Blocks O’ Code: An Interactive Platform Coding Learning Tool

John Gierlach, Robert Buch, Tyler Goldsmith, Nandhu Jani

University of Central Florida, College of Engineering and Computer Science, Orlando FL

Abstract – This paper outlines the design for “Blocks O’ Code,” a project meant to serve as a learning aid and platform for students to learn to and experience fundamental coding concepts (such as variables, conditionals, and loops) in a tactile manner. This is accomplished via blocks which hold 8-bit addresses, each corresponding to a unique code phrase that can be combined with other phrases to create more complex codes. The placement of these blocks on the platform’s board is read, compiled, and run in real-time with the use of an ESP32 and Raspberry Pi, with the resulting code output being shown and updated on an LCD screen. This paper breaks down this design into three main categories: (1) Hardware Design (dealing with the design of the PCBs of the code blocks and platform board), Embedded Software Design (dealing with communication between the board and the Pi), and High-Level Software Design (dealing with block address mapping to code segments, code compilation, and output processing).

I. INTRODUCTION

A. Project Overview and Motivation

Learning programming is becoming increasingly important for young students, serving as a baseline for future academic and professional pursuits in Computer Science. The larger challenge for instructors lies in fostering a comprehensive understanding of programming as a language rather than merely introducing them to the syntax of coding. This requires a shift in problem-solving techniques, where students learn to integrate logic with creativity to devise novel solutions.

"Blocks O’ Code" is designed as a solution to the issue of sustaining interest in programming for young students. Drawing inspiration from the intuitive way children engage with physical toys, this project seeks to transform the coding learning process into an interactive and tangible experience. The premise is simple yet innovative: allow students to construct code physically using blocks, thereby rendering abstract concepts into a format that is both accessible and engaging.

The motivation behind "Blocks O’ Code" was to create a tool that leverages the diverse capabilities of our team and addresses a clear need in contemporary education. Traditional digital coding tools often fail to captivate young learners, lacking the engagement necessary to stimulate sustained interest. By contrast, "Blocks O’ Code" proposes a tactile approach, wherein students manipulate physical blocks representing different coding elements—such as loops, variables, and functions—on a specialized board. This hands-on interaction is complemented by real-time feedback provided through an integrated system comprising an ESP32 and Raspberry Pi, which reads, compiles, and displays the resulting code on an LCD screen.

B. Technical Overview

The "Blocks O’ Code" project is underpinned by three technical pillars:

1) **Hardware Design**: The focus here is on the physical components—the coding blocks and the interactive board. Each block is embedded with an 8-bit address, signifying a unique code segment. The design priorities include durability to withstand frequent use and precision to ensure accurate interaction with the board.

2) **Embedded Software Design**: This aspect entails the development of software that facilitates communication between the physical blocks/board and the microcontrollers (ESP32 and Raspberry Pi). It is crucial for interpreting the spatial arrangement of blocks into executable code, thereby enabling the real-time feedback mechanism that is central to the "Blocks O’ Code" experience.

3) **High-Level Software Design**: At this level, the focus shifts to the software responsible for mapping the block addresses to specific code segments, compiling the code, and processing the output. The high-level software ensures that the students’ physical arrangements translate into coherent, executable code, which is then reflected on the LCD screen, including error messages and execution results.

II. SYSTEM COMPONENTS

A. ESP32

The ESP32 microcontroller chip stands as a comprehensive system-on-chip (SoC) tailored to excel

in various domains such as IoT, wearable tech, and smart home appliances, while prioritizing low-power consumption. Its repertoire includes an array of communication protocol peripherals, 2.4 GHz WiFi, robust clock speeds, LCD interfaces, hardware security features, timers, and more. This versatile chip finds applications in low-power IoT sensors, RGB LED lighting control, POS machines, energy consumption monitoring, and beyond, making it adaptable to diverse needs. Leveraging its features for data processing and clock generation in code blocks can prove highly beneficial. For our purpose we plan to utilize this chip due to its high clock speeds, low-power capabilities, low cost, GPIO pins, and large amount of on-chip memory. This processor, compared to many others, proved to be the best fit for our purposes for our project [1].

B. Raspberry Pi Zero 2 W

The final proposal for compiling the user-generated C code involved using a Raspberry Pi variant with significantly lower power consumption. This Raspberry Pi would primarily handle two tasks: compiling user code and maintaining communication with the ESP32. Equipped with essential GPIO pins, this version of the Raspberry Pi seemed suitable for the job. The Raspberry Pi Zero 2 W, known for its minimal power consumption of around 2.5 W, emerged as an ideal choice, aligning well with the system's power requirements. Its power efficiency also makes it compatible with USB A connections, further enhancing its suitability. Additionally, the Zero 2 W boasts official support for the Ubuntu Linux distribution, a crucial feature given the project's reliance on Ubuntu for compiling programs. This support reduces the risk of encountering significant issues with the operating system. Furthermore, priced at $15, the Zero 2 W is a budget-friendly option compared to other Raspberry Pi models, contributing to significant cost savings for the project. Overall, the Raspberry Pi Zero 2 W was selected because of its communication protocols, power consumption, and C-code complications [2].

C. I2C LCD1609 Screen

After thorough evaluation, we have determined that the I2C LCD1609 display is the most suitable option for our requirements, primarily due to its dimensions, low GPIO requirement, cost-effectiveness, and power consumption. Despite the adequacy of other displays, we aimed to minimize GPIO usage considering the finite availability on the ESP32. Furthermore, ensuring adequate screen size for comfortable readability without straining or necessitating proximity was imperative. With dimensions of 5 inches in length and 2 inches in width, the chosen screen provides an optimal balance for practical use. Additionally, the I2C communication protocol offers enhanced flexibility, enabling straightforward integration with the ESP32 through readily available libraries and support for ASCII characters, facilitating the display of pertinent information such as code output or errors to the user. This LCD will be perfect how our use due to its low pin requirements, power usage, and dimensions [3].

In summary, these components will all cooperate with each other to take in data from the grid of blocks which is done by the ESP32, which will then send that data to the Raspberry Pi for C-code compilation and finally display the result to the LCD screen. These components allowed for a lower power system design that enabled us to tailor our efforts for a flexible and expandable product for teaching younger students the fundamentals of programming. Others section will go further into detail describing how each of these independent subsystems will function and be integrating together. Figure 1 shows the overall system flow diagram.

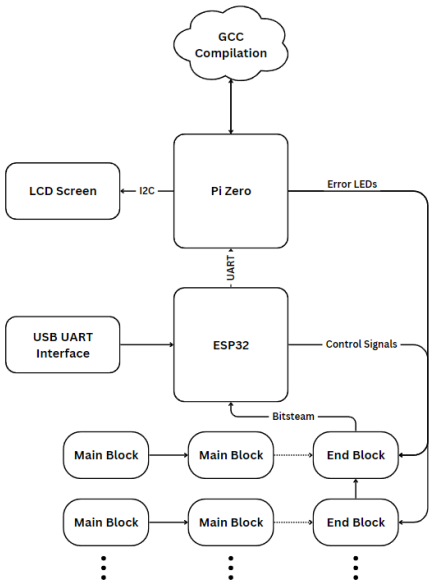

Fig. 1: System Flow Diagram for Blocks O' Code

III. PROPOSED SOLUTION

A. Hardware Summary

The hardware design for Block o' codes primarily focus on translating user-oriented blocks into binary

representable data, then moving this data in a predictable manner for processing by the microcontroller. In the end the most reliable way to move data from the board into the microcontroller for compilation is to represent each unique block with a unique 8-bit binary address, then to use a series of shift registers to move the data in a serial manner horizontally across the board, then vertically into a single pin on the microcontroller.

The major hardware component that facilitates the movement of data is the control PCB that is stored within each clock, which will serve to store the unique address and shift this address into its neighboring blocks on the board. To allow communication between the blocks placed on the boards we designed dataflow PCB grid with physical connectors that are complementary to connectors on the blocks. This grid creates connections between the blocks, allowing the serial passage of data, as well as the sharing of control and power signals from the microcontroller. The final hardware component is an embedded board that creates a connection between the ESP-32 to the grid to allow for control of the dataflow and creates a connection between the ESP-32 and the Raspberry Pi 0 to allow for compilation of the received data.

To allow realization of our dataflow pipeline we have designed two iterations of the block design: a main and an end block. Each block (both main and end blocks) contains a dipswitch to hold its unique address, as well as a horizontal shift register that will combine with the other blocks in the grid row it is placed in to make a long shift register. For example, a row populated by 3, 8-bit blocks can be viewed as a single 24-bit shift register with every series of 8-bits within the register loaded with the unique addresses of the blocks. To move the data into the microcontroller, at the end of each line (indicated by end characters such as ‘;’, ‘{‘, and ‘}’) is an end block, which contains all the components of a main block, but with the addition of a vertical shift register, a multiplexer to control the input of this register, and a flag pin to indicate that the line is finished and needs power from the microcontroller. The vertical shift register takes input according to the state of the multiplexer. In one state this register takes horizontal input from its row, and in the other state it takes vertical input from the end block below it (or 0 if nothing is below). Because of this when the multiplexer passes vertical column data this register (like the horizontal registers) combines with the vertical registers in the end blocks of each line to make a large vertical shift register which feeds into the microcontroller.

With all these designs in mind the overall process for moving data into the microcontroller follows the following steps: Unload data from the dipswitch into the horizontal shift register, shift 8 bits from the horizontal to vertical shift register, shift all the vertical data into the microcontroller, then repeat the process until all columns have been read.

B. Software Summary

The software for Blocks O’ Code executes across two primary processors, the ESP32 microcontroller and the Raspberry Pi Zero 2 W. The ESP32 handles the collection of data from the block elements of the system. The ESP32 detects the number of rows of blocks that must be processed and collected and then proceeds to collect this data and store it in a 2D array so that it can be restructured as a C program. With this data collected, the ESP32 opens its UART connection with the Raspberry Pi Zero 2 W and proceeds to transfer all of the block data in the order of the 2D array so that it is correctly ordered when received by the Raspberry Pi Zero 2 W. With the block data now on the Raspberry Pi Zero 2 W, it executes the process of mapping each of the bytes to a unique piece of C code. With all of the bytes mapped to C code, the entire C file is constructed to match the design created by the user on the board. Next the Raspberry Pi Zero 2 W compiles the generated C code and captures the output of the compiler as necessary. If it is determined that the compiler has failed based on the returned output, the Raspberry Pi Zero 2 W prints an error message to the connected LCD. On the other hand, if it is determined that the compiler succeeds, the Raspberry Pi Zero 2 W will then execute the C code from the Linux command line and read those results in so that they may be printed to the LCD display. On completion of printing to the LCD display, both boards will check if the other board is ready to begin another cycle of data collection, transferring, and processing. If it is determined that both boards are ready, the system restarts the process, and the steps repeat. Otherwise, each board will idle until both boards are ready to execute the process again.

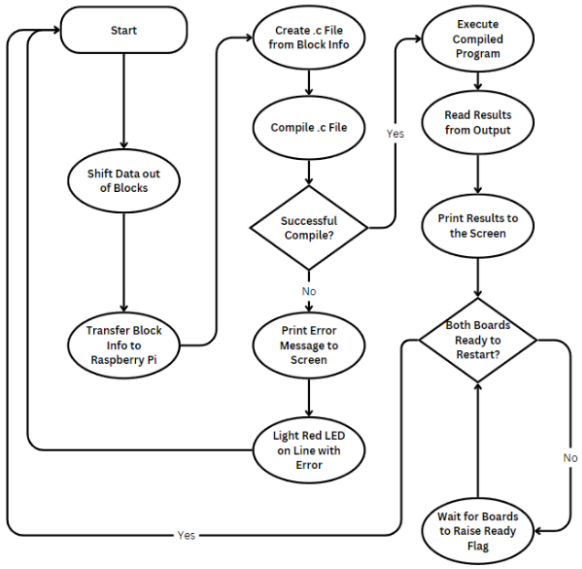

Fig. 2: Software Flow Diagram

IV. HARDWARE DESIGN

A. Grid Board

The main responsibility for the grid PCB is to create wired connections between the blocks and the ESP-32. Similarly to the blocks there are also two different types of grids PCBs; main grid and end grid. Each grid has vertical and horizontal solder points to create connections with other grid segments to build up a larger system. Our system is a 4x4 grid, which means we utilized 16 main grids arranged in a 4x4 square, and 4 end grids at the end of each row.

The end grid is the PCB placed at the end of each row and is responsible for passing signals from the ESP-32 to the grid. It creates a vertical signal rail that is always active if the system is powered. This rail includes VCC, GND, CLK, and dataflow control signals. These PCBs also include control logic to pass signals from the vertical rail into each row only when an end block is present in the row. This control logic activates each row according to an active low signal from the row that is pulled high (pull-up resistor) when no end block is present, and pulled to GND, when a block is placed in the row (as indicated by a connector pin unique to the end block that is grounded when placed).

The main grid PCB, Figure 4, makes up most of the overall grid. Each main grid includes a spot for a single magnetic connector that interfaces with a complementary connector on the base of the blocks. This PCB is responsible for pulling the signals from the connector and connecting them to a horizontal signal rail routed from the ESP-32 through the end grid segments. Additionally, these PCBs include both vertical and horizontal serial data lines. The horizontal serial line simply moves output serial data from one block to its neighbor. The vertical data line, although functionally like the horizontal, is realized differently because the ends of each row may not line up precisely. Because of this the vertical data line creates a horizontal rail across the entire row and connects to a pin on each connector that only makes connections with placed end blocks (on the end blocks this signal is pulled down to handle the case in which no end blocks are in the rows below).

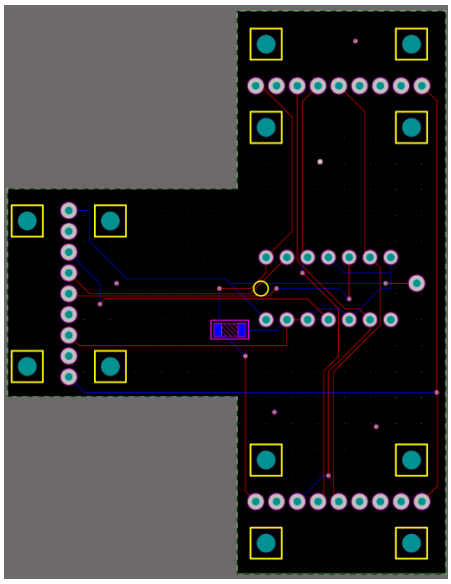

Fig. 3: End Grid Segment PCB

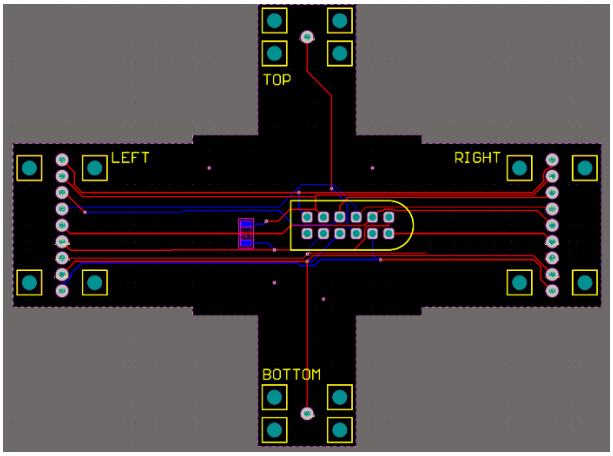

Fig. 4: Main Grid Segment PCB

B. Main Block

The main block, Figure 5, is responsible for unloading unique addresses into a horizontal shift register, then moving this data horizontally into a vertical register in an end block. To realize these capabilities the control board includes an interface between the board and the magnetic connector at the base of the block, an 8-bit configurable dipswitch to hold the address, and an 8-bit horizontal shift register. The connector includes power, ground and clock signals, in addition to two control signals from the ESP-32 to control whether the block will unload it's data into the shift register or move its data into its horizontal neighbor.

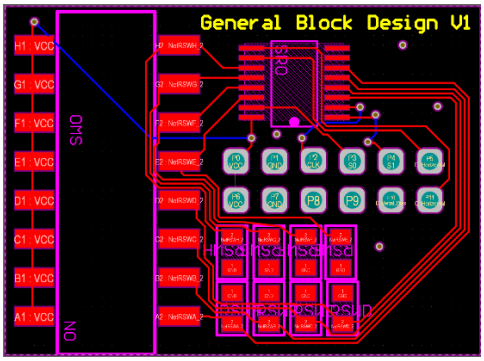

Fig. 5: Main Block Design

C. End Block

The end block, Figure 6, is responsible for moving horizontal data from the row up into the microcontroller. This means that it has both a horizontal functionality to receive the horizontal data, and a vertical functionality to shift this data up into the microcontroller. This means that it includes all the components of the horizontal shift register (magnetic connector slot, address dipswitch, and horizontal shift register). In addition to these the design includes a vertical shift register, as well as a NAND gate multiplexer to control whether the vertical shift register receives horizontal or vertical serial data. In addition to this the end block utilizes an unused magnetic connector pin and connects it to ground to flag when the row is active and should receive power and includes a pull-down resistor on the vertical data pin to shift in all 0's when no end block is placed below it.

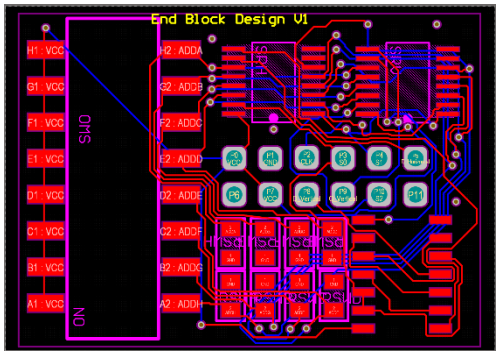

Fig. 6: End Block Design

D. Embedded Board

The embedded board is responsible for connecting the ESP-32 to the Raspberry Pi 0 and to the grid. The interface for the ESP-32 and the raspberry pi is a simple one that includes a flag for when the Pi 0 and the ESP32 have completed their tasks, and a UART connection to allow communication between the two. The interface between the ESP-32 includes control signals (CLK, S0, S1, S2) and the serial data shifted out of the grid. Additionally, the board creates connections between GPIO on the raspberry Pi to error LEDs on the board to indicate which lines are incorrect when an error occurs. In addition to these interfacing features, this board is also responsible for powering the entire system over a microUSB connection, which also doubles as a programming interface for the raspberry Pi 0.

E. Power Supply Design

The power distribution system for Blocks O' Code is simple and purely DC. There will be two main rails that will supply 5V and 3.3V for various components that will be described later in this section. The maximum current we are planning to supply is 2A, giving us a maximum power consumption of 10W. USB 2.0 will be used to connect the power rails and data rails to the embedded board and thus to other devices. The components using the 5V rail will be the LCD screen and Raspberry Pi Zero 2 W. The components using the 3.3V rail will be the ESP32, grid/blocks, and USB-UART interface. Based on the recommended currents that each of these devices require, it is expected that our project will take ~4.5W of power which is well within the 10W limit that is established. Figure 7 shows the high-level power system design.

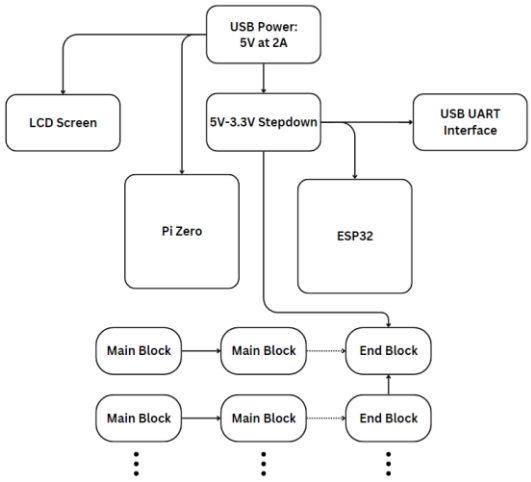

Fig. 7: Power System Design for Blocks O' Code

V. EMBEDDED SOFTWARE DESIGN

The embedded software of Blocks O' Code is straightforward as it contains two main responsibilities, providing the control logic signals to the grid and collection the data from the grid via the user connecting the blocks. The software was designed using C++ and Platform IO to flash the program to the ESP32 for proper execution. Once the ESP32 completes these two tasks, it will send the data that it collects to the Raspberry Pi. The following sections will go further into detail to understand how each of the steps are designed in the software.

A. Control Logic

The grid design for our projects requires several control pins as described in the previous section, these control signals will be generated by the ESP32 as specific times such that the grid is able to shift, he data of each block correctly based on the End-block and Main-block characteristics. The signals that the ESP32 will be sending to the board will be S0, S1, S2 for shifting logic, CLK for a 16MHz clock generation for the shift registers and digital gates on the blocks/grid, and lastly a receiver pin labeled Q_Rx which is the serial information sent to the ESP32 from the blocks on the grid.

B. Data Collection

The data collected by the grid will be received by the ESP32 via the Q_Rx pin that will take in serial information. The state is specifically collected during the state of the embedded architecture in which the End Blocks will vertically shift the information contained in a shift register that will send the entire columns address. The address it will send is 8-bits long, which indicates what kind of block it is (i.e. X, Y, Z, while(, etc.). To collect the data of all the segments in the grid, it is best if the data is stored in a 2D-array where is index is a one-to-one representation of the grid with its corresponding blocks. Once the data collection process is complete

C. Embedded Software Architecture

The ESP32 program is written in the form of a state machine that has five main stages being IDLE, SHIFT, COUNT_ROWS, SHIFT_H, and SHIFT_V. These steps determine the state of the S0, S1, and S2 shifting logic pins by sending either a HIGH or LOW signal through them via the ESP32's GPIO pins. During each of the stages, there will be a function to drive a clock pulse at a frequency of 16MHz with a period of 62.5ns. This clock frequency is fast enough to process the data in real-time and safe for all the digital logic ICs used on the grid/blocks.

1) IDLE State: The IDLE state will ensure no shifting is happening for a given cycle.

2) SHIFT State: The SHIFT state will shift the value of the dipswitches into a shift register that will be stored until it is ready to shift down to the next block.

3) COUNT ROWS State: The COUNT_ROWS state is responsible for getting the number of rows so that the software can perform enough SHIFT_H and SHIFT_V states to get the data of every block within the grid (it's important to note that it is necessary to perform an IDLE and SHIFT state after COUNT_ROWS is performed). Once the number of rows is found, the SHIFT_H and SHIFT_V states will be called the number of times equal to the number of rows.

4) SHIFT_H: State: During the SHIFT_H state the blocks will shift the data contained in the shift registers down their row.

5) SHIFT_V State: The SHIFT_V state will shift the data held within the End blocks to the ESP32 via the Q_Rx signal via serial communication. The ESP32 will store this data into a 2D-array and once the program reaches the end of the grid, it will exit the SHIFT_H/V loop and return to IDLE to send the 2D-array to the Raspberry Pi Zero 2 W for compilation.

The embedded software is the main translator for from the ESP32 to the grid and is essential for getting the user's data via the blocks. The previous sections give a detailed description of how the software is

defined and designed. Figure 8 shows a state diagram of the embedded software architecture.

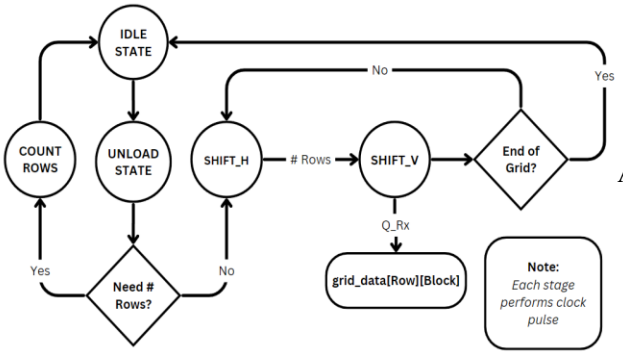

Fig. 8: Embedded Software Architecture

V. HIGH-LEVEL SOFTWARE DESIGN

On the side of the Raspberry Pi, when Blocks O' Code is active, the Pi runs a Python program which opens its own end of the ESP32-to-Pi serial connection using the PySerial library, uses that connection to collect the bitstream generated by the ESP32, parses it into readable code using a dict, inserts the generated code into a C file, runs that C file, collects the outputs, parses the output for readability on the LCD screen, and sends that information to the LCD to be displayed to the user.

A. Data Parsing

Once initial pins are set and the serial connection between ESP32 and Pi is established, the Pi Python program sets the output pin to LOW, signaling to the ESP32 (which is in its own idling while loop) that it is ready to start receiving the block bitstream stored within its 2D array. Once the ESP32 receives this information and starts writing to the serial connection (along with a 'start' and 'stop' code to ensure only block information within it is processed), the Python program reads each successive byte of information, finds its corresponding code segment in the code mapping dictionary provided to it, and appends that code segment to a string (for example, if the program reads the byte '00010001', it looks to the dictionary, finds that '00010001' maps to 'int ', and appends 'int ' to a string).

This string is appended to for each block/byte processed, with a newline character added after every 4 blocks so that the final C program's compile errors point to the specific line in the user's code where it broke (the grid is 4x4, so we know that each line can only be at max 4 blocks long, and if this measure were not implemented, every line of user-created code would show up as one line to the C compiler, leading to user confusion if a line-specific error occurs). When 16 blocks' worth of information (whether actual byte values or NULL addresses from empty grid spaces) is read by the program, the output pin is set to LOW to indicate to the ESP32 to cease block address transmission over the serial connection.

B. File Compilation/Generation

A. From here, the completed string of code (corresponding to the written meaning of the user's code blocks) is inserted into another string in the format of a conventional C-file (with library imports, variable declarations for x, y, and z where all three are set as ints with value 0, and print statements that print the final values of x, y, and z to an 'output.txt' file). This newly created file on the Raspberry Pi, 'generated_code.c', is then run in the Python program using the subprocess library, with all output types (return code, compile and runtime errors, and program output) captured by the program.

Once the C file is run, if a non-zero return code is read (indicating some sort of error occurred), the error message is extracted, parsed for the line number(s) where the error occurred while ensuring that there are no duplicate line numbers, and adjusts it so that the lines where the error occurred match what the user would see on the grid (i.e. if the error occurred on line 12 of the actual C program, since the user does not see the headers or import statements, displaying 'Compile Error! Lines 12' would not match with what the user sees on the board, so it must be adjusted). The line numbers are then appended to a compile error message which is sent to the LCD; if another type of compile error occurs where the line number of the error is uncertain, a generic compile error is written to the LCD instead. From here, if the return code for 'generated_c_file.c/a.out' returns a non-zero error code, indicating a runtime error, a "Runtime Error" message is written to the LCD instead.

C. Output Code Display

Finally, the Raspberry Pi Zero 2 W outputs the results from compiling and running the program to the connected LCD 1602. The I2C connection model is used between the Raspberry Pi Zero 2 and the LCD to send data from the Pi to the LCD to be displayed. In order to use this I2C interface, an additional I2C board was attached to the back of the LCD display that reduces the number of pin connections required from 16 to 4 under the I2C communication protocol. On the Raspberry Pi side, the rpi_lcd python library is used to interface with the display from a software perspective. As previously discussed, based on the results of compiling and running the program, one of three

outputs is printed to the LCD. Figure 9 shows what the LCD display will generate based on the blocks used.

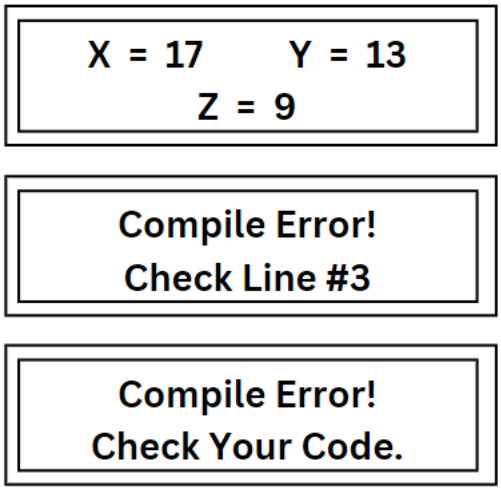

Fig 9: Possible LCD Display Outputs

To clear the display for the next set of results from the compilation and running process during the next cycle, the LCD must be cleared immediately before printing the results for the current cycle. This prevents the LCD from being displayed as empty while results are being collected while also preventing data from two separate cycles from being displayed simultaneously.

VI. CONCLUSION

The project aims to develop a cohesive programming learning tool for educators to teach younger students the fundamentals of computer programming. It involves hardware components like the ESP32, Raspberry Pi Zero W, LCD screen, and LED indicators, each playing specific roles in the data flow. Testing will be conducted in stages, focusing first on hardware functionalities, then on software development and integration. Challenges include hardware connectivity, power management, and understanding current limits for GPIO ports. The system's design enables real-time feedback for users, fostering transparency between hardware and users. The tool features various programming blocks representing key coding concepts, aiming to teach C language fundamentals. Its compact size supports an 4x4 block grid for standard coding. Compared to existing tools like Scratch, this project offers a physical interaction with code, potentially enhancing problem-solving and programming skills in early education settings, including high schools, summer camps, and middle schools, to prepare younger generations for the software development workforce.

BIOGRAPHY

John Gierlach is a 23-year-old graduating with a Computer Engineering degree with a focus in VLSI Design. After getting his bachelor's degree he will plan to get his master's degree in Computer Engineering at the University of Central Florida while working at AMD as a Silicon Design Engineer.

Robert Buch is a 22 year old graduating with a degree in Computer Engineering. He will be working with Lockheed Martin as a software engineer once he graduates, and pursuing a master's degree in computer engineering at the University of Central Florida.

Tyler Goldsmith is a 22 year old graduating with a degree in Computer Engineering with a focus on VLSI design. He will be working at AMD once he graduates, and pursuing a master's degree in Computer Engineering at the University of Central Florida.

Nandhu Jani is a 21 year old graduating with a degree in Computer Engineering. He is planning to work at ODP Corp once he graduates, with an interest in pursuing a master's degree in Computer Engineering in the near future.

REFERENCES

[1] *ESP32WROOM32D & ESP32WROOM32U Datasheet*. Espressif. (n.d.). https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf

[2] *Raspberry Pi*, Raspberry Pi 5, https://www.raspberrypi.com/

[3] SunFounder. (n.d.-a). *I2C LCD1602*. SunFounder. (n.d.). I2C LCD1602. https://docs.sunfounder.com/projects/raphael-kit/en/latest/components/component_i2c_lcd.html