is a programming environment that uses a block language to create IoT programs for a Raspberry Pi [5]. The blocks in BlockyTalky are designed to follow a beginner's way of thinking about how the Raspberry Pi is working. BlockyTalky requires all the same setup as a Raspberry Pi, but if Alex uses BlockyTalky she knows exactly what she will need to install and what hardware to buy that will work with BlockyTalky. BlockyTalky is meant to be installed, developed, and run on the Raspberry Pi so after it is set up no more code transfer is necessary. The user can simply open BlockyTalky and create programs.

2.6 BBC micro:bit

The BBC micro:bit is the closest existing platform to the desired design of App Inventor's IoT system. The micro:bit is a hardware device like an Arduino that has BLE capabilities, built in devices, such as an accelerometer, and pins that users can connect to additional components [14]. Users can use the Javascript Blocks Editor (PXT) to program their micro:bit using a visual block language similar to App Inventor's and then load that code onto their micro:bit to run. The micro:bit also includes a mobile app that allows users to write and send code to their micro:bit over a BluetoothLE connection. This allows users to easily change the code on their micro:bit without having to deal with compilation or physically loading code onto

their device. This overall design is very similar to what the App Inventor system aims to provide for users: the ability to program an autonomous IoT device from the computer or a mobile device and the ability to connect and interact with it through a mobile app. A disadvantage to using the micro:bit is that it is limited to a single platform and is not transferable to other platforms such as Arduino or Raspberry Pi.

2.7 GraspIO

GraspIO (Graphical Smart Program for Inputs and Outputs) provides hardware and software for developing electronics projects [8]. GraspIO makes "Cloudio" which is a hardware shield for a Raspberry Pi that comes with additional sensors that, like the micro:bit, provides the basics for sensing and output but also provides ports to add additional components. GraspIO also creates a mobile app "GraspIO Studio" that provides a drag-and-drop programming interface for users to use to create programs which communicates through the cloud to the GraspIO shield. The programming blocks are mostly color- and picture-based instead of text-based and provide high level abstractions with which to program for easy and quick development.

Since Cloudio is built on the cloud, the innate capabilities are greater than Arduino, which assumes an 'offline' presence. One of the features of Cloudio is that it is connected to IFTTT (If This Then That) which is a platform that connects many different programs and systems on the internet and in IoT [11]. Because Cloudio is connected to IFTTT, Cloudio can respond to communication from anything that works with IFTTT such as Gmail or Alexa. This greatly expands the reach of this IoT platform without that much added effort by the user. However, since GraspIO Studio is only an app to program Cloudio that can also display sensor output, it does not really have a way to integrate it with a personal, customizable mobile application. GraspIO is trying to be an IoT device that can be used in conjunction with a bunch of already existing IoT applications where the hardware is the most customizable part.

2.8 Node-RED

Node-RED is a platform that allows users to connect various hardware and software components into a single system [16]. The platform is built on Node.js and is intended to be used on hardware such as a Raspberry Pi, though it is not restricted to any such device and can run on other devices such as an Arduino. The main concept in Node-RED is called a *flow* which are used together to create networks that describe a program. A *flow* is a series of components (*nodes*) that are connected by *wires* which define data paths. Node-RED requires users to think of their programs in terms of data flows, an abstraction which may not be ideal for beginners as you need to know exactly what you want to do before you start which is not great for experimentation. However, Node-RED also provides excellent APIs which allow users to access online data sources such as Twitter or Weather Underground easily. These APIs allow a user to create a far-reaching IoT application similar to what one would be able to create with GraspIO but with more customization.

2.9 MIT App Inventor + IoT

MIT App Inventor recently introduced IoT into the repertoire of things that a user can add to their project. In order to use it, a user must use an Arduino 101 or micro:bit as their IoT device platform and they must download and use specific component extensions specific to the Arduino 101 or micro:bit. Among the IoT extensions is a BluetoothLE component that allows their mobile app to connect to and communicate with their IoT device. After the setup with the BLE component is complete, the user can create an IoT application with the same process with which they would use to create a normal App Inventor app by creating the layout in the designer and the logic in the blocks editor [12].

In order to interact with the IoT device, there are non-visible components for each of the supported sensors and devices that you add to the app. Once they are added, they can be used just like other components in the logic blocks. For example,

if Alex presses a button on the app, it could send a BLE message to the IoT device that says turn the LED on. However, because the IoT components are part of the mobile app logic, it is not possible to run anything on the IoT device autonomously. Every interaction is through BLE through the phone so the IoT device becomes just an extension of the phone while the mobile app is running. If Alex wanted to use App Inventor's IoT, she would have to have her phone connected in order to press the physical button that turns her light on and off. In order to make a full-fledged IoT project Alex wants to be able to autonomously run programs on the IoT device.

2.10 Summary

Alex could use any of these systems to create a part of her smart light project. In all but App Inventor's current IoT, Alex can make a standalone app that lets her control the light through a physical button and have it automatically turn on and off based on the amount of light in her room. In order to add the ability to turn her smart light on or off remotely, Alex wants to make a custom mobile application.

If she were using GraspIO, the mobile application that Alex used to write her original program is the only option for her to use to control her smart light remotely. If she were using micro:bit, Arduino, or Raspberry Pi, Alex could create a mobile application in MIT App Inventor and connect it to her device. To connect to the micro:bit, Alex would need to simply use the new IoT features in App Inventor. To connect to her Arduino or Raspberry Pi, she would need to orchestrate communication through App Inventor's Web component that can send HTTP requests or Bluetooth.

Using multiple platforms requires Alex to do extra work in figuring out how to orchestrate the communication back and forth between her devices. This causes hassle because Alex must update both systems if she makes any changes that has to do with both of the systems and problems can arise if the systems get out of sync. Making changes and making sure they are in sync add up to make a slow development cycle and makes it hard to see the impact of changes as they are made.

Chapter 3

System Design and Implementation

3.1 Introduction to the MIT App Inventor System

As discussed in Section 1.2, MIT App Inventor is an online platform for making mobile applications that includes a mobile app called the App Inventor Companion (or Companion), a platform for live test development. If Alex wants to use the Companion for live development, she would download the Companion app onto her smartphone, use the connect feature online and on her phone to start communication, and then see her app appear on her phone along with any changes she that she might make in real time. The Companion app contains all of App Inventor's components and logic to emulate Alex's project on the mobile device.

The App Inventor website communicates with the Companion by making requests to an HTTP server running within the Companion. When Alex makes a change in her project, the updated state is sent from her browser to the Companion. If, for example, she changed the font of the title to bold, a message would be sent over that indicated this and the Companion would find the title component the change is associated with and set the font to bold. The same process would happen if Alex added a play sound action when she presses a button on her app. The logic in the blocks would be turned into code that is sent over as a change to the button component.

All of Alex's app is made up of components. These components are defined by what they add to the app, whether they are visible (such as text for the title) or

non-visible (such as sound). Each component has properties that Alex can change in the designer such as bold font or the source of the sound as well as associated logic in the blocks editor, such as what should be done if the button is pressed or what actions combine to play a sound. This grouping of properties and logic into a component is an abstraction provided by App Inventor to aid Alex in the creation of her app. Instead of worrying about needing to check if the button is pressed, she just has to provide logic that App Inventor will run whenever the button is pressed.

Since Alex's app revolves around the components that she adds so does the logic. Every piece of Alex's block code is executed as a part of a component handler. The blocks that Alex pieced together are turned into code for the Companion app (and later the actual apk of her app) by a custom App Inventor generator. This generator essentially compiles the blocks into code that is sent over to the Companion when Alex makes changes.

Having the Companion app means that Alex can test her app in real time while developing it. This is important because it means that it is not necessary for her to redownload and install the app on her phone every time she makes a change. She only needs to install the Companion once and wait a short moment during the development process before she can see the changes she made and decide if she likes them or wants to make more changes.

3.2 Overview of the IoT Embedded Companion

When creating the mobile app in MIT App Inventor, Alex uses the live development feature in order to add features iteratively and she felt so productive using it that she would like to be able to use live development when creating the IoT program. My thesis introduces a prototype of the IoT Embedded Companion (or Embedded Companion) which serves a similar purpose as the Companion app, but for an IoT platform instead of a mobile device. If Alex is using the Embedded Companion then she can make changes to her IoT application and see the changes she makes in real time on her IoT device. The Embedded Companion also provides a similar

abstraction of components, so that Alex can focus on describing what she wants her IoT application to do instead of trying to figure out how to know when something has happened. For example, the button handler block for an Android button (in Figure 3-3) and for an Arduino Grove Button (in Figure 3-2) carry out the same action.

When Alex opens a new project in App Inventor, she will have the ability to connect an IoT device to her app. This adds an IoT device screen to her project, to which she can add the variety of sensors and electrical components that she needs to create her smart light (Figure 3-1). Then, in the blocks view, she can program her IoT device to turn on when the light sensor has sensed that the room is too dark or to correspond to a variable that was set by the mobile application (Figure 3-2). If she switches to the mobile app blocks view, Alex can add logic so that when she presses the on button in the app, it will send a signal to the IoT device to change the same variable she used earlier in the IoT blocks view to have the value 'on' (Figure 3-3).

Once Alex is ready to test her new app, all she has to do is upload the MIT App Inventor IoT Embedded Companion code to her target device, connect the Companion app to her project, and connect her phone to her IoT device via BluetoothLE. Then, when Alex changes the LED to always blink, for example, the LED on her IoT device will start blinking without any further work from Alex. If Alex wanted LEDs on many different IoT devices to blink, all she has to do is make sure that the Embedded Companion is on the device and connect to it through her phone and the LED will start blinking! Her original device is also still blinking because by connecting the Embedded Companion to the app (whether via the Companion app or the final downloaded apk), the program is permanently stored onto the IoT device and will run until Alex makes changes to it or powers the device down.

For my thesis, I designed the IoT Embedded Companion and implemented a proof-of-concept version that works for Arduino devices. However, the Embedded Companion itself is general enough to use on any IoT device that can run a C program. The IoT Embedded Companion was modeled after the Companion app that is currently provided by App Inventor so it contains an interpreter for the code that Alex wants to run on the IoT device so the website can just communicate these changes by

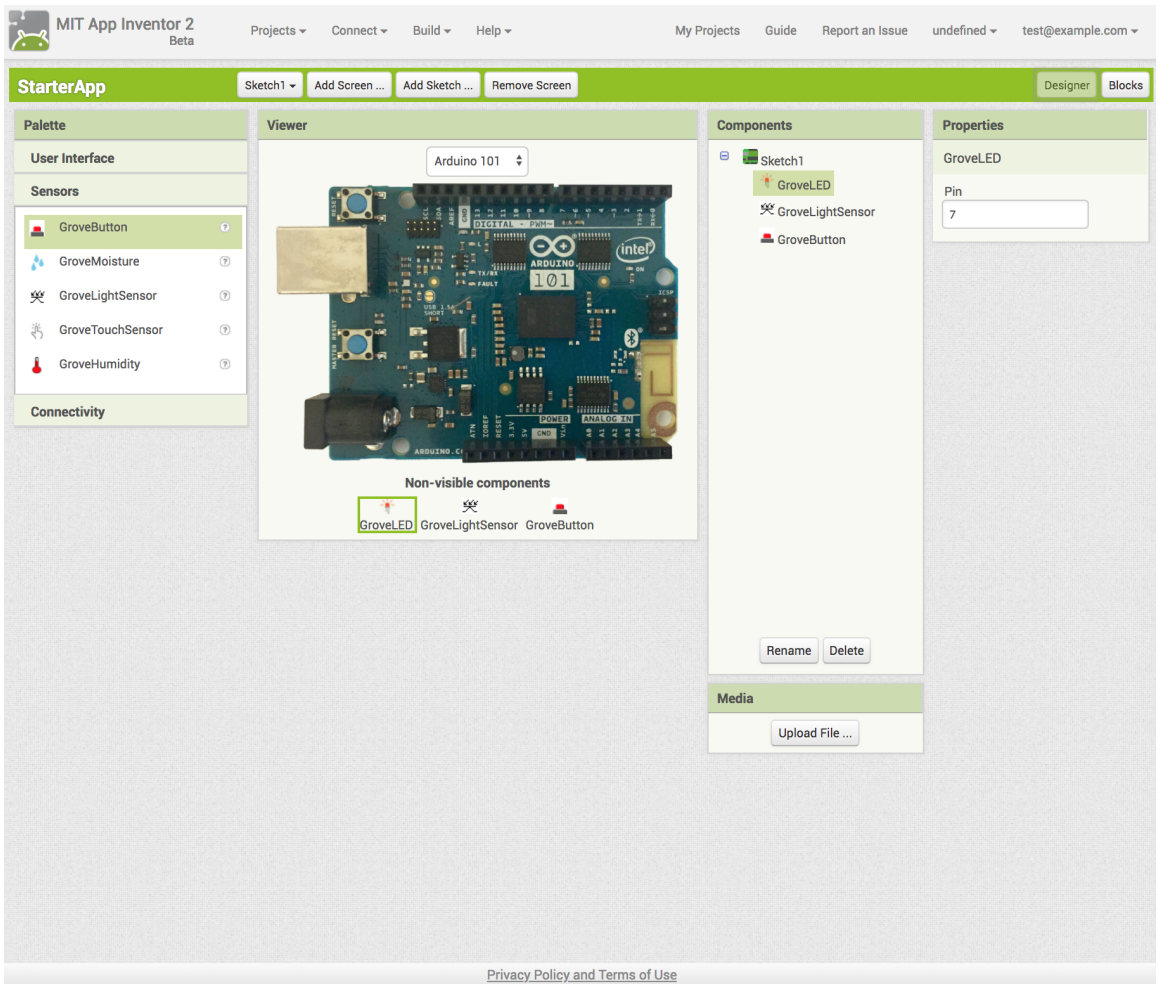


Figure 3-1: MIT App Inventor IoT Embedded Companion: IoT Sketch Designer

sending over new code to interpret. In order to integrate the Embedded Companion into App Inventor I created a specification for an IoT bytecode, added a generator to the website that turns blocks into IoT bytecode, and modified App Inventor to communicate new changes to the Embedded Companion.

3.3 The IoT Embedded Companion & Interpreter

The App Inventor IoT Embedded Companion is the software that runs on every App Inventor connected IoT device. The Embedded Companion has (1) a bytecode interpreter that runs user code on the device autonomously, (2) definitions for how to interface with available devices, and (3) an interface that communicates with the

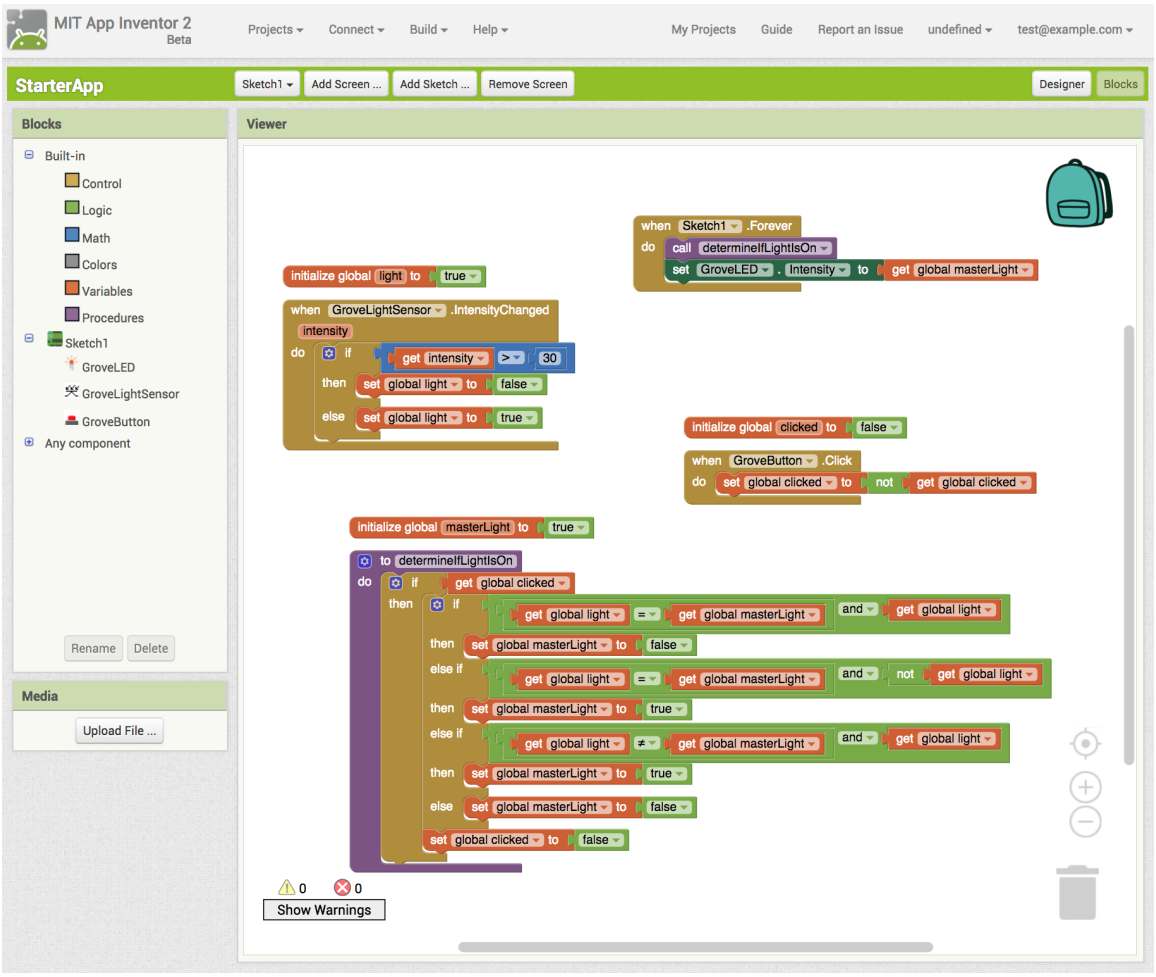


Figure 3-2: MIT App Inventor IoT Embedded Companion: IoT Blocks Editor

Companion app. Since only the user programs can change and they are not part of the Embedded Companion, users can update their program dynamically (through the Companion app), which makes experimentation with their IoT device easy.

The main components of the Embedded Companion are device information, Bluetooth communication, and interpreter code. Bluetooth communication sets up the system through which the IoT device can talk to either the Companion app or Alex's final app. Device information includes the declaration, definition, and implementation of device components (an example can be found in Appendix G). The devices Alex would need for her smart light would be a light sensor, button, and light (most likely an LED). In the blocks editor, Alex would create a device event handler to turn the LEDs on or off when she presses the button. The website would notice the

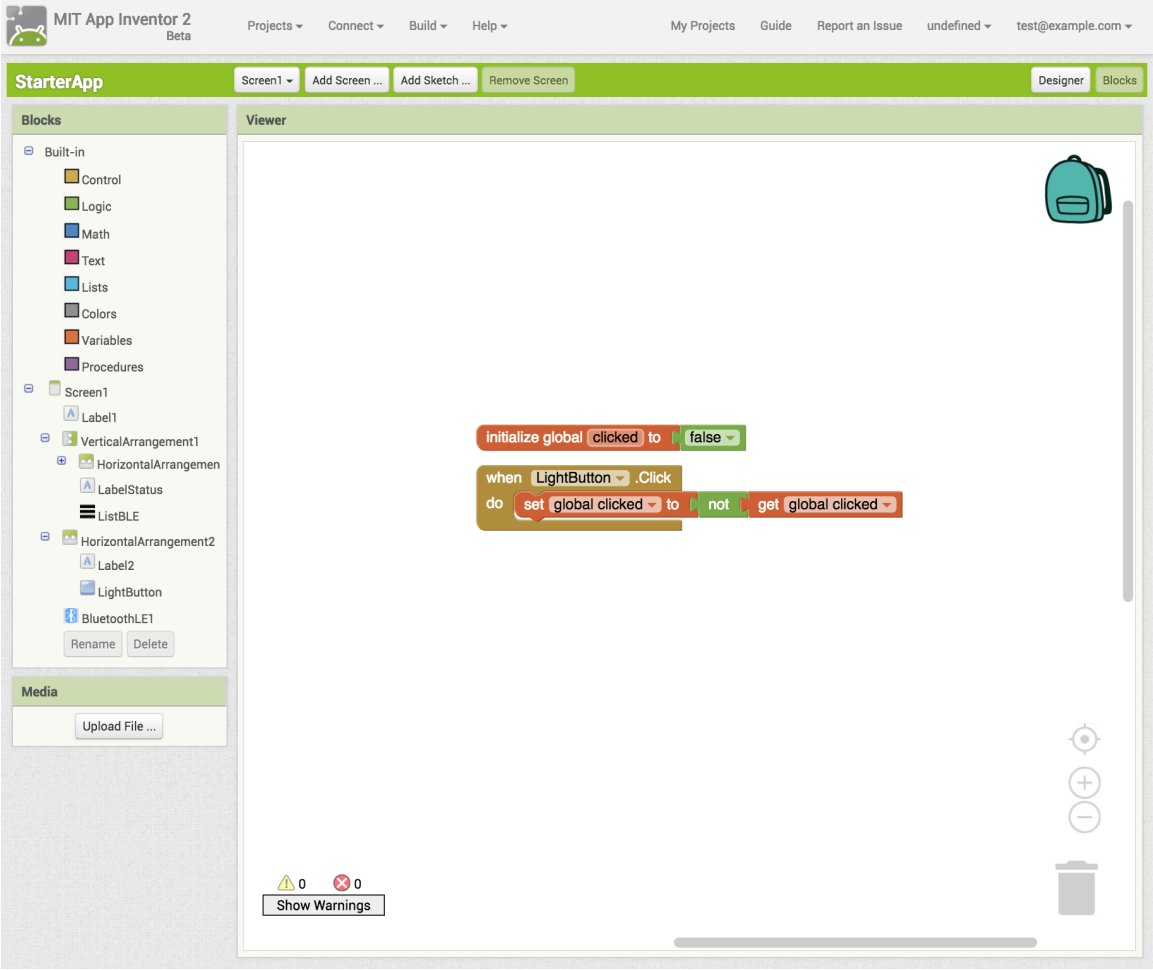


Figure 3-3: MIT App Inventor IoT Embedded Companion: Screen Designer

change in Alex's IoT program and package a message to send to the phone, which would forward it on to the IoT device to change Alex's program content. On the IoT controller, the interpreter would run the new device handlers and main program autonomously.

3.3.1 Previous Work

Evan Patton, a software engineer at MIT App Inventor, set up the initial architecture for the IoT Embedded Companion and how it should fit in with the rest of App Inventor. This skeleton consisted of an organization of the Embedded Companion and method headers for essential methods. Specifically, the Embedded Companion was split into five parts: documentation, examples, include files for the library, source

code for the project, and tests. The project code itself was divided into the following four categories: device, interpreter, platform, and util.

The device code consists of declaration, definition, and implementation code for devices that Alex can connect to her hardware device, such as an accelerometer, buttons, LEDs, etc. Each of these device definitions is specific to a hardware platform and device (Section 3.3.5). The interpreter code consists of data structures and functions that make up the core of the interpreter including the layout of program bytecode and the definition of opcodes. The platform code contains information on each target IoT platform such as the amount of available memory and the size of an integer. The util code provides various platform-independent utility code that does not belong in the previous categories.

3.3.2 IoT Bytecode

The IoT bytecode that runs on the Embedded Companion was originally based on the Java bytecode to opcode (operation code) mapping for the instruction set that runs on the JVM (Java Virtual Machine) [6]. However, over time we condensed this bytecode to be smaller and more focused on the instructions that are essential to running an IoT program based on the blocks provided by App Inventor. A list of opcodes and descriptions can be found in Appendix F.

In total, there are only 31 opcodes in the Embedded Companion. Each opcode is represented by a single byte and any additional static information that is needed is represented by an extra byte in the program. Dynamic information is available on the stack. An example of an opcode with both static and dynamic information is `store_global` which takes an extra byte to represent which global variable to store into and a dynamic value on the stack as the value to store to that global variable.

The reason we need to keep the number of opcodes to a minimum is that the mapping between bytecode and opcode definition is stored in memory. Some IoT devices have very little memory, e.g. the Arduino Uno has only 2KB of SRAM, and since we want the Embedded Companion to work on as many devices as possible we want to limit the amount of memory it uses as much as possible.

Field	Size	Purpose
`n_user_procs`	`uint8_t`	The number of user-defined procedures specifies the length of the `proc_table`.
`n_strings`	`uint8_t`	The number of strings specifies the length of the `strlen_table`.
`n_persistent_globals`	`uint8_t`	The number of global variables specifies the length of the `global_table`.
`n_devices`	`uint8_t`	The number of devices specifies the length of the `device_table`.
`sz_bytecode`	`uint16_t`	The number of bytes in the program, starting at the first byte of initialize (byte 0) and ending after all the code for initialize, forever, user-defined procedures, and event handlers.
`forever_address`	`uint16_t`	The index of the byte (address) that the forever code starts at.

Table 3.1: IoT Bytecode Program Header Structure

3.3.3 Program Structure & Memory

Alex's program is completely represented by bytes, but is more than just opcodes. At the beginning of the program, there is information about the program in what is called the program header. The program header tells the interpreter how long the program is, how many procedures there are and what address they start at, how many global variables and devices there are, and finally where the setup code and the forever loop code begin (Table 3.1). The forever loop contains the code that will be continuously run on the IoT device. All of this information is found in bytes in the program header, which is included in the program when transferred over from Alex's project (Table 3.2).

In addition to the program header and the program itself (the various instructions that make up the setup, forever, and procedure sections), the program structure holds other data types that are essential to the execution of the program. These include tables for strings, global variables, and device information as well as space for the heap and the stack. The program structure has a fixed size that memory is allocated for, and each piece expands as necessary (e.g. if there are no global variables the global variable table is of size 0). After the program header, the instructions themselves,

Byte	Field	Description
0x01	`n_user_procs`	1 user-defined procedure
0x00	`n_strings`	0 strings
0x03	`n_persistent_globals`	3 global variables
0x03	`n_devices`	3 devices (LED, Button, & LightSensor)
0x91 0x00	`sz_bytecode`	There are 0x91 = 145 bytes in the program.
0x89 0x00	`forever_address`	The forever loop code starts at address 0x89.
0x2c 0x00	`proc_table`	The procedure `blink` starts at address 0x2c and has no arguments (the top 3 bits are 0).
0x04	`start`	Byte 0 of the program (the beginning of the initialize code).
...	...	...

Table 3.2: Example IoT Bytecode Program with Explanations

and tables, there is a lot of memory left over in the program structure. The top of this memory (right after the tables) is where the heap starts. The heap is where objects are instantiated and where their data resides. The only thing that the Embedded Companion uses the heap for at the moment is to store the properties of each device (device information can be found in Section 3.3.5).

At the very bottom of the piece of memory allocated for the program structure, growing upwards (toward the bottom of the heap), is where the stack is located. The stack is used for every dynamic value used in operations whether the operation is subtracting two numbers (both of which are located on the stack) or branching to a new location. The beginning of any program execution starts with an empty stack. Every procedure call has its own stack and in order to keep track of the information about each procedure call's stack, the stack frame includes the location of the previous stack frame in the stack, the location of the current stack pointer, the current program counter, and the values of any local variables.

3.3.4 Interpreter

The heart of the interpreter is a function called `interpreter_run` that takes a starting frame and runs the function based on the given frame. The starting frame specifies

the location of the first instruction as well as the current status of the stack. The function picks out the opcode found at the first instruction, looks it up in the opcode table, and runs it. The opcode definitions increment the program counter and the stack pointer as necessary, so after an opcode is done running all `interpreter_run` needs to do next is repeat the process and pick out the opcode of the new instruction pointed to by the program counter.

In this sense, all program execution starts in an opcode (even `device_setter`). However, in order to maintain consistency and keep the definitions of opcodes simple and easy-to-understand, the interpreter provides abstractions for working with the stack. The interpreter provides methods for pushing and popping a variety of values to and from the stack such as bytes, longs, floats, or even a custom type called a `value_t` which can be either a 22-bit integer or 32-bit float. The interpreter also provides methods for setting up new stack frames or returning to the last one.

Lastly, the other primary job that the interpreter performs is run each device's duty cycle. A duty cycle is a function that is run every time step that checks a device's properties and triggers event handlers as necessary. When she wrote her program, Alex pieced together blocks that described what her IoT controller should do if, for example, the physical button was pressed. Since the interpreter is always running, there is no way for the button device to tell the code to stop and run something else because the button was pressed. Instead, what happens is the interpreter will check the status of each device every so often and if something triggered one of Alex's event handlers, it will then dispatch the event that happened and tell the interpreter to run Alex's handler code. This checking is done by a device's duty cycle. The method `interpreter_duty_cycle` iterates through each of the devices, runs the duty cycle, and calls `interpreter_dispatch_event` if one of the events was triggered with all the requisite information to run the corresponding event handler. Once every device's duty cycle has been run, the interpreter gathers all of the dispatched events and runs each respective handler. Also included in the `interpreter_duty_cycle` is the code that Alex included in the `forever` loop of the IoT controller. (Note that the `initiliaze` code is only run once and therefore is not included in the

`interpreter_duty_cycle`.)

3.3.5 Devices

Every device definition in the Embedded Companion defines a set of properties, known as the device data, that corresponds to the information that Alex can see on the website. These properties include physical properties like what pin the device is connected to as well as current sensor readings and the address of an event handler. Every created device of a certain type (e.g. button or LED) has its own device data; this is the data that is stored in the heap for every device. The rest of the device definition in the Embedded Companion is static and shared for every device of that type. This would include the property getters and setters and the device's duty cycle (Appendix G). Each of these functions act upon a device's specific data, which is why the same functions can be used for any of the devices.

A device is declared, constructed, and added to the device table through the opcode `device_create`. The opcode takes two extra bytes which indicate the type of device and the number it should be in the device table. First, an entry is made and added to the correct index of the device table. Next, through the device's allocator, the device data is allocated on the heap and connected to the device entry. Finally, the device's constructor is called and the device becomes ready for use.

Alex can interact with her device by getting or setting properties such as the pin or the amount of light on a light sensor or by calling methods on the device such as print on the console. Included in the definition of the device are custom additions to setters. For example, the setter method for the Grove Button, as well as setting a property such as the pin, has a custom addition that also calls the Arduino method `pinMode` to set the pin of the button with the argument `INPUT`.

3.3.6 BluetoothLE Communication

In order for the IoT device to receive a program from the Companion app or from the final project's apk, the Embedded Companion needs to setup Bluetooth Low Energy

communication. Bluetooth Low Energy (BLE) is similar to classic Bluetooth but the way you use it has significantly lower power consumption [3]. There are two principal actors in BLE communication: the central and the peripheral. A BLE peripheral broadcasts information for the BLE central to read and the BLE central reads this information but can also choose to write information back to the BLE peripheral. In the Embedded Companion, the IoT device is the BLE peripheral and the mobile app is the BLE central.

There are only two different types of messages that go between the Embedded Companion and the mobile app: `update_code` and `update_variable`. When the Embedded Companion receives an `update_code` message from the Android Companion, it stops the interpreter execution, sets up the new code, and starts interpreting the new code. Since these BLE messages are small, there may need to be many of these messages, the last of which is explicitly marked as the final message, before the entire new program has been communicated. The second message type, `update_variable`, does not interrupt the execution of a program, but rather just changes the value of a global variable before resuming normal execution. This message type is also used in the other direction; the Embedded Companion uses the `update_variable` message to tell the mobile app that it has changed the value of a shared global variable. In this case, the Embedded Companion acts as a traditional BLE peripheral by updating the `update_variable` message which notifies the BLE central, the mobile app, that the Embedded Companion has updated the variable.

3.3.7 IoT Embedded Companion Program

In order to run the Embedded Companion on an IoT controller, there needs to be a specific program that orchestrates everything. This will be slightly different for each IoT controller, but each will do the same thing. For example, the Embedded Companion needs to be formatted as an Arduino sketch in order to run on an Arduino. This "main program" will initiate communication with the mobile device, update the program code and variables when it receives messages to do so, run the interpreter, and send back new variable values when appropriate.

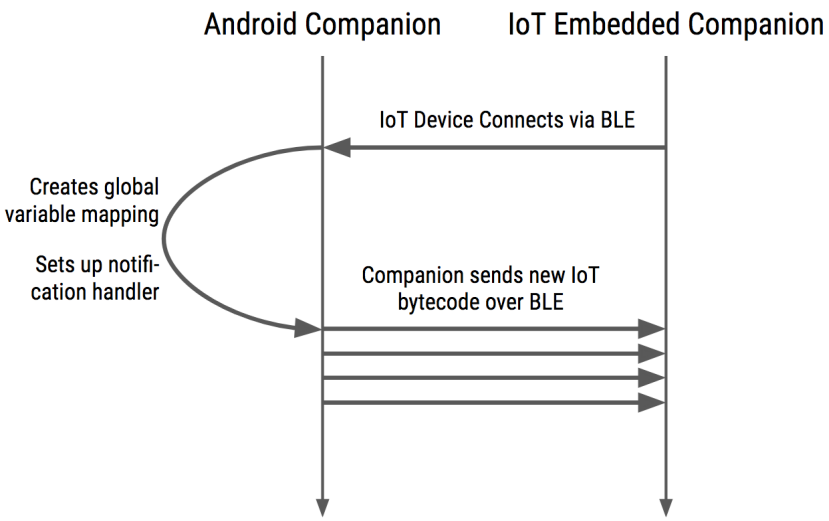


Figure 3-4: IoT Connection Protocol

3.4 App Inventor Modifications

3.4.1 App Inventor Companion App

In order to orchestrate all the necessary actions between the mobile app and the Embedded Companion, I added a class called `IotManager`. This class defines all the actions for what to do when an IoT device connects to the mobile app, when there is a new code update, and when variables are changed or need to be changed. Whenever the Companion needs to communicate with the Embedded Companion, it will do so through the `IotManager`.

When the Companion connects to an IoT device through BluetoothLE, the device notifies the `IotManager` that there is now a connected IoT device (Figure 3-4). The `IotManager` then sets up the global variable mapping between the IoT variable numbers and the Companion app's variable names as well as the BLE notification handler for when the Embedded Companion writes the update variable message. This handler receives and decodes the update variable message to figure out which global variable the Embedded Companion wants to update using the variable mapping and constructs a piece of code that will trigger the update and runs it (since code is interpreted on the Companion app).

If Alex changes the IoT component setup or the blocks, then the website will send an HTTP request to the Companion (Section 3.4.2 for more information). When the Companion receives this request, it tells the `IotManager` about the new code that constructs a `update_code` message. Because BLE messages are slow, instead of sending the entire new program, the `update_code` message only contains the pieces of code that are different from what is currently running on the Embedded Companion. When a variable is updated on the Companion, the `IotManager` figures out if the variable is a shared variable with the Embedded Companion and, if it is, constructs a BLE message to tell the Embedded Companion to update its value.

3.4.2 App Inventor Website

When Alex opens the App Inventor website in order to create an IoT application, she sees the option to add a "Sketch" to her project. This sketch is similar to an app screen except that instead of a screen on the designer section it shows an IoT platform, such as an Arduino, and IoT components, such as LEDs and light sensors (Figure 3-1). The designer has the same familiar drag-and-drop interface for the components as the designer for the app screen has. Each component has a list of properties that Alex can change in the designer. There are very few properties for each component, but they all have at least one: the pin number which describes where on the IoT microcontroller the device is connected. The blocks editor displays blocks that correspond to logic necessary for an IoT project as well as any components that Alex adds to her project (Figure 3-2). The setup and forever blocks that describe all of the program logic, outside of event handlers, are found under the microcontroller component. These two methods are analagous to the Arduino functions `setup()` and `loop()` as mentioned in Section 2.1.

When Alex makes a change on either the designer or the blocks editor, the website catches the change and triggers an update code function (Figure 3-5). This function combines the sketch JSON, which is a description of the current components of the sketch designer, and all the IoT blocks to generate new IoT bytecode. This bytecode is sent to the Companion (if connected), which forwards the update on to the Embedded

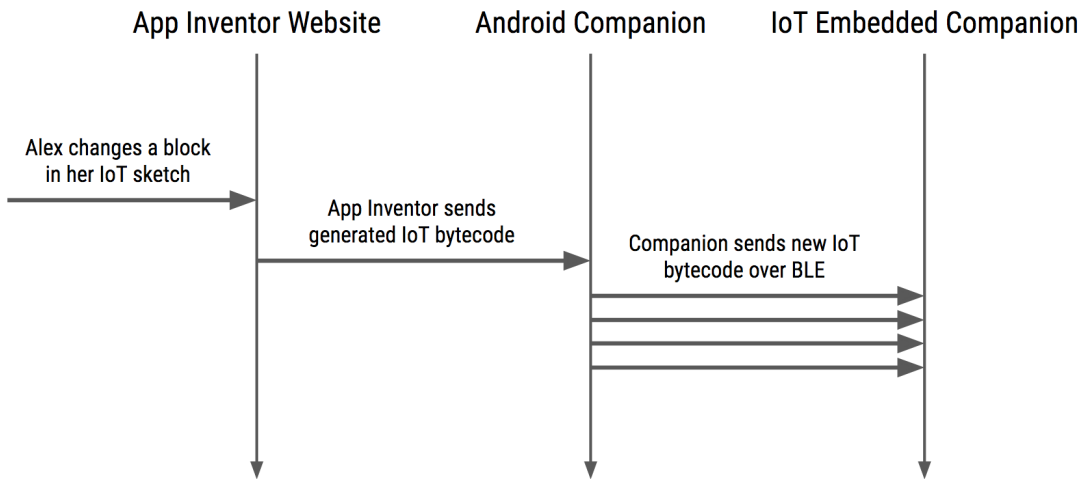


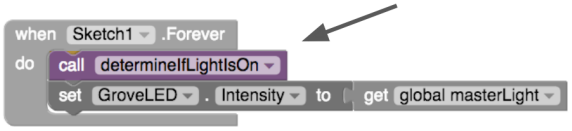
Figure 3-5: How a change is propagated from Alex to the IoT Embedded Companion

Companion.

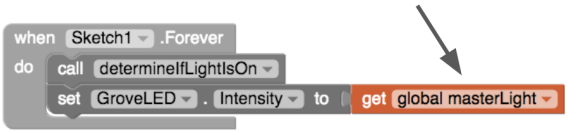
3.4.3 IoT Blockly Generator

The App Inventor IoT generator compiles blocks to the Embedded Companion's bytecode. It first records information about the total program, such as how many variables or devices there are, and uses this information to set up state that it uses to compile each stack of blocks. This state contains the mapping of variable names, device names, and procedure names to indices, which is what the Embedded Companion interpreter uses for references. Figure 3-6 has on the left a stack of blocks from Alex's smart light program from Figure 3-2 and on the right the bytecode that was generated from her blocks.

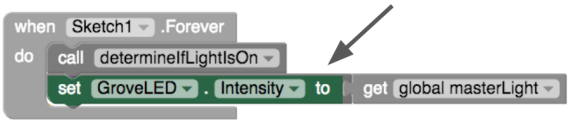
Once each name has been given an index, the generator uses the sketch JSON to create bytecode that creates each device. This entails adding a `create_device` instruction along with any property setters that correspond to properties set by Alex in the designer. All of the event handler addresses are reserved space at this time even though the addresses are unknown. This device setup code is followed by Alex's initialize block code, if she has any. Global variable declarations are added to the initialize block code as well. Each stack of blocks is then individually compiled. After all of the blocks have been compiled, the program bytecode is constructed. The



This is a `procedure_callnoreturn` block: it calls the user-defined procedure `determineIfLightIsOn` which has no arguments and is procedure 0.
Bytecode: 0x1B (`invoke_static`), 0x00 (procedure 0)



This is a `lexical_variable_get` block: it looks up the value of the global variable `masterLight` (global 2) and pushes it onto the stack.
Bytecode: 0x0A (`load_global`), 0x02 (global 2)



This is a `component_set_get` block: it sets the `Intensity` property (property 1) of a device called `GroveLED` (device 0) to the top most value of the stack.
Bytecode: 0x04 (`iconst_1`), 0x1E (`device_setter`), 0x00 (device 0)

Figure 3-6: IoT Blockly Generator Example

program header is created from the state and the program body is constructed from all the individual pieces of bytecode from each of the blocks. Once every procedure and event handler are in place and have an address, the compiler goes back and edits any place where an address is necessary, such as when setting the device to have a specific handler.

Chapter 4

Workshop Methodology

In order to test and evaluate the usability of the IoT Embedded Companion, I designed and ran a workshop based around the Embedded Companion. The workshop's goal was to introduce the concept of IoT to young people and give them a chance to make their own IoT project in order to learn about the usability of the Embedded Companion and what kind of impact IoT has on them. I applied this methodology in a workshop that I ran with fourteen students between the ages of 9 and 13 that were in grades 4 to 8. The workshop was hosted on April 7, 2018 between 1pm and 5pm for a total of 4 hours. The workshop was scheduled as a one-time event for students interested in learning more about IoT and how to harness related technology. I worked with Theresa Richards, the FIRST Robotics program coordinator at Carnegie Mellon University (CMU) in Pittsburgh, to organize this event. The FIRST Robotics program at CMU supports many FIRST LEGO League (FLL) teams in the Pittsburgh area [7]. FIRST LEGO League is a program for students in grades 4-8 that focuses on teaching and applying STEM (science, technology, engineering, and math) concepts through developing a solution to a real-world problem, such as recycling, and designing and building a robot using LEGO MINDSTORMS [22]. Theresa Richards reached out through the FLL teams to find students who might want to attend the workshop. Because of this, some of the students had experience with STEM and programming, though not all. It is important to note LEGO MINDSTORMS blocks only consist of pictures and therefore even though the concepts are the same

(e.g. a while loop or using variables), the way they are expressed are different and not easily related to one another [13].

4.1 Workshop Lesson Plan

The four hour-long workshop was split into two parts in order to allow a break in the middle and give the students a chance to absorb the information that they learned in the first part to help them with the second part.

The goal of the first part was to introduce the workshop and the students to IoT and MIT App Inventor and teach them how to make a project with the Embedded Companion (Table 4.1). The beginning of part 1 was the most important conceptual part of the workshop because it was the part that focused on what IoT was and where it appears in our lives. This part was meant to be a short introduction of the basic concepts of IoT and then a discussion to start the students thinking about what they interact with that uses IoT technology and what they can do with IoT technology themselves.

Part 2 introduced the independent project and related materials provided to assist the students in completing it. The students were given a template in the form of an MIT App Inventor project (AIA) file and a handout that corresponded to the project explaining the main components (Appendix B). The students were also given a handout that listed and explained how to use all of the sensors and components available to them for their project (Appendix C). The remainder of the workshop was devoted to students working on their independent projects.

4.2 Data Collected

Data was collected during the workshop through written questionnaires, project files, and observation. Students filled out a pre- and post-questionnaire during the workshop. Both of the questionnaires are structured around the students' conceptual understanding of IoT and what they understand about the capabilities of computers

Time	Activity
5 min	Class introductions: name, any programming (Scratch, Python, etc) or Arduino or Lego MINDSTORM experience
5 min	Have students fill out the Pre-Questionnaire (Appendix D)
15 min 10 min	What is App Inventor? • A website that lets you create mobile applications easily What is the "Internet of Things" ("IoT")? • A system of connected small devices through the Internet (usually wifi or bluetooth) • Ask students what devices they think of that fall under IoT • Popular examples: "smart" outlets, light switches, watches, TVs • Give a do-it-yourself example: You have a device with a motion sensor that plays a song every time someone passes by. You can change this song or turn it on/off from your smartphone. How to use IoT with App Inventor • We can use inexpensive Arduinos as our IoT device and connect to it different sensors and components • In App Inventor, you program what happens on the Arduino – When people interact with the components (press buttons) – When sensor data changes (it becomes darker in the room) – What it does in its free time (every minute it flashes a light) • Also in App Inventor, you program the mobile application – What it looks like – How people can interact with the app (buttons, pictures) – How people can change the Arduino (change it to flash every second instead of minute)
10 min	Introduce the Autonomous Healthy Plant project • Show the finished and working Autonomous Healthy Plant project • Talk about what sensors are used and why in the project
1 hr	Have students go through the Healthy Plant Tutorial (Appendix A)

Table 4.1: Workshop Part 1 Introducing App Inventor Lesson Plan

Time	Activity
5 min	Introduce the Project • Tell the students that the goal of the project is to let them explore the capabilities of IoT by trying different sensors and components and using them to make a project of their choice • Tell the students that they will get a template application that they can alter to create their project
15 min	Give the students the AIA for the Project Template App and Template Info Handout (Appendix B) • Explain the functionality of each of the sensors and devices • Explain any interactions that may happen between the mobile application and the Arduino • Explain how they can change the blocks to it do something else
1 hr 10 min	Give out the Sensors and Components Handout (Appendix C) and let students start creating their projects. Make sure the students download and email their project AIA files to helik@mit.edu.
7 min	Have students briefly explain what their project does to the class
7 min	Have students fill out the Post-Questionnaire (Appendix E)
1 min	Thank the class for coming and tell them they can now go off and make their own IoT projects!

Table 4.2: Workshop Part 2 Building an Independent IoT Project

and how they relate that capability to themselves. The post-questionnaire also includes questions about how the students' projects went and how hard it was to use App Inventor and the Embedded Companion.

The pre-questionnaire was given at the beginning of the workshop before the introduction and discussion of IoT. The first and last questions tried to probe at the students' understanding of the capabilities of computers and what they think they can create with them. The second question tried to gauge what familiarity the students had with IoT and whether they could acknowledge it as such before introduced to the definition of the term.

The post-questionnaire was given at the end of the workshop and asked all the same questions as the pre-questionnaire with a few additional questions about the project the students worked on during the workshop. The goal of the post-questionnaire was twofold: (1) to gauge how much the students learned over the course of the workshop and how it changed their perception of computing capabilities and what they can do with them and (2) to gather information on the students' experience while working on their project with App Inventor and the Embedded Companion.

The students' MIT App Inventor project files were also collected at the end of the workshop. These files can be opened in the version of App Inventor that I have implemented for the IoT Embedded Companion. The file can be opened to view the designer and blocks editor of the app screen and the IoT sketch. There also exists a tool made by the App Inventor team to query aspects of the collection of AIAs [1]. Additionally, any noteworthy interactions with students were collected.

[1] AIA Tools `https://github.com/mit-cml/aiatools`

Chapter 5

Results and Discussion

The workshop consisted of fourteen students in grades 4-8. All had their own computers to work with (all but 4 were personal laptops, the rest were provided by the FLL teams Theresa Richards manages), but a couple were not able to connect to the internet so four students worked in pairs, totaling the number of projects to 12. The data collected during the workshop sought to answer the following research questions: (1) to what extent can students understand the functionality of IoT and its applications, (2) how has learning about IoT changed the students' perception of computers and what they themselves are capable of making with them, and (3) how well can students create projects with IoT in App Inventor using the Embedded Companion.

To address the first question, I analyzed students' pre- and post-questionnaire responses using the pre-questionnaire responses as a baseline understanding of IoT technology and comparing the post-questionnaire responses against this baseline. The second question uses the students' responses to the first and third questions from the pre- and post-questionnaires. The final question uses the second section of the post-questionnaire, project files, and the amount and type of help students asked for while working on their projects.

It is important to note that because of the internet in the space provided for the event, I was not able to run a local server to App Inventor and students were not able to use the Companion app for their projects. Unfortunately, this meant that a lot of time was spent trying to load and connect to the remote App Inventor server

and students were not able to use live development. Therefore, the results from this workshop are insufficient to address the entire IoT Embedded Companion system. The results, discussion, and conclusion provided in the rest of this work focus on the students' conceptual understanding of IoT and how well students are able to use the Embedded Companion on the App Inventor website to create projects.

5.1 Students' Understanding of IoT

In part 1 of the workshop, the definition of IoT along with some popular and do-it-yourself examples were introduced to the students and a discussion with the students followed. Before this, the students completed a pre-questionnaire that asked the question *"What do you think the "Internet of Things" (or "IoT") is?"*. At the end of the workshop, about three and a half hours later, the students completed a post-questionnaire with the same question.

5.1.1 Pre-Questionnaire Responses

The responses to the pre-questionnaire fell into three categories: guesses at what IoT was, a vague idea of the main concepts of IoT, and a display of understanding of IoT. It is important to note that a couple students arrived late and therefore were answering the pre-questionnaire during the introduction of IoT.

Out of the fourteen responses, seven of the responses fell into the guesses category. On the two extreme ends, one student didn't try to guess but simply put *"?????"* and another student attempted to use the name *"Internet of Things"* to create an answer *"I think the "IoT" is the internet that you can search tons of things."* Two of the responses were similar and said *"I think it is a program that creates apps."*

Three of the responses fell into the vague idea category. Each student had a unique answer that hit on one of the main concepts of IoT:

- *"Small devices that are programmed to do things (Alexa)"*
- *"A way to allow different softwares to talk and communicate with each other"*

- *"A way to physical[ly] involve computer programming"*

The IoT concepts that these answers mentioned are small devices, communication, and the physical nature of IoT applications.

The remaining four responses fell into the category of a display of understanding IoT. An example response is *"A network of "smart-devices" working together and synchronizing with each other."* Every response involved networking or communication, but only one answer provided examples that used IoT technology: *"The means by which you connect your toaster to your smartphone: a computer network composed of home appliances, sensors, and controllers."* Only this last answer expressed an understanding of how IoT is used and where it appears in daily life.

5.1.2 Post-Questionnaire Responses

The post-questionnaire responses can also be divided into the same three categories. Overall, the number of guesses decreased to four, the number of vague ideas decreased to one, and the number of answers that displayed understanding of IoT increased to nine.

Three of the responses that fell under the guesses category said that IoT is an *"app builder."* These responses are very similar to some of the ones in the pre-questionnaire. It is unclear if the students who responded like this were the same as the students in the pre-questionnaire because the questionnaires did not ask for name and the pre- and post-questionnaires were not connected by student. Without this information, it is hard to distinguish the case where the students answered the same on both questionnaires because they did not learn anything about IoT and the case where some students' understanding of IoT became focused on creating applications because they made IoT applications in the workshop. In either case, the introduction and discussion of IoT did not have the desired effect on the students. It seems that since the bulk of the time of the workshop was spent creating applications, this is what the students took away from the workshop.

The one response that fell under the vague concepts category displayed less un-

derstanding of IoT than the responses in the same category in the pre-questionnaire: *"I learned about the arduino and how to make music."* However, the student is able to understand that they created an IoT project even if they cannot provide an explanation of what IoT is exactly.

The nine responses that fell under the understanding category all mentioned the connection or networking of the devices of IoT. Two of the responses included examples, both of which included *"Alexa"* indicating that they were heavily influenced by the examples mentioned in the discussion of IoT in part 1 of the workshop. Three of the responses included a mention of IoT applications using data, sensors, or interacting with the physical world. This displayed some understanding on how IoT devices can be used to do things that conventional personal computers cannot. One response in particular displayed understanding about IoT without simply repeating what was discussed in the beginning of the workshop: *"Smart devices working together to create a good experience."* This student took the discussion about IoT and how many IoT devices communicate to create some desired functionality and synthesized a colloquial way of thinking about it.

5.2 Students' Perception of Computing and Capabilities

The other two questions on the pre-questionnaire will be used in order to address the second question about how students' perception of computing and capabilities, both of computers and of themselves, have changed after learning about IoT. The two questions are as follows:

- *"What types of things do you think computer programs can do? List everything you can think of."*
- *"What do you think YOU can create with computer programs?"*

Since these questions were both open-ended, each response consisted of multiple answers. I sorted these answers into the following categories: robots, apps and games,

Category	Pre-Questionnaire		Post-Questionnaire	
	Question 1	Question 3	Question 1	Question 3
Robots	6	3	6	2
Apps & Games	8	10	4	8
Problem Solving	5	2	2	1
Data & Computation	5	3	3	0
Controlling Devices	5	2	10	9

Table 5.1: Category Breakdown of Questionnaire Responses

problem solving, data and computation, and controlling physical devices. These results are summarized in Table 5.1 where 'Question 1' refers to what students thought computers can do and 'Question 3' is what the students thought they could create. Not all of the students' responses fell into these categories and are therefore omitted from this table. In particular, one student answered *"Give[n] the time and resources anything I want"* to question 3 in both the pre- and post-questionnaire which is not reflected in this table.

For question 1 of the pre-questionnaire, each category had roughly the same number of responses. However, in the post-questionnaire, responses to the same question were more unevenly split between categories—"Controlling Devices", for example, had twice as many responses, when comparing the pre- and post-questionnaires, while "Problem Solving" and "Data & Computation" had half as many responses each. The increase in responses in the "Controlling Devices" category implies that many students were influenced by the workshop and thus responded with IoT-related answers. Similarly, this influence also likely accounts for the reduction in responses in other categories.

I am skeptical to interpret the lack of responses in other categories as meaning that students no longer believe that computers can function in the same ways that they listed before. However, the fact that students did not include many responses in all of the categories possibly indicates that they were not thinking about all the capabilities of a computer at once. The increase in responses about controlling devices shows that students learned that computers have that capability, but the lack of other responses demonstrates that students have not yet integrated this understanding into

their conception of computing. Based on these responses, learning about IoT did not strictly increase the students? knowledge and perception of computing.

A similar shifting in the number of responses in each category occurs in question 3 as well. The number of responses increases by 7 in the controlling devices category and there are 7 less responses in all of the other categories combined. However, the mean and standard deviation in the pre-questionnaire are 4 and 3.4, respectively. This standard deviation is even higher than that of question 1 in the post-questionnaire. This is because 50% of the students' responses fell into the apps and games category. In the results from the post-questionnaire, the mean was still 4 and the standard deviation was 4.2, even higher than in the pre-questionnaire. The cause of this is now instead of one main answer (apps & games), there are two (apps & games and controlling devices). In the post-questionnaire, 45% of the responses fell into the controlling devices category and 40% into the apps and games category. For question 3 as compared to question 1, it seems that the students' belief that they can make apps and games did not diminish, but their belief that they can make programs that control devices increased greatly. However, the other categories received less responses than in the pre-questionnaire which again shows the influence of the workshop on the students' beliefs of their computing capabilities.

5.3 Creating an IoT Application in App Inventor with the Embedded Companion

As noted before, the internet in the space provided for the event was slow and restricted such that time was lost trying to connect to the remote App Inventor server and students were not able to use the Companion. This created a non-ideal scenario in which the students were not able to fully test the Embedded Companion system. However, students still learned how to use the App Inventor workshop to create an IoT project and started to create their own projects even though they were unable to finish or test them.

5.3.1 Students' Responses On Project Completion

The second half of the post-questionnaire pertained to the IoT projects that the students spent time creating. The first question asked *"Did you finish the app you wanted to make?"* to which all but 2 responses were *"No."* The next question asked the student to explain the answer further if it was a no: *"If no, why not? Did you run out of time? Could you not figure out how to do something? Do you think you're unable to create it in App Inventor?"* Out of the 14 responses, there were 6 indicating that they ran out of time.

The reasons for running out of time varied from *"I ran out of time since my computer didn't load"* to *"It also took too long to figure out some things that did not line up with the tutorial."* These types of responses unfortunately do not correspond directly to the functionality of the Embedded Companion in App Inventor itself. Both responses explain why the student ran out of time to complete their independent project and neither of the reasons are directly related to the functionality of App Inventor. The first response corresponds to the unfavorable conditions in the workshop of slow internet and computers which meant that trying to connect to the server took up a lot of time. This problem was unavoidable in the situation and should be taken into account when analyzing the students' projects and responses.

The second response also was not directly because of App Inventor, but instead about the teaching materials that I used during the workshop. Unfortunately, some of the tutorial pictures got out of sync with the current system and therefore did not match directly. These small differences between the system the students were using and the tutorial caused a lot of confusion. The students asked many questions around these discrepancies and once they received answers, they were able to move on to the rest of the tutorial or project. However, this did cause a problem during the workshop and a few students echoed this sentiment with responses about being *"confused"* or *"starting to get frustrated"* and that they *"didn't [finish] because [they] didn't know how to program it."* This emphasizes the importance of correct teaching materials and tutorials because small differences make a difference and can hinder

the students' learning.

5.3.2 Students' Responses On Difficulty

The last question on the post-questionnaire was *"How hard was it to build your project?"* The responses fell into a few categories as described in the students' responses: *"not really," "medium" or "from 1-10, about a 5," "hard",* and *"very hard."* Out of the 14 responses, 2 fell under the *"not really"* category, 4 under *"medium,"* 7 under *"hard,"* and 1 under *"very hard."* These responses indicated that about half the students were okay with the content and pace of the workshop, but the other half were not. The question itself did not ask for clarification about why they thought it was as hard as they indicated, so it is difficult to know exactly what the problems were.

However, some of the responses had an explanation which provides insight into why the students thought building the project was hard. Four students mentioned that *"the basics"* were hard to figure out. One of these students responded that *"it was fairly difficult learning the basic parts of the software."* These responses indicated that part of the difficulty in creating a project was with using the basics of App Inventor for the first time, which is unsurprising since the students only had 2 hours to learn how to use with App Inventor before starting their project. Some students were able to get past this hurdle and said *"it was not very hard once me and my partner figured out the basics."* For others, it was the source of difficulty for their project *"medium - we didn't get past the tutorial."*

All of the other responses either had no explanation or the provided explanation expanded only on the level of hardness such as *"harder than I expected" or "not to[o] hard but it was a bit challenging."* None of the responses indicated that the concepts of IoT or the specific tools for IoT were difficult to understand or use.

5.3.3 Students' Project Files

Unfortunately, not all of the data from the students' project files were saved as a result of the workshop. Here I will analyze the screen components, screen blocks, and the IoT sketch components since it was the IoT sketch blocks that were lost. Additional data was gathered later from a different group of students whose IoT sketch blocks will be analyzed. This group consisted of four students who were all familiar with App Inventor, but had never used the IoT Embedded Companion before. Each student spent approximately 90 minutes working on making the Healthy Plant Tutorial (Appendix A) and adding some personal adjustments. These projects are the ones analyzed here.

The following 3 tables summarize the results of analyzing the students' projects. Table 5.2 shows the analysis of the projects created in the workshop. Since all of the projects were collected at the end of the workshop, some of the projects were abandoned (due to connectivity issues) and thus appear as a 0 in most of the categories analyzed. Table 5.3 shows the analysis of the projects created after the workshop during further data collection. Because of the limited time the students had for these projects, some of the students only focused on creating the IoT sketch instead of the mobile app screen and therefore had 0 additional screen components, but the greatest number of additional sketch component types used.

In all of the projects analyzed, almost no students added any additional component types to their mobile app designer screen. An "additional component type" is a component type that was not used during the tutorial. Among all projects, only 1 student used any additional components on the mobile app designer screen and this students used 3 different types of components. However, 7 students added additional component types to the sketch designer with a maximum of 5 different types to a single project. This difference indicates that students were more focused on adding to the IoT part of their project. Without additional information the reason cannot be determined, but three possibilities are as follows. Students added to the IoT part of the project because (1) students were more comfortable adding to the IoT sketch,

	Average	Range	Standard Deviation
Screens	2	1 - 4	0.6546536707
Screen Components	18	1 - 29	6.761234038
Additional Screen Components	1	0 - 8	2.555506259
Additional Screen Component Types	0	0 - 3	0.8017837257
Screen Global Variables	1	0 - 3	1.059456927
Sketches	1	0 - 4	1.180193689
Sketch Components	7	0 - 13	4.250450156
Additional Sketch Components	1	0 - 5	1.533636468
Additional Sketch Component Types	0	0 - 5	1.293626448

Table 5.2: Workshop Project Results

	Average	Range	Standard Deviation
Screens	1	1 - 1	0
Screen Components	15	1 - 22	8.584142357
Additional Screen Components	0	0 - 1	0.433012701
Additional Screen Component Types	0	0 - 0	0
Sketches	1	1 - 2	0.4330127019
Sketch Components	8	6 - 12	2.165063509
Additional Sketch Components	2	0 - 5	1.802775638
Additional Sketch Component Types	1	0 - 3	1.089724736

Table 5.3: Additional Project Results

(2) students are only thinking about the IoT part of the application, or (3) students added to the IoT sketch first and did not have time to add to the screen. In order to determine which of these is the case, another study should be run using these as research questions. The results would help improve teaching materials and curriculum surrounding IoT.

Table 5.4 shows the types of additional sketch components that students used in projects. These additional components are ones that are included after the ones used in the tutorial. For example if the tutorial uses 1 GroveLED and the student used 3 GroveLEDs in their project, 2 additional GroveLED components have been used. The students in the workshop started with the Template App (Appendix B) while the students in the second data collection group did only the Healthy Plant Tutorial (Appendix A) so the distribution of additional types of sketch components used is different. Across both sets of projects, students added some new components (e.g. the GroveMoisture component for the workshop projects and the GroveBuzzer for

Type of Component	Number Used in Workshop	Number Used in Additional Data Collection
GroveMoisutre	1	0
GroveBuzzer	1	3
GroveLED	4	0
GroveRGBLCD	3	1
GroveHumidity	2	0
GroveButton	2	4
Console	1	2
BluetoothLE	1	0
Components Used in Tutorial	7	1
New Components	8	9

Table 5.4: Additional Sketch Components Used

	Project 1	Project 2	Project 3	Project 4
Shared Global Variables	4	4	4	0
Mobile App Shared Global Get-Actions	11	11	11	0
Mobile App Shared Global Set-Actions	0	0	0	
IoT Device Shared Global Get-Actions	1	1	0	0
IoT Device Shared Global Get-Actions	4	4	4	0
Sketch Events	4	7	4	0
Initialize Block Height	0	0	0	0
Forever Block Height	0	0	0	0
Sketch Component Get-Actions	0	0	0	0
Sketch Component Set-Actions	2	8	1	2

Table 5.5: Additional Projects - IoT Sketch Blocks Editor Results

the additional projects). Interestingly, the students in the workshop added about an equal number of components from and not from the tutorial while the students in the additional data collection group only added 1 additional component from the ones used in the tutorial. This difference may be a result of the length of time each student spent working with the IoT Embedded Companion or the types of components used in the tutorial.

The projects from the second data collection group saved properly and therefore further analysis can be made on the IoT sketch blocks in the students' projects (Table 5.5). The first part of this table shows the analysis on shared global variables. The mobile app and the IoT device communicate through shared global variables when

they are connected. Without any connections, the variables work like normal, but when they are connected the result of a mobile app or IoT device write to a shared global variable is sent to the other and updated accordingly. Therefore to analyze the amount of mobile app and IoT device communication, we need to analyze the shared global variables.

The number of shared global variables is the amount of state that the two applications share at any given time. Each project had 4 shared variables with an outlier of 0. The number of IoT device shared global get-actions describe how much information the IoT device receives from the mobile app. The 2 projects that each have 1 get-action perform the get after a set-action and therefore this information is not actually received from the mobile app. The number of IoT device shared global set-actions describe how much information the IoT device sends to the mobile app. Similarly, the number of mobile app shared global get-actions describe how much information the mobile app receives from the IoT device and vice versa for the set-actions. None of the projects had any extra global variables or get- or set-actions than needed for the tutorial. Because most of the time spent was doing the tutorial and there were only 4 projects analyzed, the results are inconclusive as to why no additional communication was added.

The number of sketch events in an IoT project describes how much an IoT device reacts to sensory input. The number of component get-actions describes how much the IoT device reads data without it triggering an event and the number of component set-actions describes how much the IoT device changes the devices that are attached to it. The Healthy Plant Tutorial, on which these projects were based, included 4 sketch events and 2 component set-actions. Only project 2 had any additions built on top of the tutorial. This was most likely due to time constraints, but it means that there is not enough data to draw any conclusions about the behavior of students when using the IoT Embedded Companion while making projects. From this one project though, we can gather some preliminary ideas to test later with more research.

The additions that were made to project 2 were an increase in the number of sketch event and the number of component set-actions. Interestingly, this does not

include any additional blocks to the Initialize or Forever block so everything that the IoT device does is reactionary. One of the benefits of the IoT Embedded Companion is that projects can have logic that is purely a result of the IoT device being on instead of as a result of some input. Again, because of the lack of data, the reason for this needs to be explored further.

5.4 Discussion

As the differences between the pre- and post-questionnaires showed, the workshop was generally effective in teaching students the basic concepts of IoT and empowering students with the ability to use IoT technology. Even though only a small fraction of the workshop was devoted to discussing IoT, the students were able to leave knowing what IoT was and with some ideas on how they could use IoT technology in a project to affect the physical world.

The workshop projects and post-questionnaire responses on them, along with the four additional projects collected at a later time, indicate that the learning experience surrounding the IoT Embedded Companion could be improved. The students in the workshop found following the tutorial without guidance very difficult and would frequently ask questions about how to proceed. Since not all students using the IoT Embedded Companion would have a teacher, it is important that tutorial be improved to help ease this confusion. In addition to making the tutorial easier to understand, it would be ideal to make the tutorial more interactive so that students learn what to do instead of following the instructions without considering what they are doing.

None of the students had enough time to use the live experimentation feature of the IoT Embedded Companion and so we do not have any information about its effectiveness. The ability to use live development is important in debugging and experimentation and further study needs to be done to determine whether this makes the IoT Embedded Companion easier to understand and use. Since students were not able to use it because they did not finish the tutorial or their project, it is important that the live development feature is introduced earlier in the learning process in future

studies.

Chapter 6

Conclusion

In this thesis, I presented a system that gives Alex the ability to create her smart light IoT application. App Inventor provides Alex with a single, online platform to create IoT applications equipped with a synchronized mobile app and IoT device. To create this system, I built the IoT Embedded Companion and modified and enhanced MIT App Inventor to take advantage of the new capabilities. I then designed a workshop to teach students about IoT technology and test their use of the IoT Embedded Companion. Afterwards I ran the workshop with 14 middle-school aged students and analyzed the results of the workshop.

Based on the data collected during the workshop, I found that students were able to understand the key concepts of IoT and that this knowledge influenced the way the students thought about the capabilities of computing. I also found that teaching the IoT Embedded Companion to students without prior knowledge of MIT App Inventor in the short amount of time described with the current teaching materials is difficult. However, once students figured out the basics of how to use MIT App Inventor and the IoT Embedded Companion, they were able to construct their desired IoT applications.

Given these results, Alex would probably have to take a bit of time to learn how to use the IoT Embedded Companion, but once she did she would only have to worry about what she wants her project to do, not how to keep separate systems synchronized—and she would have only had to learn one system!

Chapter 7

Future Work

In order to complete and improve the IoT Embedded Companion as a full-fledged system ready to be added to MIT App Inventor, there are two main focuses for future work. The first is a better tutorial system. As the results from the workshop showed, the tutorials need to be improved to help students learn how to use the Embedded Companion effectively so that they are able to utilize it fully in their own projects. Further research into what types of tutorials work best to help students learn and what the best way to teach students about the full capabilities of IoT is necessary.

The second focus for future work is improving the system itself. Because this work only provides the basic implementation of the IoT Embedded Companion, not all of the App Inventor basic logic blocks were translated for the Embedded Companion. Further improvements to the system would be the addition of basic logic blocks such as text, lists, local variables, and delay. Other improvements would focus on optimizing the IoT bytecode generator, enhancing the sketch editor to be interactive like the screen designer, and extending the number of sensors and devices available.

Lastly, the most important improvement is to generalize the IoT Embedded Companion so that it is available to be used on multiple platforms. The core code of the Embedded Companion is already accessible to all platforms that run C programs, but the device and BLE code need to be generalized to work with any platform.

Appendix A

Healthy Plant Tutorial

The rest of this appendix includes the PDF of the Healthy Plant Tutorial given to students at the workshop.

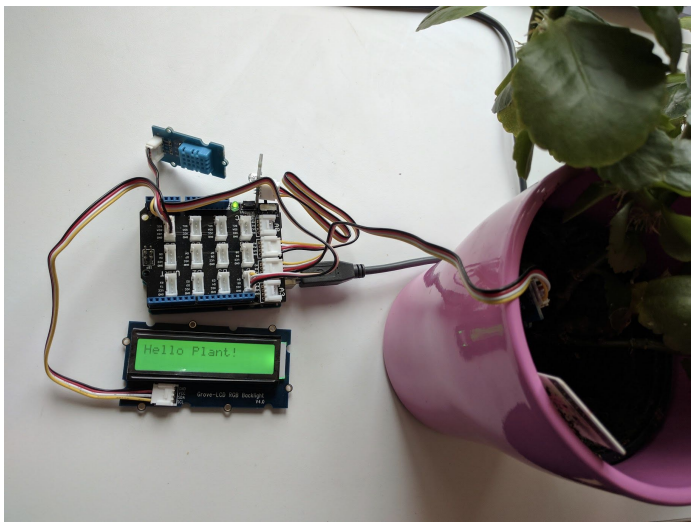
App Inventor + IoT: Building an Autonomous Healthy Plant App

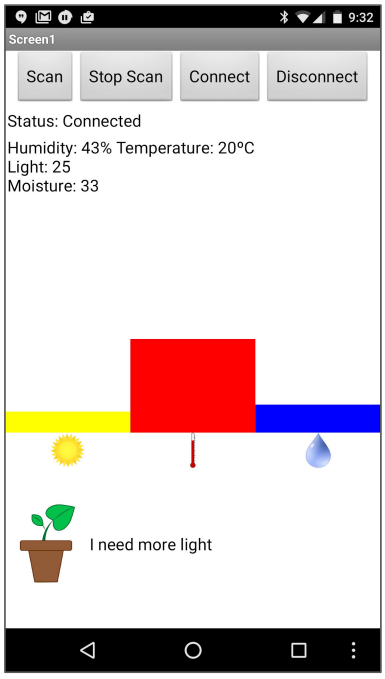
60 min

This tutorial will help you get started with building an App that connects and respond to the physical world - often termed the Internet of Things or "IoT".

In this project we're going to learn how to build an app that connects to a microcontroller called Arduino 101 via Bluetooth and runs on the Arduino even when the app is not connected via Bluetooth. You can use this equipment to monitor various conditions (i.e., light, humidity, temperature, moisture) that can help you track the overall health of a plant. We will also learn how to graph this data and how to autonomously respond to the data by turning on an LED when the plant needs to be watered.

We are also using a Seeed Grove shield for this tutorial.

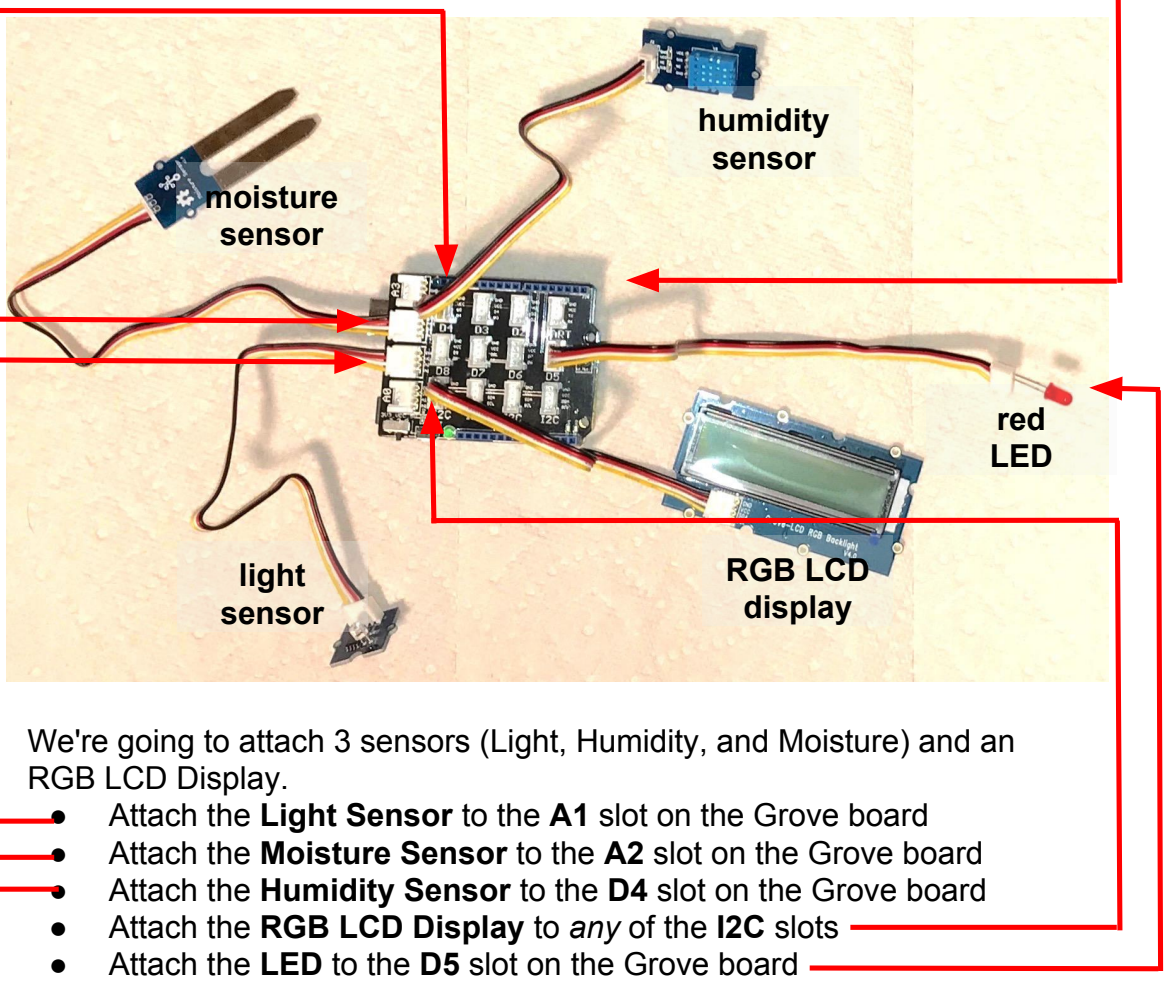




Setting up the Arduino

Let's start by connecting all the sensors we're going to use to our Arduino. For this project, we are also using a Seeed Grove shield attached to the top of our Arduino. While the Grove board isn't necessary for IoT projects, it makes things much easier. We will also need the following components for this tutorial:

- A moisture sensor
- A light sensor
- A humidity sensor (also works as a temperature sensor)
- An RGB LCD Display
- An LED

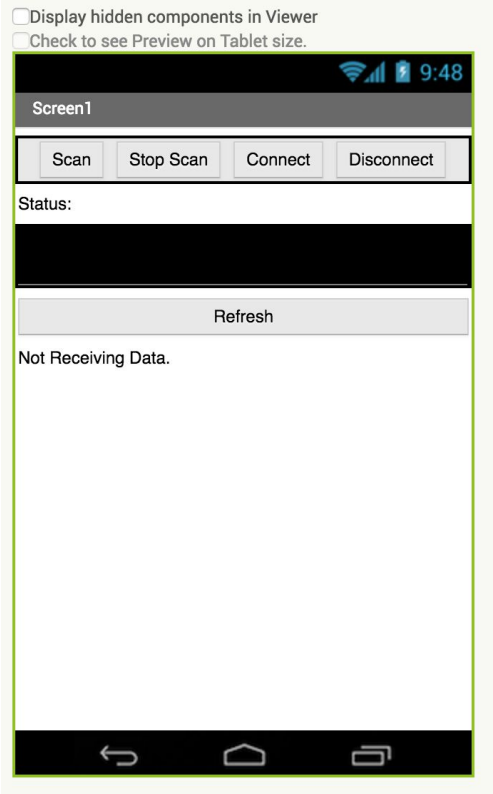


We're going to attach 3 sensors (Light, Humidity, and Moisture) and an RGB LCD Display.

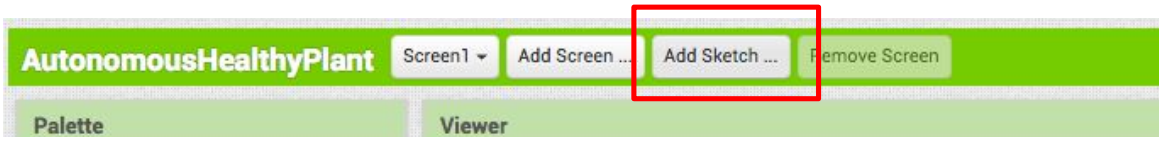
- Attach the **Light Sensor** to the **A1** slot on the Grove board
- Attach the **Moisture Sensor** to the **A2** slot on the Grove board
- Attach the **Humidity Sensor** to the **D4** slot on the Grove board
- Attach the **RGB LCD Display** to *any* of the **I2C** slots
- Attach the **LED** to the **D5** slot on the Grove board

Building the App in App Inventor

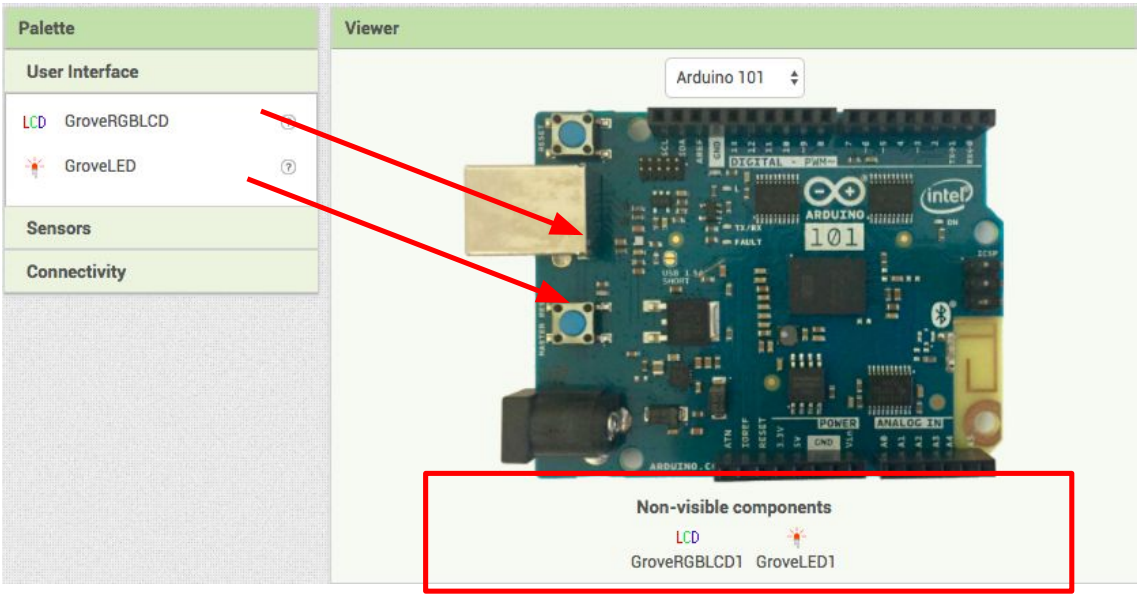
Open the AIA project for the Healthy Plant Tutorial in App Inventor. It is currently just an app that looks for and connects to BluetoothLE devices.



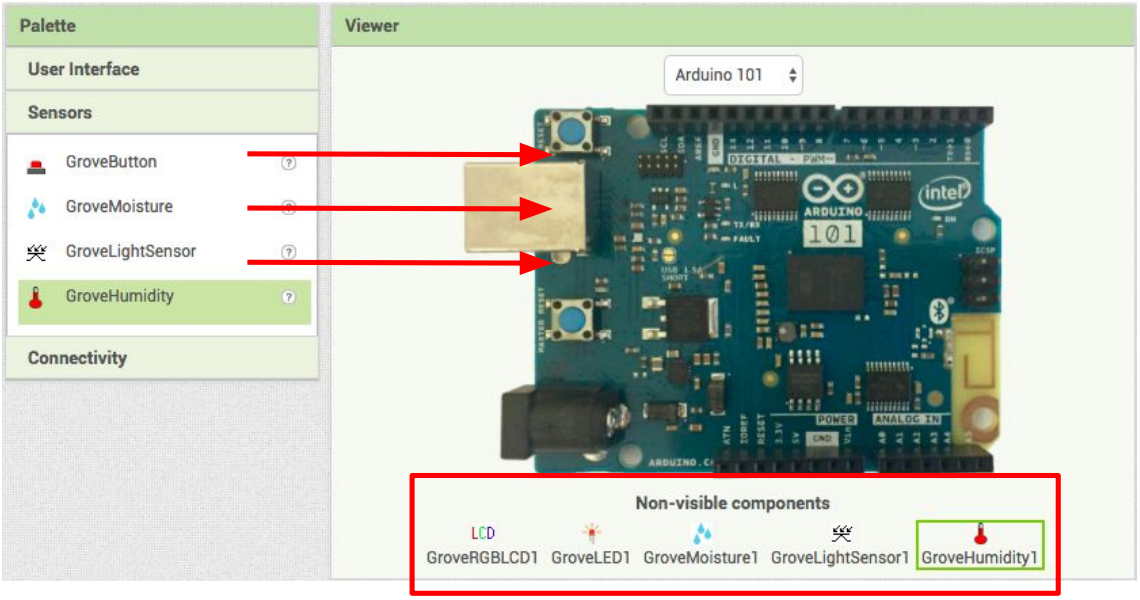
Now we want to add all the sensors and displays from the Grove kit that we already wired to the Arduino. To do this, first add a sketch with the “Add Sketch…” button as shown below. The dropdown button next to the project title will switch to “Sketch1” and a picture of an Arduino will appear.



From the “User Interface” list, drag the GroveRGBLCD onto the viewer.



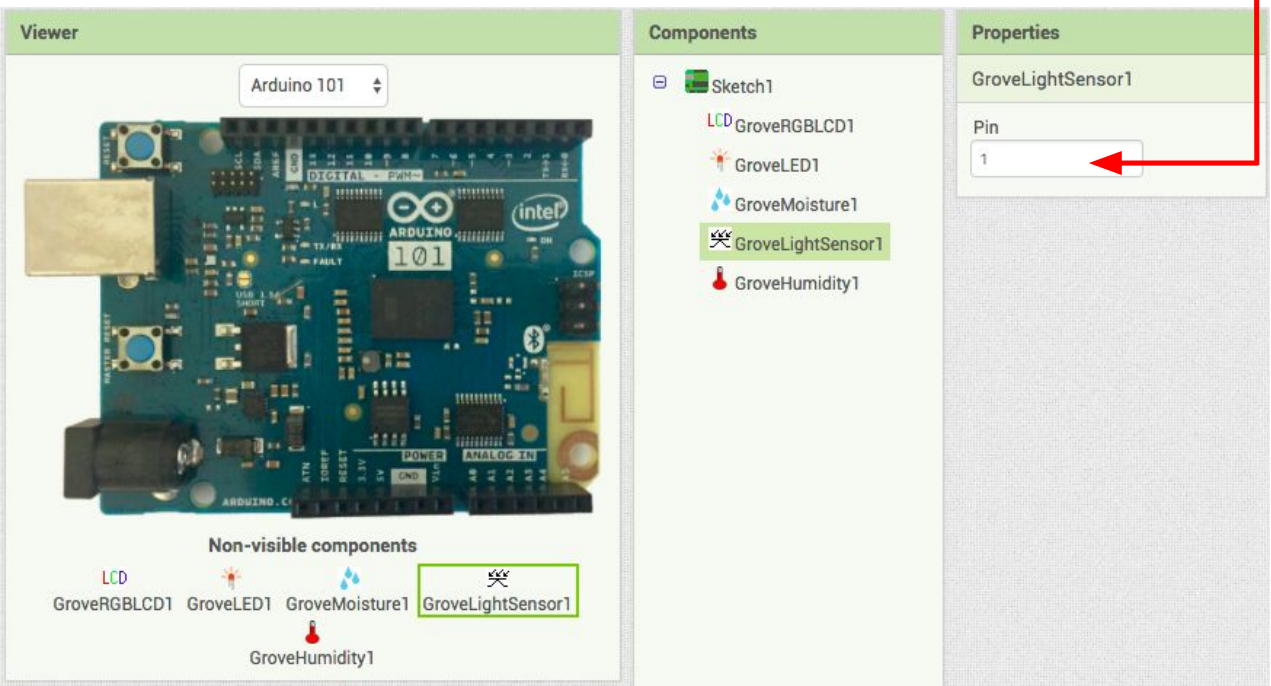
From the “Sensors” list, drag the following sensors into the viewer: GroveMoisture, GroveLightSensor, GroveHumidity



Next, we need to let App Inventor know which pins on the Grove board the different sensors and the LCD screen are connected to.

First, let's set the pin for the Light Sensor

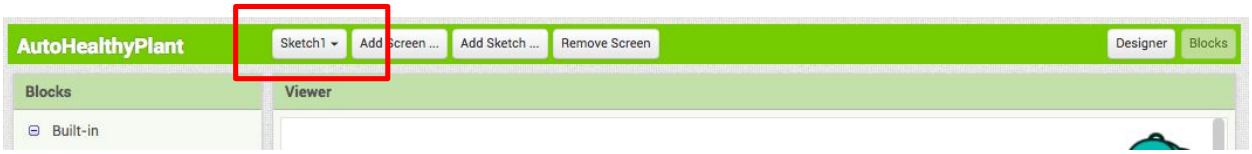
- Click on **GroveLightSensor1** in the Components pane.
- In the Properties pane, under **Pin**, enter *only the number* that corresponds to the analog pin the light sensor is plugged into on the Grove board (in this case A1).
  - *Note: You only need to set the number (1) not the letter (A)*
- Now, let's do the same thing for the rest of the sensors
  - Click on **GroveMoisture1**, set its pin to 2
  - Click on **GroveHumidity1**, set its pin to 4
  - Click on **GroveLED1**, set its pin to 7
    - *Note: The LED's intensity can be set to 1-100. Intensity will only affect the brightness of the LED if it is plugged into a pin supporting pulse width modulation (PWM). For the Arduino 101, the PWM pins are 3, 5, 6, and 9. For all other pins the LED will either turn on or off with no change in intensity.*
- For the **GroveRGBLCD1** you don't have to set the pin; App Inventor will take care of it. In the properties for the LCD, you can set the background color to whatever you wish!



Now we are ready to program the Arduino!

Switch to the Blocks Editor view for Sketch1

Blocks can be added to either the Sketch or Screen Blocks editor - you can switch between these with the dropdown boxed in red below.



The blocks for any program you want to run **autonomously on the Arduino** must go in the **sketch** blocks editor. Any program for just the **app** will go in the **screen** blocks editor.

Since this project is autonomous, we'll handle each sensor's data in the Blocks Editor for our sketch, "Sketch1."

Now we need to store the data of each sensor.

- From Variables drawer drag four **initialize global name to** blocks and name them **light**, **moisture**, **temperature**, and **humidity.**
    - set each one to a value of **0**

We're also going to want to turn the LED on and off sometimes, so we can make procedures to do this. Procedures make it easy to reuse blocks and they are a good way to keep blocks neat.

- From Procedures drawer drag two purple **to procedure do** blocks and name them **TurnOnLED** and **TurnOffLED.**
    - For **TurnOnLED**, we want to set the LED's intensity to 1. From the GroveLED1 drawer drag **set GroveLED1.Intensity to** block. Set this value to **1**.
    - For **TurnOffLED**, repeat the same step as the previous procedure but set the value to **0** instead of 1.

Next, we want to update the global variables we made when we receive data from the sensors.

- From GroveLightSensor1 drag **when GroveLightSensor1.IntensityChanged**
- from Variables add **set global light to**
  - hover over the orange "intensity" in the **.LightSensorDataReceived** block to see the **get intensity** block.
  - Drag the **get intensity** block from this window and snap to **set global light to**

Now, we'll repeat this for humidity, temperature, and moisture.

- From GroveHumidity1 drag **when GroveHumidity1 .HumidityChanged**
  - from Variables add **set global humidity**
  - hover over the orange "humidity" to see **get humidity**
  - drag the **get humidity** block and snap to **set global humidity to**

- From GroveHumidity1 drag **when GroveHumidity1 .TemperatureChanged**
  - from Variables add **set global temperature**
  - hover over the orange "temperature" to see **get temperature**
  - drag the **get temperature** block and snap to **set global temperature to**

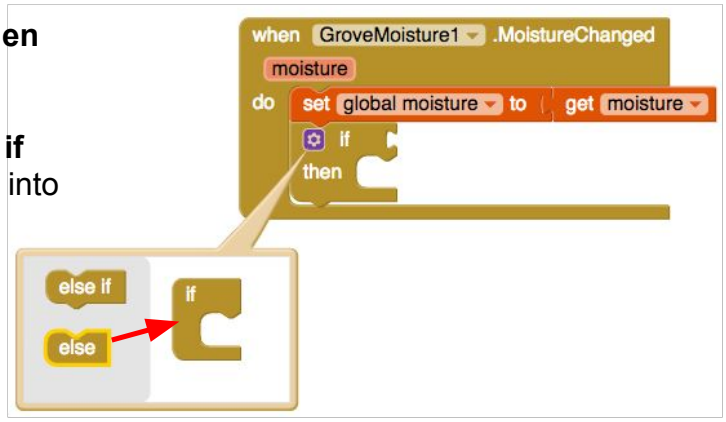
Repeat this one more time for the moisture sensor.

- From GroveMoisture1 drag **when GroveMoisture1 .MoistureChanged**
  - from Variables add **set global moisture**
  - hover over the orange "moisture" to see **get moisture**
  - drag the **get moisture** block and snap to **set global moisture to**

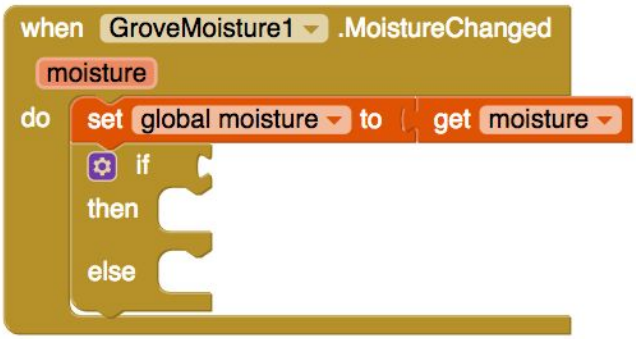
For the moisture sensor, soil is considered "too dry" is the moisture is below 300. Let's turn the LED on whenever the soil is dry, so that we are reminded to water our plants. So,

if moisture is less than 300, **then** turn the LED on, **else** turn the LED off.

- From Control drawer, drag **if then** and snap below **set global moisture to**
  - click the gear icon on the **if then** block and drag **else** into the **if** block

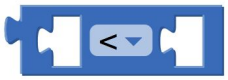


Your block should now look like this:



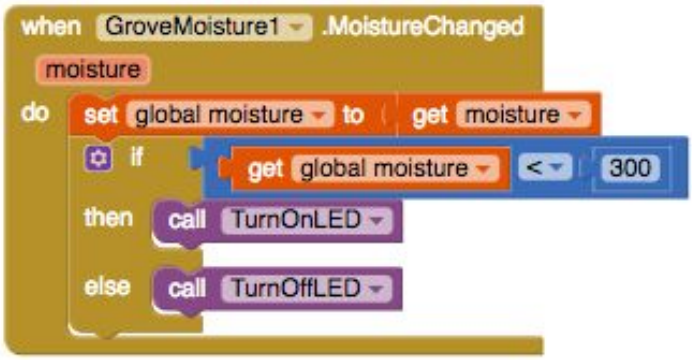
Desired function: **if** moisture is less than 300, **then** turn the LED on, **else** turn the LED off.

- From Math drawer, drag a blue **compare equals** block and change the compare from equals to **less than**



- From Variables add **get global moisture** and snap to first position in blue compare block
- Set second position in compare block to **300**

- Snap this to the **if** inside our **GroveMoisture1.MoistureChanged** block
- From Procedures drawer drag purple **call TurnOnLED** and snap to **then**
- From Procedures drawer drag purple **call TurnOffLED** and snap to **else**



Now we're finished with our blocks for the sketch, so the last step is to build the blocks for our app screen.

Switch to the Blocks Editor view for Screen1

Start by making global variable for moisture, light, temperature, and humidity. These mimic the same global variables that we made for our sketch. AppInventor takes care of keeping these variables up-to-date with the global variables from the sketch which are set by the Arduino.

- *Note: these variables should only be set in the Screen Blocks editor (except for initialization) because only the Arduino's sketch should set these.*

- From Variables drawer drag four **initialize global name to** blocks and name them **light**, **moisture**, **temperature**, and **humidity.**
  - set each one to a value of **0**

To check if we have received data, check if any of the global variable have a value other than the initial value we set, **0**. We can do this by linking together **or** blocks from the Logic drawer.

- From Logic drawer, drag three **or** blocks and link them together as shown below

- From Math drawer, drag four **compare equals** blocks and change them to **not equal** to **0**

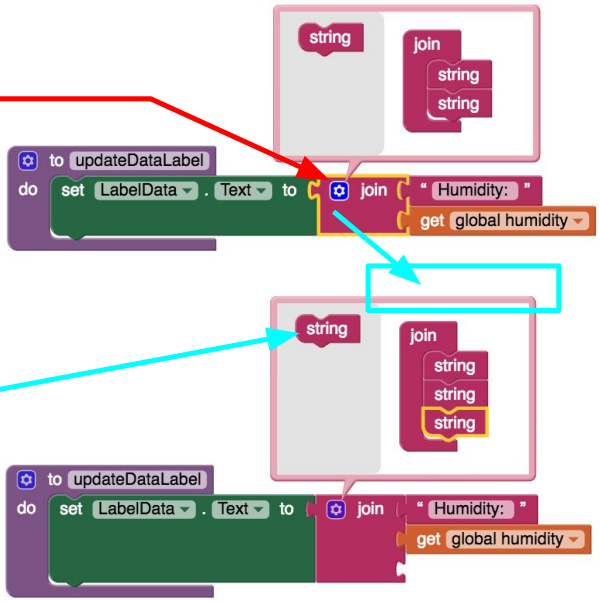
- Snap **compare not equal** into the four available spaces in our linked **or** blocks

- From Variable drawer, drag four **get** blocks and change the dropdown to be each of the four global variables: **global humidity**, **global light**, **global moisture**, and **global temperature.** Snap each of these into the available spots in our linked **or** block
- From Procedure drawer, create a **to procedure result** block named "checkIfAppIsReceivingData" and snap linked **or** block into procedure's result

Let's make a new procedure to display the current readings in the LabelData. Let's rename it **updateDataLabel**

- From the LabelData Drawer add **set LabelData.Text to**
    - From the Text Drawer connect a **join block.**

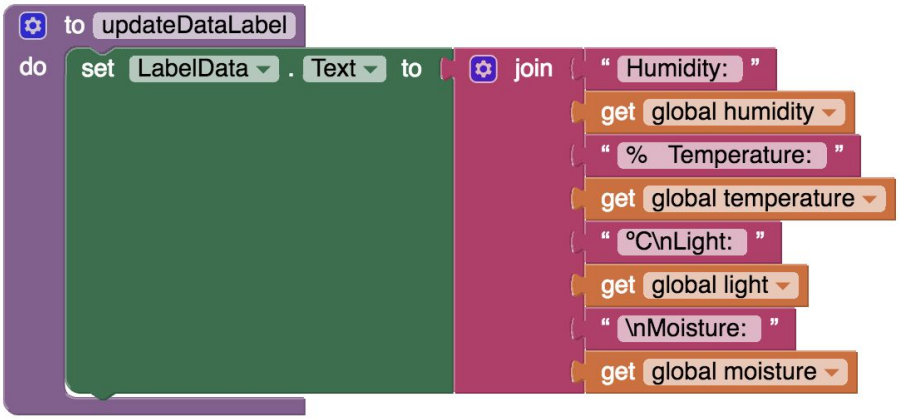
You'll notice that the join only has two slots at first and we have 8 items! This is an easy fix. In the **Join** block you'll see a blue gear, click on it and a new window appears.



Then drag the **string** block on the left side under the **string** blocks inside the join. This will add a new slot. Do this 6 times in total.

    - Add the following blocks to the join (for some of them it might be easier to copy and paste than to type them yourself):
        - from the Text Drawer add a text block and type in **"Humidity: "**
        - from the Variables Drawer add a block **get global humidity**
        - from the Text Drawer add a text block and type in **"% Temperature: "**
        - from the Variables Drawer add a block **get global temperature**
        - from the Text Drawer add a text block and type in **"ºC\nLight: "**
        - from the Variables Drawer add a block **get global light**
        - from the Text Drawer add a text block and type in **"\nMoisture: "**
        - from the Variables Drawer add a block **get global moisture**

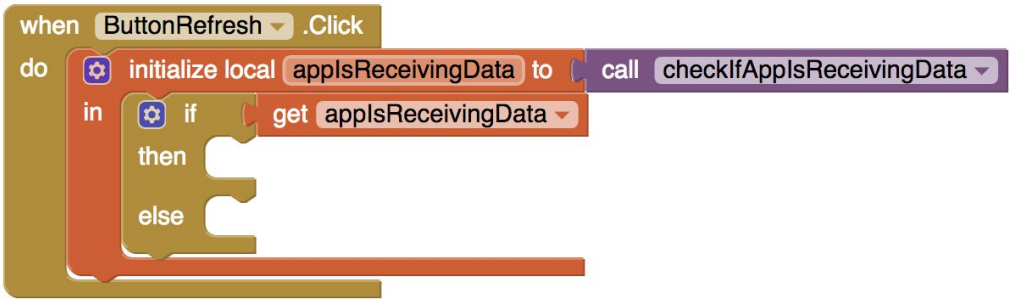
Once we're done, the final procedure should look like this:



Now, let's update LabelStatus and LabelData to reflect current data when we are receiving data.

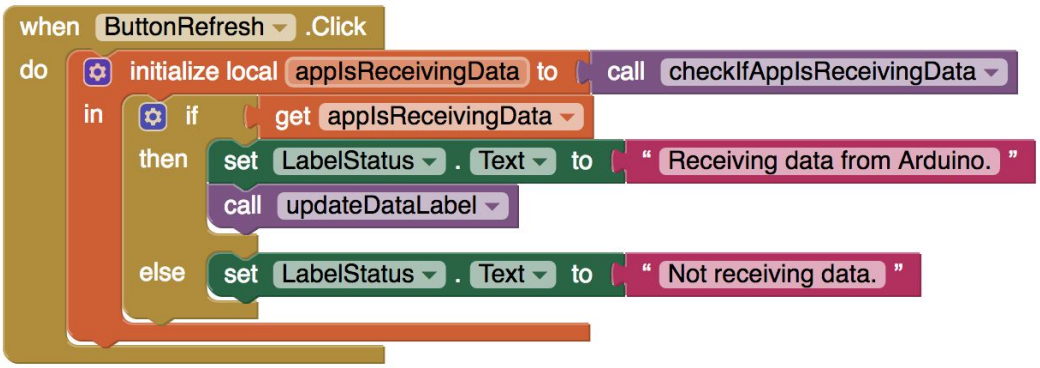
- From ButtonRefresh drawer, drag **when Button.Click do** to viewer.
- From Variables drawer, drag **initialize local variable to** and name the local variable "appIsReceivingData"
- From Procedures drawer, snap **call checkIfAppIsReceivingData** into **to** so that our local variable is true when we have data from the Arduino

Next, we'll use this local variable to decide what to set our labels to. First, make an **if then else** block from the Control drawer like we did with the moisture sensor on page 9.



- From LabelStatus bucket, drag **set LabelStatus.Text to** into **else** spot and set it to **"Not receiving data."**

- From LabelStatus bucket, drag **set LabelStatus.Text to** into **then** spot and set it to **"Receiving data from Arduino"**
- From Procedures drawer, drag **call updateDataLabel** and snap underneath **set LabelStatus.Text to** in the **then** section

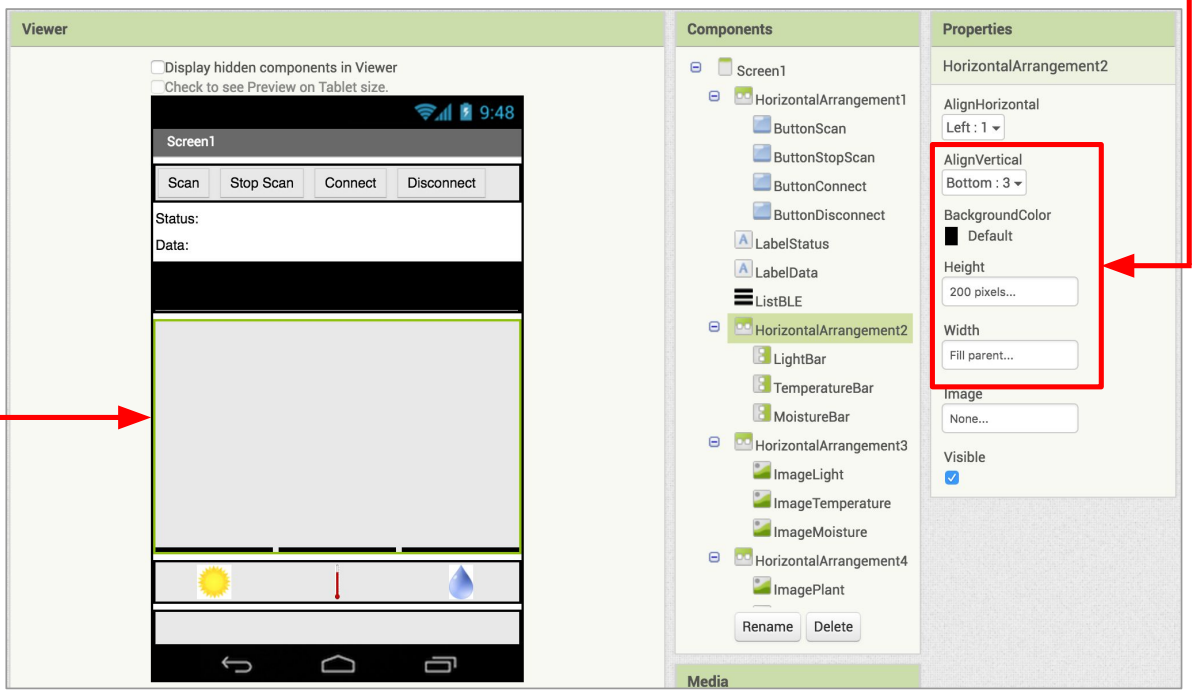


Now let's make it look nicer by adding some colorful bars to graph some of our data.

Switch to the Designer view

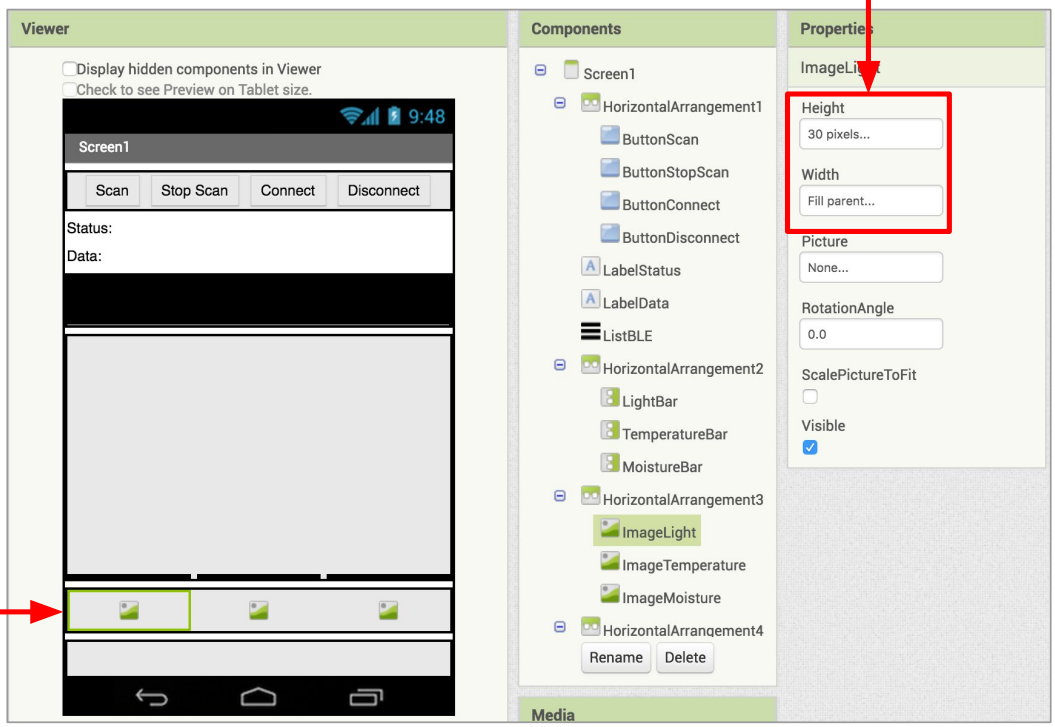
We need to create the area for the bar graphs.

- Drag a *HorizontalArrangement* from the Palette and place it below **ListBLE**
  - Set its properties as follows:
    - *AlignVertical*: **Bottom: 3**
    - *Height*: **200px**
    - *Width*: **Fill parent**
  - Add 3 *VerticalArrangements* inside the *HorizontalArrangement* and rename them **LightBar, TemperatureBar, MoistureBar**
  - Set each *VerticalArrangement*'s height to **0px** and width to **Fill Parent**
    - Now set **LightBar** *BackgroundColor* to Yellow, **TemperatureBar** *BackgroundColor* to Red, and **MoistureBar** *BackgroundColor* to Blue



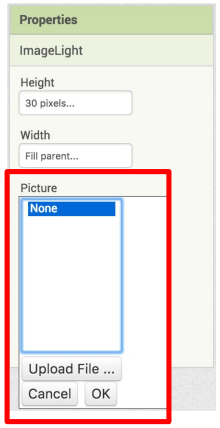
Let's create a legend so we know what each bar represents.

- Drag a *HorizontalArrangement* from the Palette and place it below *HorizonalArrangement2*
  - Leave its *Height* at **Automatic** and set its *Width* to **Fill parent**
  - From the User Interface Palette, drag 3 **Image** components onto the *Horizontal Arrangement*, and rename them "ImageLight", "ImageTemperature", and "ImageMoisture"
    - Set each Image's properties to:
      - *Height* to **30px** and *Width* to **Fill Parent**



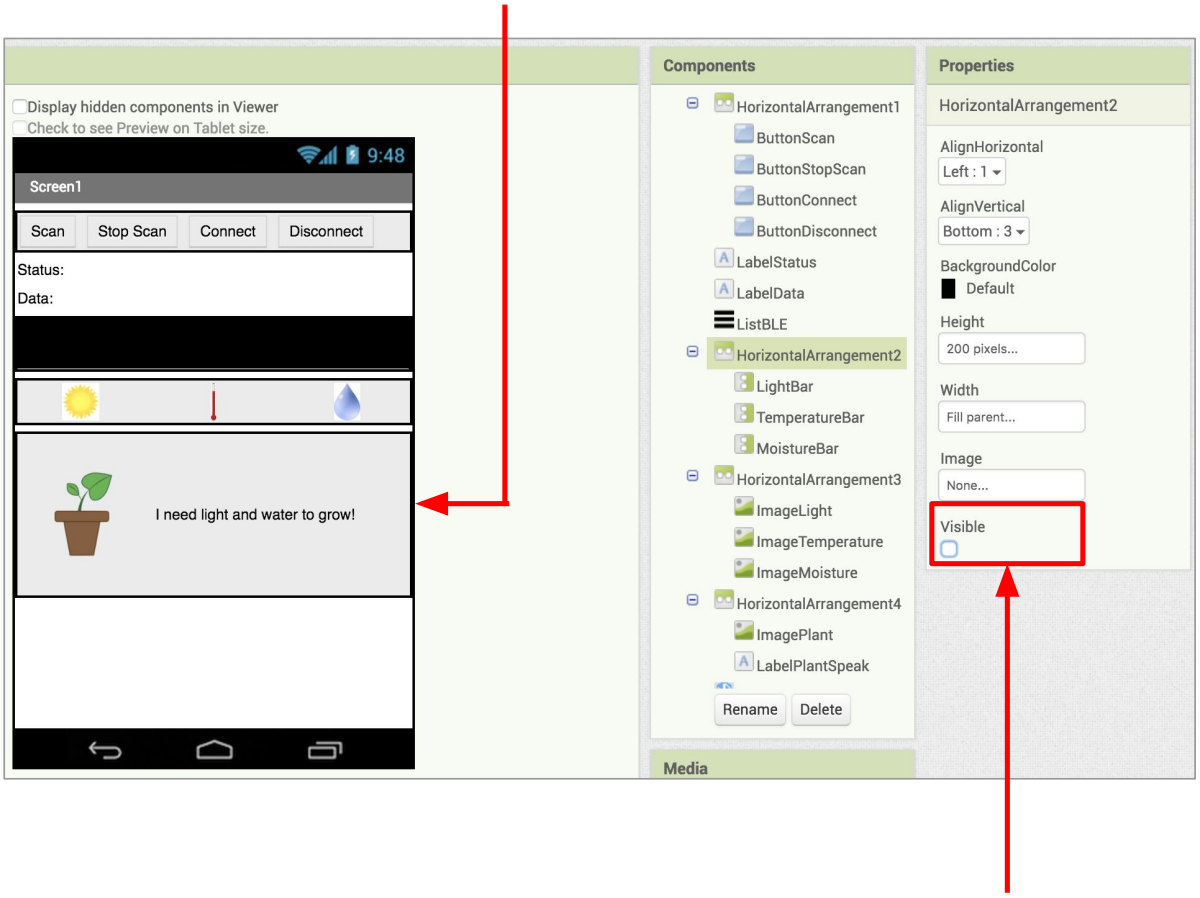
Now we want to add the images for the legend.

- Download the following 3 pictures to your computer:
  - Sunlight, Thermometer, and WaterDrop
- Under the Properties pane for **ImageLight**, click on Picture.
  - In the pop-up window click on "Upload File…"
  - Find the Sunlight image on your computer and upload it
  - Repeat this process for **ImageTemperature** and **ImageMoisture**



Let's set up a space so that our plant can "talk" to us based on its status.

- Drag a *HorizontalArrangement* from the Palette and place it below *HorizontalArragement3*
  - Set its *AlignVertical* to **Center: 2**, *Height* to **130px**, and *Width* to **Fill parent**
  - Download the PottedPlant picture to your computer and then Upload it to the project
    - Drag an **Image** component onto *HorizontalArrangement4*, and set its picture to the Potted Plant picture.
    - Rename the Image "ImagePlant"
    - Set its *Height* to **70px** and *Width* to 70**px**
  - Drag a **Label** to the right of ImagePlant, rename it "LabelPlantSpeak" and change its *text* to "I need light and water to grow!"

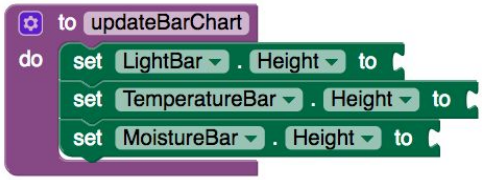


NOTE: If HorizontalArrangement4 is off the screen, try hiding HorizontalArragement2 temporarily by clicking the Visible button in HorizontalArragement2's Properties pane.

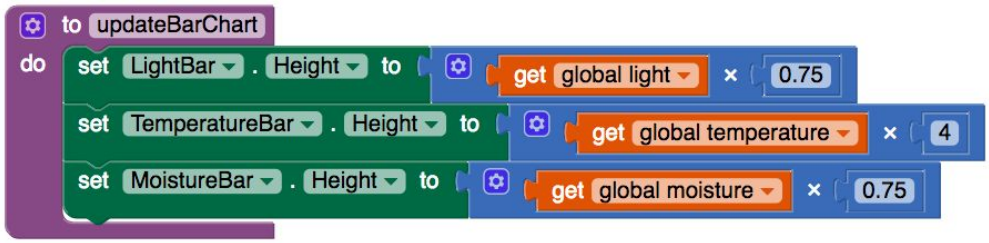
Switch to the Blocks Editor view for Screen1

To update the graph as we get data, we're going to add a procedure to update the height of LightBar, TemperatureBar, and MoistureBar. By setting the height of each of the vertical arrangements, we can create bars reflecting the sensor values by their change in size.

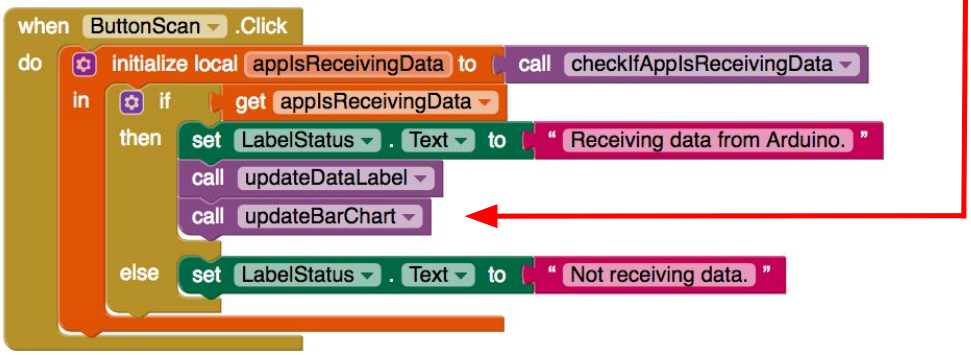
- Create a procedure named "**updateBarChart**"
- Snap **set** blocks for each vertical arrangement



Now, use the Math drawer and global variables to set the height as shown below.



Finally, add a call to this procedure after the call to **updateDataLabel**



Now try out your app using the MIT AI2 companion - when the sensor data changes, the bar graphs should also go up and down.

A few other things you could do to enhance your app!

- You could change the color of the LCD display if the soil is too dry or too wet instead of turning on/off an LED.

- Or, change the colors of the graph bars when the conditions change.

This is just one example of how App Inventor + IoT can work together to help us understand, and change, our everyday lives. If you come up with more, be sure to share them with us! You can reach us by emailing appinventor@mit.edu . Enjoy!

Appendix B

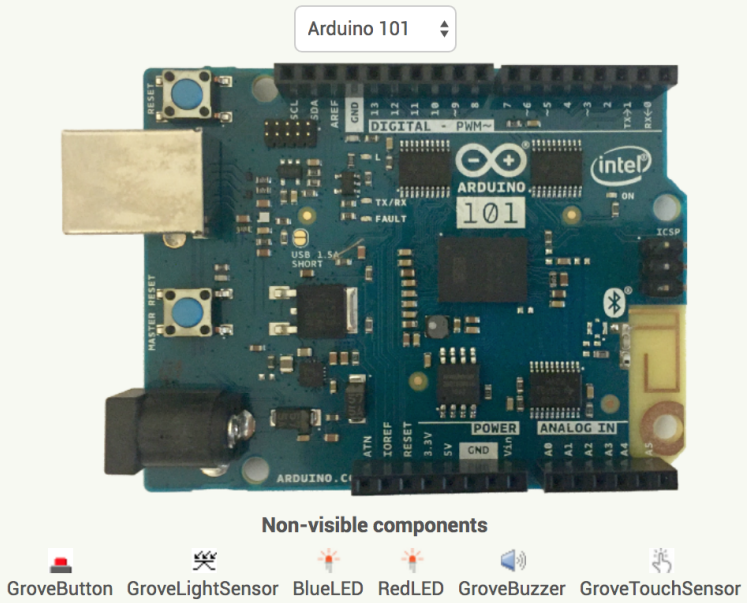
Template Handout

The rest of this appendix includes the PDF of the template handout given to students at the workshop.

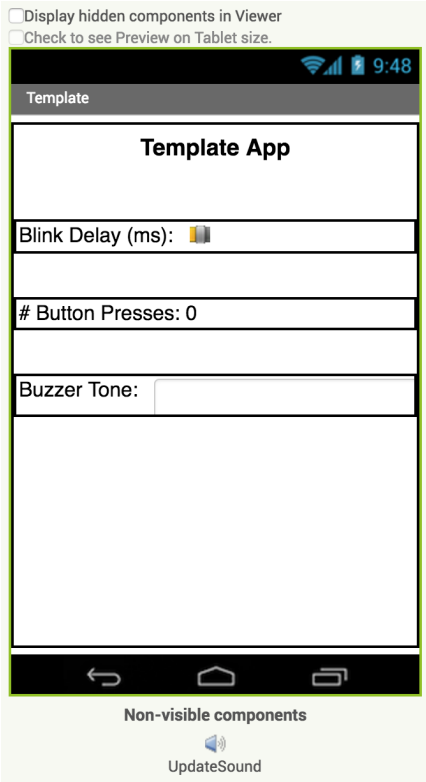
Template Handout

Starting Setup

Below pictures what Arduino components are in template project. The components are a button, light sensor, blue LED, red LED, buzzer, and touch sensor. You can find what each of these components are for further down in this handout. Before you connect any physical components to your Arduino, think about which ones you actually want to use for your project. Remember take this template and make it your own!



The mobile app screen has a place to interact with the blink blue LED, the button, and the buzzer.

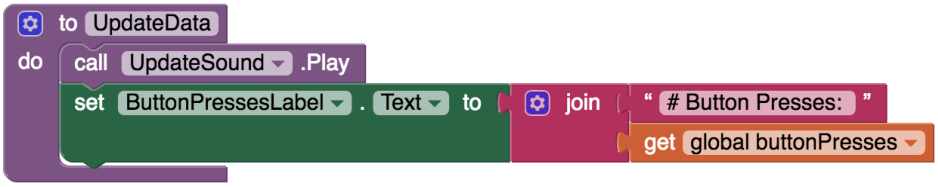


Counting Button Presses

When the button is pressed, the Arduino adds 1 to a tally of how many times the button has been pressed. The mobile app also displays this count.

Arduino: When you click the button, it adds 1 to the total number of button presses.

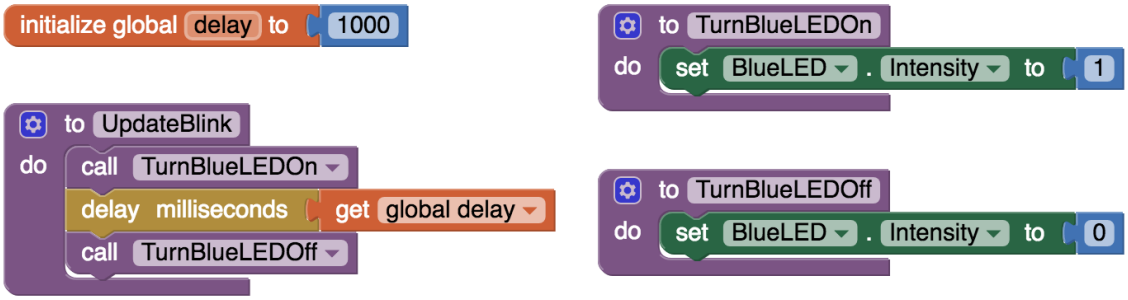
Mobile App: When you get a new number of button presses, update the display.



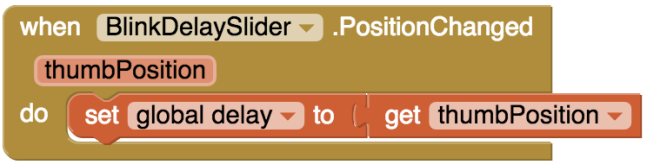
Blinking an LED

An LED connected to the Arduino blinks at an interval given by the variable “delay.” This variable can be adjusted on the mobile app by moving a slider.

Arduino: The blocks in the Arduino sketch uses a variable called “delay” to set the interval of the blink. To blink the LED, there are procedures to turn on and off the blue LED.



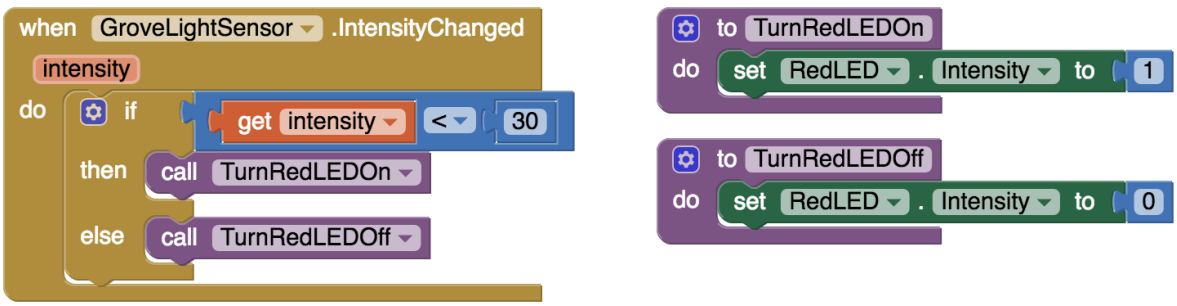
Mobile App: The slider on the mobile app allows you to change how fast the LED blinks. The “delay” variable is shared between the mobile app and the Arduino.



Sensing Darkness

A light sensor measures the lightness of the area and if it goes below the threshold (which starts at 30), the red LED turns on and if it goes above the threshold, the red LED turns off.

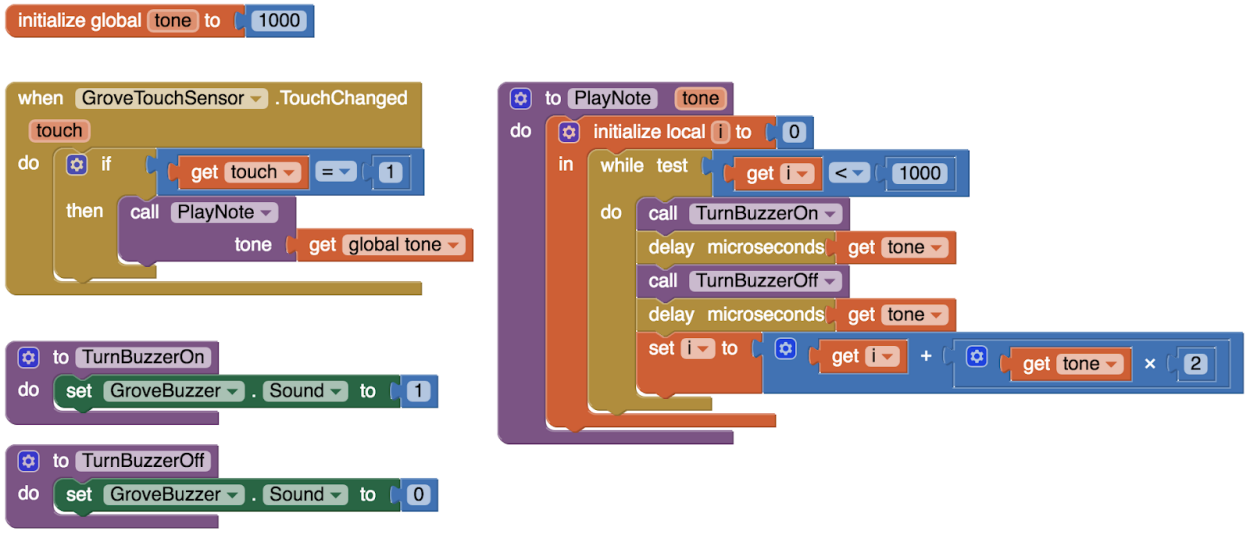
Arduino:



Playing Music

When you touch the touch sensor, a sound plays on the buzzer. The mobile app lets you change what tone should be played next.

Arduino: When you touch the touch sensor, the buzzer plays the tone set by the shared variable "tone." The "PlayNote" procedure creates what is called a square wave to change the frequency (or tone) of the sound that is played.



Mobile App: After you enter a number into the buzzer tone textbox, set the shared variable "tone" to that value.

Appendix C

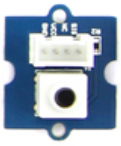
Sensors and Components Handout

The rest of this appendix includes the PDF of the sensors and components handout given to students at the workshop.

Sensors and Components

Here is a list of the sensors and components that you can use for your project! You can find a picture of the sensor or component and some examples of what blocks to use with it.

Button



Use this block to trigger actions when the button is clicked:

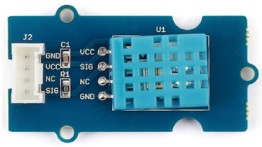
Buzzer

Use this block to turn **on** the buzzer:

Use this block to turn **off** the buzzer:

Use this block to determine if the buzzer is on or off. The value will be 1 if it is on and 0 if it is off:

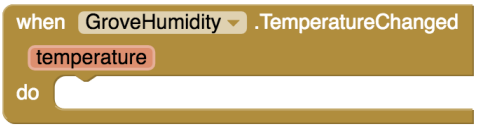
Humidity Sensor



Use this block to get the current temperature anywhere:

Use this block to get the current humidity anywhere:

Use this block to trigger actions when the temperature changes. You can use the “temperature” variable to get the current temperature:



Use this block to trigger actions when the humidity changes. You can use the “humidity” variable to get the current humidity:

LEDs

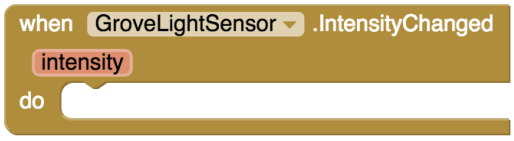
Use these blocks to turn **on** the LED:

Use these blocks to turn **off** the LED:

Use this block to determine if the LED is on or off. The value will be 1 if it is on and 0 if it is off:

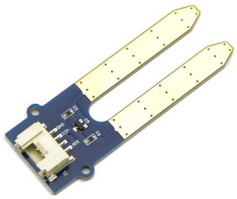
Light Sensor

Use this block to trigger actions when the light changes. You can use the "intensity" variable to get the current brightness of the light:

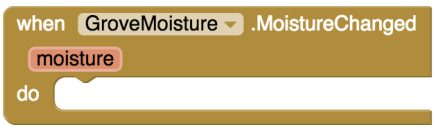


Use this block to get the current brightness of the light anywhere:

Moisture Sensor

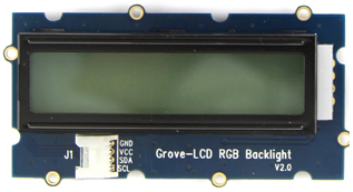


Use this block to trigger actions when the moisture changes. You can use the "moisture" variable to get the current moisture level:



Use this block to get the current moisture level anywhere:

RGB LCD Display

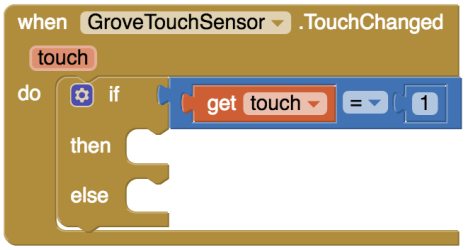


Use this block to set the color of the LCD display:

Use this block to get the current color of the LCD display:

Touch Sensor

Use this block to trigger actions when you either start or stop touching the touch sensor. The "then" section of the if block is run if you just started touching the sensor and the "else" section runs if you just stopped touching the sensor:



Use this block to determine if the touch sensor is being touched or not. A value of 1 means that it is and 0 means that it is not:

Appendix D

Pre-Questionnaire

The rest of this appendix includes the PDF of the pre-questionnaire given to students at the workshop.

Pre-Questionnaire

What types of things do you think computer programs can do? List everything you can think of.

What do you think the “Internet of Things” (or “IoT”) is?

What do you think **YOU** can create with computer programs?

Appendix E

Post-Questionnaire

The rest of this appendix includes the PDF of the post-questionnaire given to students at the workshop.

Post-Questionnaire

What types of things do you think computer programs can do? List everything you can think of.

What do you think the “Internet of Things” (or “IoT”) is?

What do you think **YOU** can create with computer programs?

Did you finish the app you wanted to make?

If no, why not? Did you run out of time? Could you not figure out how to do something? Do you think you unable to create it in App Inventor?

How hard was it to build your project?

If you wanted to use IoT to solve a problem you see in the world, what would it be?

Appendix F

Opcodes

Mnemonic	Opcode	Extra Bytes	Stack Usage	Description
`nop`	0x00	0	0	No operation should be performed and the PC moved to the next operation.
`aconst_null`	0x01	0	0	Push a null pointer onto the stack.
`iconst_m1`	0x02	0	0	Push the value -1 onto the stack.
`iconst_0`	0x03	0	0	Push the value 0 onto the stack.
`iconst_1`	0x04	0	0	Push the value 1 onto the stack.
`bipush`	0x05	1	0	Push the byte immediately following the instruction onto the stack.
`sipush`	0x06	2	0	Push the short immediately following the instruction (the next two bytes) onto the stack (as one long).
`dup`	0x07	0	1	Duplicate the value at the top of the stack.
`swap`	0x08	0	2	Swap the two top values on the stack.

Mnemonic	Opcode	Extra Bytes	Stack Usage	Description
`load_local`	0x09	1	0	Load the value from the local variable at the index given by the extra byte and put it onto the stack.
`load_global`	0x0A	1	0	Load the value from the global variable at the index given by the extra byte and put it onto the stack.
`store_local`	0x0B	1	1	Store the top value on the stack to the local variable at the index given by the extra byte.
`store_global`	0x0C	1	1	Store the top value on the stack to the global variable at the index given by the extra byte
`pop`	0x0D	0	1	Pop the topmost value from the stack.
`fadd`	0x0E	0	2	Add the top two values on the stack and put the result onto the stack.
`fsub`	0x0F	0	2	Subtract the top two values on the stack and put the result onto the stack.
`fmul`	0x10	0	2	Multiply the top two values on the stack and put the result onto the stack.
`fdiv`	0x11	0	2	Divide the top two values on the stack and put the result onto the stack.

Mnemonic	Opcode	Extra Bytes	Stack Usage	Description
`fneg`	0x12	0	1	Negate the top value on the stack and put the result onto the stack.
`iand`	0x13	0	2	Logical AND the top two values on the stack and put the result onto the stack.
`ior`	0x14	0	2	Logical OR the top two values on the stack and put the result onto the stack.
`ixor`	0x15	0	2	Logical XOR the top two values on the stack and put the result onto the stack.
`ifeq`	0x16	1	1	Jump if the top value on the stack is equal to 0. Jump the number of bytes given by the byte immediately following the instruction.
`freturn`	0x17	0	1	Return from the current procedure and put the top value on the stack onto the caller's stack.
`return`	0x18	0	0	Return from the current procedure with no return value.
`getstatic`	0x19	1	0	Get static field of the object given by the extra byte.
`invoke_virtual`	0x1A	1	1	Invoke a device method. The device to use is given by the topmost value on the stack and the method number is the extra byte in the program.

Mnemonic	Opcode	Extra Bytes	Stack Usage	Description
`invoke_static`	0x1B	1	n	Invoke the static method given by the extra byte. Static methods can be user-defined procedures or native methods (such as `sqrt`). The number of values on the stack used is the same as the number of arguments of the method invoked.
`device_create`	0x1C	2	0	Setup a device by allocating space on the heap and calling its constructor. The first extra byte is the device number in the program and the second is the index of the type of device it is supposed to be.
`device_getter`	0x1D	1	1	Get a device property. The extra byte is the device number in the program. The topmost value on the stack indicates which property to get and the result is put onto the stack.
`device_setter`	0x1E	1	2	Set a device property. The extra byte is the device number in the program. The topmost value on the stack indicates which property to set and the second topmost value is the value to which to set it.

Appendix G

Device Example: GroveButton

```
1  #include <stdint.h>
2  #include <appinventor/device/device.h>
3  #include <appinventor/interpreter/interpreter.h>
4
5  #ifdef MOCK
6  #include <appinventor/device/mock_arduino.h>
7  #else
8  #include "Arduino.h"
9  #endif
10
11 struct Button_data;
12 void Button_property_getter(struct Button_data *);
13 void Button_property_setter(struct Button_data *);
14 void Button_duty_cycle(struct Button_data *);
15
16 struct Button_data {
17   struct {
18     int8_t pin;
19     int8_t status;
```

```
20    } properties;
21    struct {
22      uint16_t press;
23    } events;
24  };
25
26  Device Button = {
27    DEFAULT_ALLOCATOR,
28    DEFAULT_CONSTRUCTOR,
29    DEFAULT_DESTRUCTOR,
30    (getter_t) &Button_property_getter,
31    (setter_t) &Button_property_setter,
32    (duty_cycle_t) Button_duty_cycle,
33    sizeof(struct Button_data),
34    /* n_properties */ 2,
35    /* n_events */ 1,
36    /* n_methods */ 0
37  };
38
39
40  void Button_property_getter(struct Button_data *data) {
41    int8_t property_index;
42    interpreter_pop_byte(&property_index);
43    if (property_index == 0) {
44      interpreter_push_byte(data->properties.pin);
45    } else if (property_index == 1) {
46      interpreter_push_byte(data->properties.status);
47    }
48  }
49
```

```
50 void Button_property_setter(struct Button_data *data) {
51   int8_t property_index;
52   interpreter_pop_byte(&property_index);
53   if (property_index == 0) {
54     interpreter_pop_byte(&data->properties.pin);
55     pinMode(data->properties.pin, INPUT);
56   } else if (property_index == 2) {
57     int32_t val;
58     interpreter_pop_long(&val);
59     data->events.press = (uint16_t) val;
60   }
61 }
62
63
64 void Button_duty_cycle(struct Button_data *button) {
65   if (digitalRead(button->properties.pin) == HIGH) {
66     if (button->properties.status == LOW) {
67       button->properties.status = HIGH;
68       interpreter_dispatch_event(button->events.press,
           VALUE_FROM_INTEGER(HIGH));
69     }
70   } else {
71     button->properties.status = LOW;
72   }
73 }
```

Bibliography

[1] Getting Started with the Arduino Web Editor - Arduino Project Hub. https://create.arduino.cc/projecthub/Arduino_Genuino/getting-started-with-arduino-web-editor-on-various-platforms-4b3e4a.

[2] Arduino - Home. https://www.arduino.cc/, 2018.

[3] Arduino - CurieBLE. https://www.arduino.cc/en/Reference/CurieBLE.

[4] Blockly | Google Developers. https://developers.google.com/blockly/.

[5] BlockyTalky - LPC. http://www.playfulcomputation.group/blockytalky.html.

[6] Chapter 6. The Java Virtual Machine Instruction Set. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html.

[7] FIRST LEGO League Teams. http://www.frc.ri.cmu.edu/girlsofsteel/our-team/first-lego-league-teams/.

[8] GraspIO Cloudio. https://www.grasp.io/.

[9] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications -ĂŤ Big Data, Scalable Analytics, and Beyond.

[10] Home · BlocklyDuino/BlocklyDuino Wiki. https://github.com/BlocklyDuino/BlocklyDuino/wiki.

[11] IFTTT helps your apps and devices work together. https://ifttt.com/.

[12] Tiffany Le, Hal Abelson, and Andrew McKinney. Controlling Bluetooth Low Energy Devices with MIT App Inventor. http://ai2.appinventor.mit.edu/reference/other/IoT.html, May 2016.

[13] Learn to program - Mindstorms LEGO.com. https://www.lego.com/en-us/mindstorms/learn-to-program.

[14] Micro:bit Educational Foundation | micro:bit. http://microbit.org/.

[15] MIT App Inventor. http://appinventor.mit.edu/explore/, May 2018.

[16] Node-RED. https://nodered.org/.

[17] Wireless and smart lighting by Philips | Meet Hue. https://www2.meethue.com/en-us.

[18] Raspberry Pi Foundation - About Us. https://www.raspberrypi.org/about/.

[19] Mitchel Resnick, Brian Silverman, Yasmin Kafai, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, and et al. Scratch. *Communications of the ACM*, 52(11):60, Jan 2009.

[20] Snap4Arduino. http://snap4arduino.rocks/.

[21] Snap4Arduino Arduino Sketch Generation. http://blog.s4a.cat/2015/06/09/Snap4Arduino-Arduino-sketch-generation.html.

[22] What is FIRST LEGO League? | FIRST. https://www.firstinspires.org/robotics/fll/what-is-first-lego-league.