Designing and Writing a JSON AST Interpreter for More Accessible and Extensible IoT Communication

Miles Higman

Under the direction of

David YJ Kim
Massachusetts Institute of Technology

Research Science Institute
July 6, 2024

Abstract

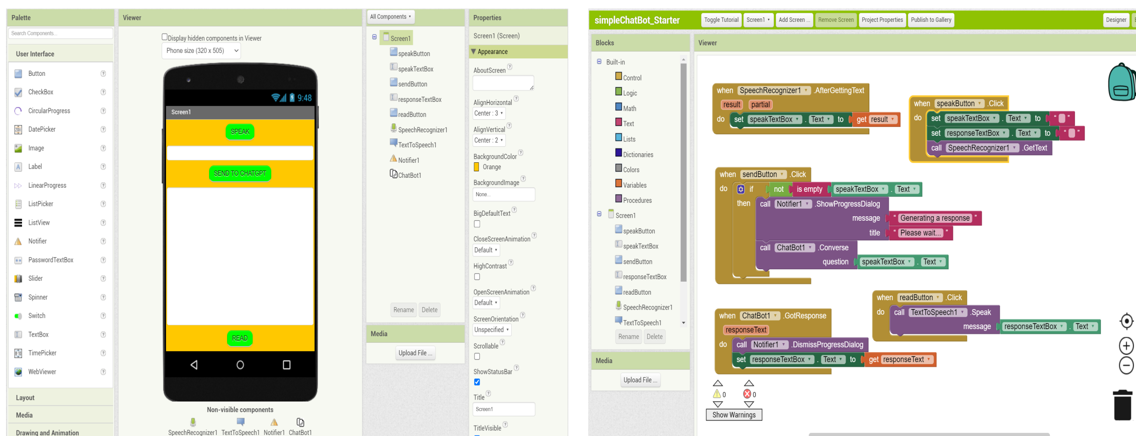
This project focuses on extending the functionality of the MIT App Inventor by developing an Arduino library that interprets and executes commands sent through JSON, which can be mapped to block code. Though block code has been used to control an Arduino before through interpreted and compiled means, this paper takes a more modular approach to its Arduino code. Fitting the program interpreting JSON into a library allows for Arduino code integration. First, we explain and elaborate on the details of the JSON schema used for storing commands, including its design principles. Then, we discuss the interpreter's program structure, like its recursion that supports type coercion, strategically-made variable buffers, and implicit type conversion between related data types. Finally, we explain the sketch that reads command inputs from both serial and Bluetooth Low Energy sources. The interpreter has been developed and successfully interprets programs with commands like loops, conditionals, pin writes, motor movements, arrays or 2D arrays, and serial or Bluetooth output. By lowering the barrier to entry to IoT, more students can create IoT applications to tackle real-world problems and learn the fundamentals of IoT at a younger age.

Summary

This project enhances the MIT App Inventor by creating a library that lets users control Arduino devices using simple visual coding blocks. This new feature supports functions like loops, conditionals, motor control, and Bluetooth communication. By making it easier to program Arduinos, the project aims to help more students, especially younger ones, learn about and create Internet of Things (IoT) applications. This lowers the barrier to entry, allowing students to start building impactful IoT projects sooner.

1 Introduction

Education has rapidly changed to respond to the ever-increasing prevalence of technology and STEM concepts. With a historic rise in new information to learn, the need for efficient and accessible education in our changing world is only increasing [1]. This paper focuses on extending the functionality of MIT App Inventor. MIT App Inventor is an open-source website that allows app development with block code to handle events and a drag-and-drop UI (User Interface) builder, as shown in figure 1. This intuitive interface enables children to create apps with real-world impact with minimal programming knowledge, teaching computer science fundamentals in the process [2].



(a) MIT App Inventor's UI builder. **(b)** MIT App Inventor's block code editor.

Figure 1: MIT App Inventor's interface with a simple chatbot app programmed into an app.

Knowing how people learn is important to making a better educational tool. One effective learning style for computer science is constructionism. As defined by Seymour Papert, constructionism has three optimal conditions for learning. Learners will learn best if they are in a shared classroom culture, given a meaningful project to complete, and have varied learning opportunities depending on learning style [3]. MIT App Inventor's lab focuses on the learning style defined by computational action: an extension of constructionism to computer science education emphasizing real-world impact [4]. To make this ideal classroom a reality,

hardware and software complexity should be simplified to effectively convey more abstract programming concepts to a younger audience.

However, the simplification of hardware complexity is more limited relative to software accessibility, even if this simplification is necessary. Some apps that learners want to build, like more formal projects, have a hardware component [5]. To more formally define these types of apps, IoT, or the Internet of Things, is an umbrella term denoting any computational devices connecting to the Internet or other devices in general, especially significantly smaller ones in everyday objects. An excellent example of an IoT device is a microcontroller that sends sensor data to an application to notify users when an oven is finished baking. To connect a microcontroller to MIT App Inventor, a learner may have to program the microcontroller and the app separately. This separation results in a novice programmer, who typically programs with blocks, potentially being forced to use C++ or C, both low-level and challenging programming languages, to program the microcontroller side of an IoT project. Flattening this type of difficulty curve should enable students to understand the concepts of IoT without understanding the caveats of low-level programming.

On the circuit side of abstraction for education, Arduino is an open-source project based in Italy, making microcontrollers accessible and affordable to end users [6]. Arduino's hardware capabilities are essential to understanding and contextualizing some of the later design decisions. Arduino also makes it accessible to program the microcontroller with `.ino` format files, an Arduino-specific extension of C++, with features such as the `digitalWrite`, `void setup()` and `void loop()` functions, and other, custom, open-source libraries that the end user can quickly implement.

We used an Arduino Uno R4 WiFi for this project to test the Arduino library and interpreter we had created. This board was chosen for its built-in communication protocols, which can prevent extra setup the end user has to do (like wiring) to establish communication between an app and a microcontroller. Also, at the time of this writing, the board is

priced at $27.50 per unit, which can be included in most schools' computer science budgets. The product's website states that our Arduino has Bluetooth Low Energy and WiFi capabilities, 32KB of RAM, and 256KB of Flash to upload compiled Arduino code [7]. Any more information about pinouts, which are essential for communication protocols, can be found on the product's datasheet and schematics [8, 9].

Previous work about abstracting away IoT complexities using an Arduino can generally be categorized into two methods: compile-time translation and interpreters. Compile-time translation involves taking block code, converting it to Arduino code, and compiling the pre-made file into a set of computer-readable instructions all at once. However, interpreters are different. In this context, an Arduino interpreter is a program running on the Arduino that receives commands *in real time* and executes the commands line-by-line. This process is similar to a conventional interpreter, where the computer executes commands line-by-line; however, there is an extra layer of communication and receiving commands. This project uses an interpreted method to control an Arduino.

Many services make block code that can directly compile a translated Arduino sketch onto an Arduino, such as Code Kit and Ardublockly [10, 11]. However, the on-demand functionality achieved from wireless communication may be a unique and advantageous approach for IoT education since students can see the real-time effects of their actions. Standard compilation can be impractical in IoT since a command representing a program subroutine, or a sectioned piece of code with a specific purpose,[1] could be a completely different subroutine relative to any other subroutine and can be sent anytime. Although JIT (Just-In-Time) compilers could be used, the fact that Arduino doesn't have an organized filesystem[2] and the library is written in a compiled language makes a JIT compiler impractical without SD cards to store compiled code, which cost money to buy in bulk at a high quality and could

[1] Subroutines are a superset of functions that do not need to return any data.

[2] Yes, Arduino does have Flash memory, but fitting a compiler into the amount of memory given isn't feasible.

make the library less accessible to socioeconomically disadvantaged school districts.

The interpreted style of programming microcontrollers to make IoT education more accessible is also not a novel concept. First, in 2018, Kathryn Hendrickson's master's thesis used a similar interpreted microcontroller method, using JVM bytecode to send code to and from the microcontroller [12]. Also, the creator of Scratch, another block-based language, is pursuing MicroBlocks, a project designed to send commands through Bluetooth to an ESP32 board, allowing learners to see the real-time output of their programs [13].

However, this project differs in two key ways. First, this project uses JSON to send commands. In addition to a different format for sending commands, this project was designed for open-source extensions of its functionality, which are contained in a library that can seamlessly integrate into other Arduino code written around it.

The main goal of this project was to contribute to the Arduino side of an educational tool for kids to program Arduino in a Blockly-based language and execute commands on demand wirelessly. We also want the library to integrate with existing Arduino code written in C++ so that students can progress from writing in simple Blockly code at the beginner level to writing in C++ at more advanced levels. To accomplish this, Blockly-generated JSON is converted to a shortened JSON format resembling an AST (Abstract Syntax Tree), or a tree-like structure that breaks a program into connected nodes to show code function more explicitly. Then, the Arduino reads this JSON and executes its commands. Figure 2 is a general flowchart tracking the lifetime of a given subroutine programmed in App Inventor under this framework.

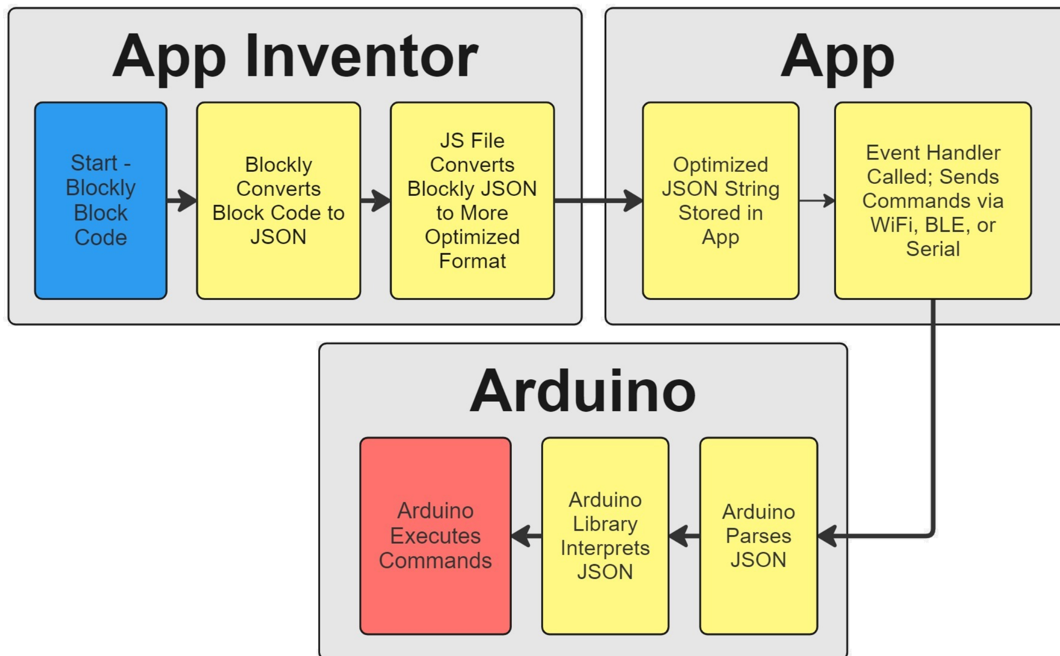


Figure 2: Broad, Flowchart-Style Plan of the Project. This paper covers the Arduino part of this plan.

2 Methodology and Design: Arduino JSON Interpreter

2.1 IoT Communication Protocols

Certain communication protocols are essential to contextualize and understand the intricacies of IoT and, by extension, abstract away specific caveats and nuances for educational use. When discussing IoT in this context, there are three prominent methods of communication between a microcontroller and some software-side applications: HTTP[3], Bluetooth Low Energy (or BLE), or simple wired serial communication.

HTTP, an application layer protocol built off of TCP for web-based data transfer, is one of the most apparent approaches to transferring data. Using the Internet's architecture,

[3]We aren't using SSL currently as there is little to gain by spoofing data off an educational project. However, SSL could become a configurable option.

we can send data to the IP address of the microcontroller through HTTP. HTTP protocol standards are complex and nuanced, and documentation for the protocol can be found here [14]. However, simple HTTP GET and POST requests with a JSON body will suffice for readable and effective communication.

Bluetooth Low Energy (or BLE for short) is another way of communicating between a low-power microcontroller and an application. This communication method is preferred as it is simple to set up, considering MIT App Inventor already has a BLE extension [15]. BLE acts like a bulletin board where central and peripheral devices read/write characteristics from a shared table through the Generic Attribute Profile (GATT) [16].

Finally, wired communication, using a standard serial communication protocol, which uploads all code that a microcontroller needs to run beforehand, is an effective communication strategy if values aren't constantly updated. However, this can also be used for constant and fast serial communication, given the device stays within the range of the cable.

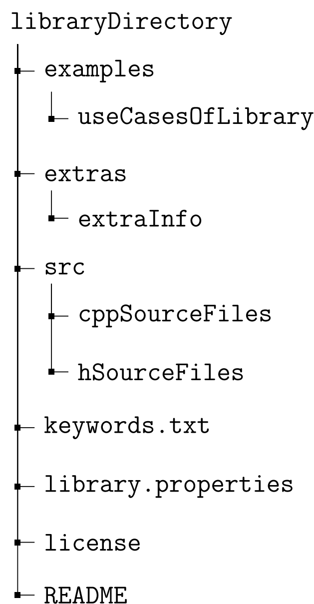


Figure 3: Directory tree for a general Arduino library.

2.2 Arduino Libraries

Arduino libraries are open-source extensions that abstract away many parts of a program using built-in Arduino and C++ functionality. Most Arduino libraries (including our parser) have a file structure like in figure 3. Everything in this file structure seems similar to C++ repositories from different sources or irrelevant to the project, excluding the files `keywords.txt`. The `keywords.txt` file allows Arduino's IDE to recognize datatypes, constants, and functions/methods for syntax highlighting [17]. Writing a library involves more conventions, such as using C++ classes for easy abstraction/documentation, namespaces, if needed, header guards for `.h` files instead of `#pragma once`, and code from multiple source code files into one header file for a more streamlined integration experience.

2.3 Arduino Parser Library Intro

Design Choices

The Arduino Parser library we developed utilizes the standard practices listed in section 2.2. However, our library was created with other design choices in mind to make its code more maintainable and extensible.

Enum types are used throughout the code to create integer-long variables that store critical information more readably, removing most magic numbers. After all, the value `Subroutine::FOR_RANGE` is more readable than `1`. Enum types can also make a measurable, non-aesthetic difference in code behavior because they have a different type than a regular `int` or `long`. Also, scoped switch-case statements switch between enum values and partition functionality between different enum values. Although long conditional blocks can be considered as a detriment to the readability of code under DRY[4] principles, the lengthy switch-case statement is probably needed, considering the variation between cases (i.e., there

[4] Don't Repeat Yourself

is little commonality to abstract away between a `for` loop and a `digitalWrite` command).

Besides conditionals, hash tables mapped to functions are an alternative solution. Still, considering memory constraints and the recursive nature of our solution, space on the call stack must be saved, meaning a function isn't optimal for this use case.

2.4 JSON Schema/Standards for Commands

When interpreting commands, defining standards for the communication stream that the parser will analyze is essential.

Why JSON?

JSON, or JavaScript Object Notation, is an eccentric decision to communicate instruction sets, considering some interpreted languages, like Python, are compiled to a less verbose bytecode before interpretation. However, JSON is a convenient way to express blocks for three main reasons. First, Blockly can convert blocks to JSON. This allows for a simpler time preprocessing and preparing commands that this Arduino can interpret. Also, JSON is a natural way to express data in communication protocols like HTTP (with or without SSL). Finally, JSON is more human-readable for fast iteration. Since we are in a very early project stage, fast iteration to see what works is essential. Optimizations to this command execution framework can come later. Given time constraints, JSON is the best option over other communication formats like XML or bytecode because of the above characteristics. XML is too verbose, and bytecode is less readable and requires more work to extract from Blockly blocks, leading to a long development time.

Schema Design Principles

With an established method of communication, standards allow for consistent representation of Blockly blocks in JSON. With this end goal in mind, three guiding design principles

exist in constructing an AST-like JSON schema. First, every key except variable identification information, type identification information, command codes, and literals takes another JSON object as input. Thinking of each JSON object as a block, the keys pointing to objects are the joints between code blocks. Although verbose, this principle is essential since each block should only infer the return type of the next block in the AST. For example, there is no way to tell whether the `double` needed in a parameter will come from a literal, variable call, or mathematical operation. Next, every object (or block) takes a command code or a byte of information that dictates the block's functionality and format. Finally, there are five different data types in the AST, classified by whether they can naturally be expressed as a `double`. The types `int` (4 bytes), `unsigned char` (1 byte), and `double` (8 bytes) are all classified as "int like" although they are all separate types. However, `bool` and `Array` are in their separate buffers due to their difference in function from the rest of the data types.[5] The `String` type wasn't included in this first version due to the heap allocations required to make an indefinitely long `String`, possibly causing heap fragmentation if overwritten with a different size [18].

Complete documentation of the standards for each command can be found in the Doxygen comments above each enum in the codebase, which is linked here.[6]

2.5 Parsing and Interpreting the JSON

With consistent JSON standards, taking the JSON string and executing commands based on its information is essential.

[5]Yes, a `bool` can technically be represented as an integer, but different operations are usually done on booleans than integers by the end user.

[6]This code is still under development and will be optimized at time passes.

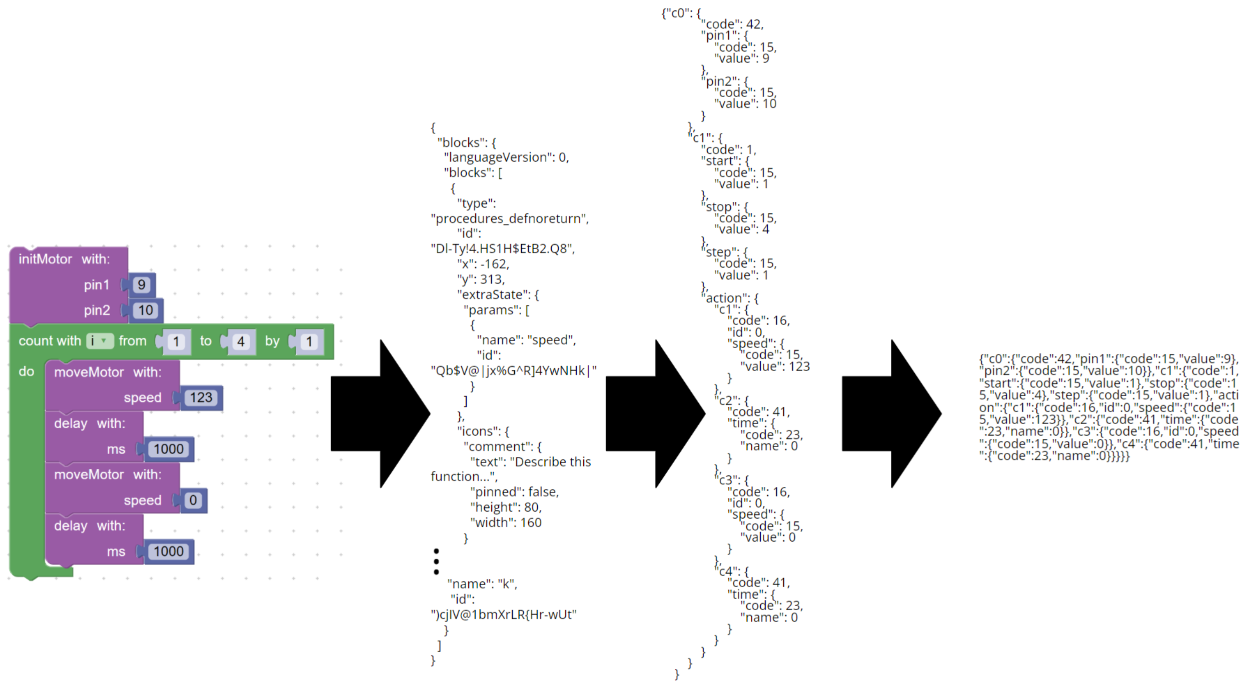


Figure 4: An example of block code made with Blockly being converted to a compressed JSON string for the Arduino Interpreter.

Dependencies

The interpreter we've developed has multiple dependencies for convenience. First, ArduinoJson is used to parse JSON. Next, Arduino's Servo library and AccelStepper handle different motor movements. Finally, ArduinoBLE is a dependency that helps facilitate writing to BLE characteristics.

Parsing

Parsing the JSON from a string to some C++ object that allows easy data access is difficult. However, ArduinoJson exists for parsing JSON with low memory usage, even supporting type coercion efficiently [19].

The Typing Problem

Even if ArduinoJson can support a form of type coercion, a problem arises in deciding what types to coerce on runtime. For example, if a value extracted from the JSON is `1`, should it be interpreted as a `bool`, `unsigned char`, `long`, `int`, or `double`? The answer here depends on the context in which the command was run. Considering the commands, their arguments, and, by extension, the expected types of each parameter are all defined beforehand, we can know a given argument's type from implicit type coercion, where the functionality of the previous node in the AST determines the type. However, due to the recursive calls to parse the commands, the return type needs to be configured to match the type of the argument. Recursion, albeit memory intensive, is an intuitive choice here as we are dealing with a tree. For a proof-of-concept early in development, a recursive traversal of an AST allows for a quicker development time and demonstrates the problem can also be solved iteratively, considering that every computing problem can be solved through lambda calculus [20].

Recursive Method Overloads

In the interpreter, type coercion is handled by method overloading. Overloading a method involves redefining a method with a different function signature. Different overloads were used to specify which argument needed to be what type, where the second parameter signifies the return type and acts as the default return value. However, a simple enum was used to save processing time for the `Array` object, which is implemented using ArduinoJson's `JsonArray`. Even though the enum is a different type that can differentiate a method overload, an enum is 4 bytes long and doesn't run a custom class construction (like `JsonArray` would), meaning it's a good choice for an optimized function parameter. Although a template could define a return type, it is an equivalent, less readable, and more error-prone[7] implementation. We would have to specialize the template since each type has different commands, resulting in

[7] Templates in an open-source project can have the chance to be buggy due to their unintuitive syntax.

almost equivalent code to method overloading.

Control Statement Handling

Control statements have a different command structure than other types of functionality. Like block code, the blocks in the control block connect differently from other blocks. The difference in behavior could be interpreted as taking in a list of commands or scope, just like the program's global scope. Consequently, if a list of commands, expressed in JSON, is a scope, then the function used to parse the global scope can be reused for any scope.

Variable Handling

The program is structured such that each type ("int like" types, `bool`, and `Array`) has a stack-allocated 20-element buffer for global variables such that upon JSON command creation, each variable is assigned an index in the buffer in the order they appear. However, this buffer system has one caveat. All "int like" types share the same buffer, which has spaces of type `double` (the biggest of the three "int likes"), with certain parts of the buffer "reserved" (except when executing a `VARIABLE_CALL` instruction) for particular data types. For example, if the end-user was given 20 variables of each data type, the "int like" buffer would allocate 60 spaces for type `double`, with spaces 0-19 allocated for type `long`, spaces 20-39 allocated for type `double`, and 40-59 allocated for type `unsigned char`. Finally, to call a variable, give the name of the space where the character is in the buffer, given a specific data type defined by the type coercion mentioned in section 2.5. Using predefined buffers for variables keeps RAM usage consistent and prevents heap fragmentation that could occur for solutions involving heap allocation.

Iterator Variables

Iterator Variables work very similarly to the variable handling mentioned above, except for the implementation of the `VARIABLE_CALL` and `VARIABLE_SET` instructions. First, iterator variables are already set for the user in buffers in the order of their "depth" relative to other for loops. If the for loop is in the global scope, the depth is 0. If the loop is in the scope of another loop, the depth is 1. Generally, if a for loop's scope is entered, the depth variable for each typed iterator variable buffer increments, with the same variable decrementing when the for loop ends. To call an iterator variable, use the `VARIABLE_CALL` instruction with either the "iter" or "iter_range" tag. Then, specify a two-character identifier with the following format: 'i' + the depth of the iterator variable buffer, where 'A' = 0, 'B' = 1, etc.[8]

3 Results: Unit Tests

After implementing the bulk of the interpreter, it is essential to demonstrate its functionality and that it's running as intended. Unit tests, or inputs with expected behavior if run by the program, are an excellent tool for testing a program.

However, to keep the library general, we left the communication methods to obtain a JSON string for the parser on a by-sketch basis. The sketch enabling the two main communication methods implemented will be located in the `examples` directory in the library.

3.1 Connecting to Serial

When taking input from Arduino's default `Serial` class (more specifically, UART communication), the program reads the Serial buffer byte by byte, excluding whitespace and the null terminator (`'\0'`) character, until a completed command is in the 3000-character

[8]The more formal way of saying this is that the index is `c - 'A'`, where `c` is a given character and subtraction is defined as the difference between both characters' Unicode values.

command buffer, the buffer can't hold the commands[9], or the Serial buffer empties. It will keep reading any information from the Serial buffer until it has a completed command, which it will give to the parser to execute.

For output, the `Serial.print` or `Serial.println` methods output to the host device.

3.2 Connecting to Bluetooth

When taking input from BLE, we used the ArduinoBLE library, which provides an interface to interact with the Arduino R4's built-in BLE module [21]. Similarly to Serial command parsing, the sketch taking BLE inputs reads from any of the three packets available, stopping on one of the three conditions specified in section 3.1.[10]

BLE output isn't as straightforward as Serial output since a `BLEStringCharacteristic` object (the object needed to facilitate communication between devices) isn't defined in all scopes like the `Serial` object. However, we can access the reference in the class methods by storing a `BLEStringCharacteristic` reference per new command read by the sketch as a class attribute. This process allows the parser library to access the object.

3.3 Executing Unit Tests

To implement unit tests for the program, we first transformed it into JSON format, excluding whitespace. An efficient unit test has many different functionalities embedded into the program, as defined by command codes, to test more program parts concurrently.

Using unit testing, we confirmed functionality for variables, loops, conditionals, pin writes, motor movements, and outputs via BLE and serial, turning the idea into a Minimum Viable Product, or a project with the minimum amount of functionality to be viable.

[9]In this case, the program will empty the buffer and keep reading.
[10]Replacing the Serial buffer with the values in any of the three writing packets.

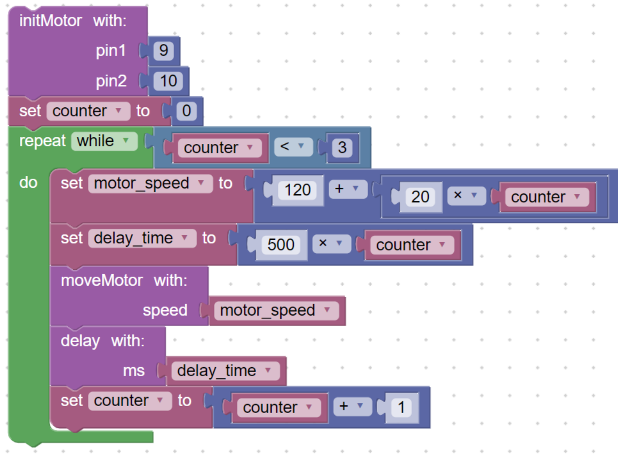


Figure 5: A representation, made in a Blockly demo app, of simple code to accelerate a motor. This was a one-unit test that tests many block subroutines.

4 Discussion

In this project, we implemented various interpreter functionalities on an Arduino. This project's scope is the baseline functions of a programming language, allowing for reasonably complex programs (for example, the program in figure 5) to be sent through Bluetooth and executed almost instantly, considering a human timescale. Theoretically, with enough memory, the tested functionalities can have many applications, such as sensing and interpreting data as well as controlling a robotic system. However, more work must be done to complete the full project, as seen in figure 2.

4.1 Limitations

JSON: Too Verbose

JSON, although being the communication format of choice in this project for reasons outlined in section 2.4, can be space inefficient. For example, the minimum amount of space

needed to store a block with no additional arguments would be 7 bytes.[11] The amount of formatting (not informative) characters in a JSON string will only increase when nesting JSON and adding arguments (especially with unoptimized key names). The size of the JSON string, while usually inconsequential, is vital in this project, considering that only 32,768 characters, including global variables, can be stored in the Arduino's RAM.[12] Therefore, the length of programs interpreted by the Arduino is limited. For example, the code in figure 5 is slightly over a quarter of the maximum command length without accounting for the compression of keys or other compression methods.

Limited Functionality

Another limitation of the library would be the need for more testing and development of different commands, like string commands, WiFi, and more advanced mathematical operations, like trigonometry and exponents. However, considering the codebase is built with readable Doxygen comments for each implemented command, these functionalities should be quick to program and implement.

Recursion and Memory

The recursive process of interpreting commands is a notable limitation. In regular computers, the amount of space each recursive call takes is tiny compared to the space allocated in the call stack. This size disparity would allow for many more recursions than reasonably necessary. However, since the JSON storing the program can often become nested, Arduino's limited memory makes a stack overflow when interpreting complex programs possible.

[11] This figure assumes that the key for the command code is a single character, which may not be the case for readability reasons.

[12] Assuming ASCII encoding from an `unsigned char`

4.2 Next Steps

Since this project is in its early stages, it is vital to have a coherent and detailed plan for future development milestones.

Optimizations and Additions to Current Code

As the project grows, many commands discussed in section 4.1 should be implemented to expand the project's use cases. However, optimizing the current product is crucial for users to use new features effectively. As far as optimizations go, minimizing the number of bytes sent per action in a command is the most pressing optimization due to the small amount of RAM on an Arduino. A quick way to optimize JSON length would be to use 1-2 letter key names, which cuts the 792-byte compressed JSON from the commands representing figure 5 to 563 bytes, a 28.91% decrease in byte count.

Possible Overhauls

As established in the previous section, most optimizations aim to compress the number of bytes needed per command. One space-efficient way to send information is bytecode, where each byte represents a command involving a specified amount of bytes after the initial command. To reduce the byte count, we theoretically could make custom standards for each different bytecode and utilize the already built program structure to interpret the bytecode. Also, recursion-related memory concerns can be optimized using an iterative process to interpret and execute commands, requiring significant changes to the codebase.

Beyond the Arduino

As shown in figure 2, the Arduino firmware side of a more general effort to connect MIT App Inventor and Arduino through interpreted means only composes one of three separate environments. Two main features must be implemented at a high level, excluding

the Arduino interpreter. We need to write an MIT App Inventor extension that adds any blocks representing Arduino functionality into MIT App Inventor's interface, converting each subroutine into our JSON AST schema or other formats via an automated script written in JavaScript since JSON is a native type in JavaScript.

5 Key Takeaways

This project was a crucial step in a larger initiative to apply real-time, interpreted code execution to IoT education using MIT App Inventor. Being fully open-source and extensible, the educational resource can be customized in a decentralized manner per any educator's needs. Abstracting away low-level and complex IoT parts allows learners to make actionable changes in the world while learning.

6 Acknowledgements

I would like to acknowledge David Kim, my mentor for this project, for helping me formulate my ideas in the presentation and providing crucial resources for the project. Also, I would like to thank Hal Abelson, the PI of my lab, for allowing me to work with MIT App Inventor's lab and building many of the paradigms for computer science education that my work is based upon. As well as my lab, I would like to express my gratitude to Marissa Sumathipala for giving me detailed feedback regarding my paper. Also, I would like to acknowledge my friend Nicolas Lu, with who I spent many nights in the library, where we worked on our projects. Also, I would like to thank my sponsor, Shikui Yan, for sponsoring me to conduct research at MIT's campus. Finally, I thank CEE and MIT for facilitating this opportunity.

References

[1] Kennedy, Teresa J and Odell, Michael RL. Engaging students in STEM education. *Science education international*, 25(3):246–258, 2014.

[2] E. W. Patton, M. Tissenbaum, and F. Harunani. MIT app inventor: Objectives, Design, and Development. *Computational thinking education*, pages 31–49, 2019.

[3] F. R. Sullivan. *Creativity, technology, and learning: Theory for classroom practice.* Routledge, 2017.

[4] M. Tissenbaum, J. Sheldon, and H. Abelson. From computational thinking to computational action. *Communications of the ACM*, 62:34–36, 02 2019.

[5] G. Qiu, Z. Zheng, Y. Xu, and Y. Deng. Development of personalized iot mobile applications using mit app inventor 2. In *Proceedings of the 4th International Conference on Computer Science and Software Engineering*, pages 163–167, 2021.

[6] Arduino. Arduino. `https://www.arduino.cc/`, 2024. Accessed: 2024-07-29.

[7] Arduino. Arduino UNO R4 WiFi. `https://store-usa.arduino.cc/products/uno-r4-wifi`, 2024. Accessed: 2024-07-16.

[8] Arduino. *Arduino® UNO R4 WiFi Product Reference Manual*, March 2024. SKU: ABX00087, Modified: 18/03/2024.

[9] Silvio Navaretti. UNO R4 WiFi Schematics. `https://docs.arduino.cc/resources/datasheets/ABX00087-schematics.pdf`, 2023. Version 1.0, Date: 23 August 2023.

[10] Carlos Perate. Ardublockly: Visual Programming for Arduino. `https://ardublockly.embeddedlog.com/`, 2015. Accessed: 2024-07-18.

[11] EduKits International Pty Ltd. Code Kit: Intuitive Drag-and-Drop Block Coding for Arduino. `https://edukits.co/code-kit-app/`, 2024. Accessed: 2024-07-18.

[12] K. E. Hendrickson. *Writing and connecting IoT and mobile applications in MIT App Inventor.* PhD thesis, Massachusetts Institute of Technology, 2018.

[13] John Maloney. MicroBlocks. `https://microblocks.fun/`, 2024. Accessed: 2024-07-18.

[14] MDN Web Docs. Resources and Specifications. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources_and_specifications`, 2024. Accessed: 2024-07-05.

[15] MIT App Inventor. MIT App Inventor Extensions. `https://mit-cml.github.io/extensions/`, 2024. Accessed: 2024-07-05.

[16] C. Gomez, J. Oller, and J. Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors (Basel, Switzerland)*, 12(9):11734–11753, 2012.

[17] J. Purdum. *Beginning C for Arduino*. Apress, 2nd edition, 2015.

[18] Arduino. Memory Guide. `https://docs.arduino.cc/learn/programming/memory-guide/`, 2024. Accessed: 2024-07-30.

[19] B. Blanchon. ArduinoJson. `https://arduinojson.org/`, 2024. Accessed: 2024-07-20.

[20] Raul Rojas. A Tutorial Introduction to the Lambda Calculus. `https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf`, 1997. Accessed: 2024-07-30.

[21] Arduino. ArduinoBLE Library. `https://www.arduino.cc/reference/en/libraries/arduinoble/`, 2024. Accessed: 2024-07-25.