DigiBlox: “Lego” blocks that render themselves in a virtual environment

A Major Qualifying Project submitted to the Faculty of
Worcester Polytechnic Institute

In partial fulfillment of the requirements for the
Degree of Bachelor of Science

by

Anas Maghfur

14th January 2011

Advisor:

Prof. John A. Orr, Worcester Polytechnic Institute

Abstract

The purpose of this Major Qualifying Project is to demonstrate a proof-of-concept for the DigiBlox. The DigiBlox are similar to “Lego” construction blocks and have the ability to determine their assembly configuration and then transmit that information to a PC where it would be used to render a 3D model of the assembled blocks in a virtual environment. These blocks are cubed-shaped, come in different colors, and have the ability to connect with one another at all six sides.

Embedded in each block is a microcontroller that will allow it to communicate with neighboring blocks. When the blocks are assembled to form a structure, a network of connections is established allowing each block to communicate with blocks beyond its neighbors.

An algorithm was developed that will utilize this network of connections to calculate the assembly configuration of the blocks. Using this network of connections, the algorithm would determine the position and color of each block relative to a unique block in the structure and send that information to the PC to render 3D models of those blocks in the appropriate position in virtual space.

Structures were built using the DigiBlox to test if the calculated assembly configuration sent to the PC was accurate. The tests showed that there was a bug in the code embedded in the microcontrollers of the blocks. Though the proof-of-concept was not attainable in the time frame of this major qualifying project, further work to eliminate this bug would determine whether this algorithm would work in a multi-processor system or not.

Table of Contents

1 Introduction 6

1.1 Goal 6

1.2 Motivation 6

2 Prior Art 7

2.1 GameBlocks 7

2.2 Tern 8

2.3 ActiveCube 9

3 Product Requirement 10

4 Design Approaches 15

4.1 Resistor Network 15

4.2 Vision Markers 18

4.3 Shift Register 20

4.4 Distributed Network Routing 28

5 MQP Specification 36

6 Design, Implementation and Testing 37

6.1 Proving the Algorithm 37

6.2 Block Diagram 39

6.3 Basic Blocks 41

6.3.1 Build Tree 42

6.3.2 Send/Receive Bits 47
6.3.3 Data Manipulation 49
6.3.4 Mechanical Construction 57
6.4 Brain Block 62
6.4.1 Generate Clock Signal 62
6.4.2 Send/Receive Bits 64
6.5 Virtual Environment 65
7 Project Results 66
8 Budget Results 68
9 Conclusion 69
10 Bibliography 70
11 Appendices 71
11.1 Source Code of Basic Block 71
11.2 Source Code of Brain Block 76
11.3 Source Code for Virtual Environment 77

Table of Figures

Figure 1: Children Using the GameBlocks 8
Figure 2: Different Types of Tern Blocks 8
Figure 3: A Software Code “written” in Tern 9
Figure 4: Children Playing with the ActiveCube 10

Figure 5: Different Views of Test Structure 1 14

Figure 6: Different Views of Test Structure 2 14

Figure 7: Different Views of Test Structure 3 15

Figure 8: System Diagram of the Resistor Network Approach 16

Figure 9: Stacking a Block 17

Figure 10: AR ToolKit used for Augmented Reality 19

Figure 11: Complete System Diagram of the DigiBlox Playset 22

Figure 12: System Diagram of a Single Block 23

Figure 13: A Stack of Blocks on the Workbench 23

Figure 14: Concept Image of the Workbench 26

Figure 15: System Diagram for the Workbench 27

Figure 16: Connected Blocks Represented as a Graph 29

Figure 17: Constructed Tree after Running BFS 30

Figure 18: Information Flow from Block to Brain Block 31

Figure 19: BFS Procedure in Psuedocode 33

Figure 20: BFS in Action 34

Figure 21: Top-Level Block Diagram 40

Figure 22: Schematic of the Basic Block 41

Figure 23: MSP430 and Development Board 42

Figure 24: Code Snippet of Main 43

Figure 25: Flowchart of Main 45

Figure 26: Testing the Build Tree 46

Figure 27: Cope Snippet of Interrupt Service Routine 48

Figure 28: Testing the Send/Receive Bits 49

Figure 29: Example Configuration 50
Figure 30: Code Snippet for Read Data 52
Figure 31: Example of Data Manipulation 55
Figure 32: Snake-esque Configuration 56
Figure 33: A Third of a Block 58
Figure 34: Assembled to Make a Cube 59
Figure 35: Front, Side and Top View of a Block 60
Figure 36: Photograph of a Basic Block 61
Figure 37: Schematic of Brain Block 62
Figure 38: Code Snippet for Brain Block 63
Figure 39: Code Snippet for Brain Block with Highlights 64
Figure 40: Early Version of Virtual Environment 65
Figure 41: Script Used to Generate Blocks in Virtual Environment 66
Figure 42: Simple Test Configuration 67

Table of Figures

Table 1: Requirement and Specifications 11
Table 2: Expected Test Results 47
Table 3: Bit Sequence Definitions 51
Table 4: Parts List for Mechanical Design 61
Table 5: Datasheet Snippet of MSP4302013 63
Table 6: Complete Parts List 68

1 Introduction

1.1 Goal

The goal of this MQP is to design and realize the DigiBlox – a playset of "Lego" blocks containing electronics that transmits the blocks' assembly configuration to a piece of software which will use it to render a representation of the assembled blocks in a virtual environment.

1.2 Motivation

The idea of the DigiBlox was engendered by the desire to promote a creative learning environment for children. With the DigiBlox, children will be able to build structures like they would with normal *Legos* and witness their creations come into existence in a virtual environment. They would then be able to perform experiments on their creations in the virtual environment that would otherwise be impractical should it be done in real life. For example, a group of children may build a bridge using the DigiBlox and may want to see if the bridge could withstand a dynamite explosion under it. Dynamites would be too dangerous for children to play with but in the virtual environment, where that bridge they just built now exists, they will be able to put virtual dynamites under it and run a simulation. They could also go one step further by assigning the bridge properties like 'all red blocks are made out of steel and all green blocks are made out of marsh mellows' and see what happens if the simulation is ran again. Surely they could have easily built the bridge directly in the virtual environment with a mouse and keyboard; however, having children directly manipulate objects instead of using a mouse and keyboard makes an interface easy to learn, to use, and to retain over time.[1] Furthermore, the "play" setting where children interact with other children will be lost with a mouse and keyboard, and instead will be replaced by a "hot seat" scenario (i.e. children taking turns using the mouse and keyboard to build the bridge). Having

[1] (Shneiderman, 1998)

a collaborative play setting is conducive to learning and creativity among children.[2] What the DigiBlox does is retain the "play" environment while taking advantage of the flexibility and possibilities of software.

2 Prior Art

The concept of the DigiBlox is relatively new thereby limiting the available prior art. No commercially available product resembles the DigiBlox but there are a few in research laboratories, albeit in prototype forms. The ones selected for further analysis were chosen for their disparate implementation method. They are the following: the GameBlocks[3], the Tern[4] and the ActiveCube[5].

2.1 GameBlocks

GameBlocks – Internet of Things Engineering Group – CSIR Mereka Institute

The GameBlocks is a "digital manipulative system for coding simple program sequences to control a toy robot". Below is a picture of children using the GameBlocks. They are placing blocks with magnets embedded in them on a tray of switches. Each block type has magnets positioned at different locations depending on the block type; this is so that specific blocks will close specific switches on the tray. By knowing what switches are closed, the robot will know what program sequence to execute.

[2] (Boden, 2004)
[3] (Smith, 2007)
[4] (Horn, 2007)
[5] (Kitamura, 2001)

Figure 1: Children Using the GameBlocks

This implementation method means that each block can be very cheap to manufacture since it only consists of magnets. But the problem with it when applied to the DigiBlox is that it only allows for 2D structures.

2.2 Tern

Tern – Human Computer Interaction Lab – Tufts University

The Tern is a tangible programming language where programmers write software code by arranging blocks on a flat surface. The only thing special about the blocks and surface is that the blocks have a marker imprinted on it and the surface has a camera mounted above it. Below shows an image of the types of blocks and another image showing the blocks connected together to form a software code.

Figure 2: Different Types of Tern Blocks

Figure 3: A Software Code "written" in Tern

When the blocks are placed together on the surface, the computer will capture images of the blocks and use image processing techniques to convert the captured images into software code.

Because each block is basically a piece of wood with an imprinted marker, this implementation method is extremely cheap. Furthermore, the lack of moving parts and electrical components implies that it is not prone to hardware failure. However, like the GameBlocks, this implementation would only work for 2D structures unless multiple cameras are used.

2.3 ActiveCube

ActiveCube – Human Interface Engineering Lab – Osaka University

ActiveCube is probably the system closest to that of the DigiBlox. It is described as "a novel device that allows a user to construct and interact with a 3D environment by using cubes." A computer recognizes the 3D structure of connected cubes in real time by utilizing the communication network among cubes as suggested in the image below.

Figure 4: Children Playing with the ActiveCube

This communication network is made possible by the Local Operating Network (LON) microchips (designed by Echelon Corporation and manufactured by Toshiba) embedded in each block. The microchips are known to be reliable and robust in networking applications thereby allowing for users to build large structures with many different types of blocks. However, all that comes at a high price. Quotes from Space Coast IC and Sierra IC put the price of each LON microchip at above US$4 per unit even in large quantities (over 5,000). Children playing with Lego blocks often build structures consisting of as many as 100 blocks. If each block contains the LON microchips as well as additional auxiliary components, not to mention the material to encase them, a normal ActiveCube structure would be worth tens or hundreds of dollars. The ActiveCube is an effective but expensive system.

3 Product Requirement

Seeing that there are no commercial products that do something similar to what the DigiBlox playset should do, a heuristic approach was used to determine the product requirement – a list of requirements for a final product releasable in the market. Below lists the requirements and explains the reason for each one of them.

Table 1: Requirement and Specifications

	Product Requirement	Reason
Robust	The playset should be scalable. In other words, it should allow for an infinite number of types and amount of blocks. However, the initial playset should consist of 15 blocks of three different types: 5 red, 5 green and 5 blue.	15 blocks of 3 different types seem appropriate for a starter kit. Since the playset should be scalable, users will be able to purchase more blocks when they are made available.
	The playset should at least allow users to build the test structures in Figure 5, Figure 6 and Figure 7.	These test structures covers even the most eccentric assembly configurations, particular ones with gaps and holes. If the DigiBlox playset can be used to build these test structures, then it could be used to build a wide variety of structures.
Responsive	The time it takes for the software to update the blocks in the virtual environment after a change has been made in the real world should be no longer that 2 second.	2 second seems to be the approximate amount of time it take for someone to switch attention from making a change to a structure and viewing the virtual environment.

Stable and Safe	There should be no current greater than 0.06A (AC) or 0.5A (DC).	Literature values for lethal current levels are 0.06A (AC) or 0.5A (DC).[6]
	A stack of at least 15 blocks should be able to stay in place.	Experience and observations suggests that 15 blocks of Duplo (the preschool line of Lego) can be stacked in place without it toppling over.
Easy to Install	The hardware should communicate with the software through USB.	USB is the standard for many peripheral devices. Most, if not all, modern personal computers have USB ports.
	The hardware should be powered by the USB port of a PC and/or off-the-shelf batteries (AA and AAA only).	The playset will need to plug into the PC via USB. It would be very convenient if the USB is the only cable that users have to plug in. Also, AA and AAA batteries are the batteries that are often expected in consumer electronics. Hence, these battery types are acceptable for the playset.
Easy to Use	The physical dimension of a block should be between 1x1x1cm and 4x4x4cm.	The dimensions of Lego blocks, which are already known for its ease of use for children, typically fall in this range.[7]
	The user should be able to pan around the structure in the virtual environment using the mouse or keyboard.	Camera control is a basic interface feature of virtual environments.[8]

[6] (Hsu, 2000)
[7] (Dimensions of a Standard Lego Brick)
[8] (Noble, 2009)

Affordable	The cost of components for each block when manufactured in large quantities (more than 1k units) should be less than US$1.	There can be structures that are made out of as many as 100 blocks. The playset would be considered a very expensive toy for children if each block cost more than US$1.
	The cost to manufacture the starting hardware, if any, in large quantities (more than 1k units) should be less than US$20.	20-100 US dollars seems to be an appropriate initial investment for a child's playset which is why 20 US dollars was selected as the target cost to manufacture the starting hardware.
	The entire hardware should not consume more than 10 watts.	The hardware itself should be affordable to run for a toy; therefore it should not consume more than 10 kW/h.[9]

[9] (Bluejay)

Important Note: Although they make look like standard “Lego” blocks, the structures in Figure 5, Figure 6 and Figure 7 are made of only 1x1x1 blocks with the ability to connect at all six sides.

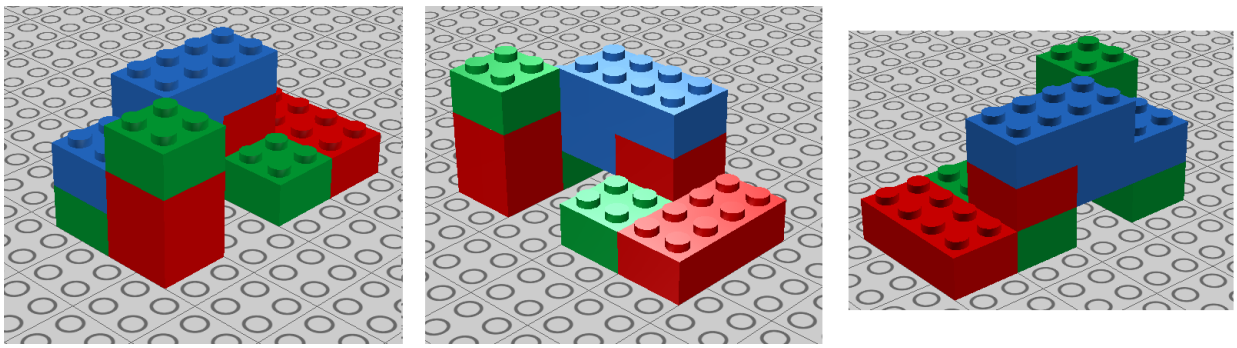


Figure 5: Different Views of Test Structure 1

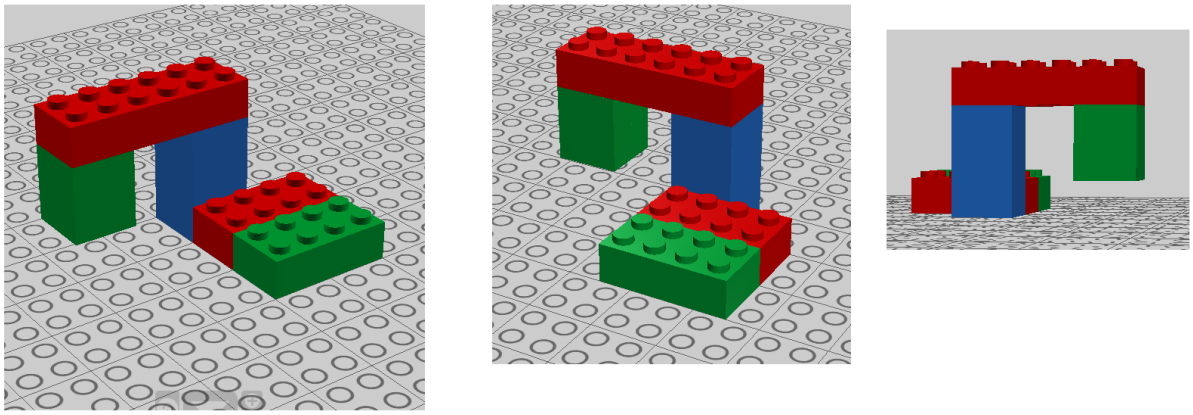


Figure 6: Different Views of Test Structure 2

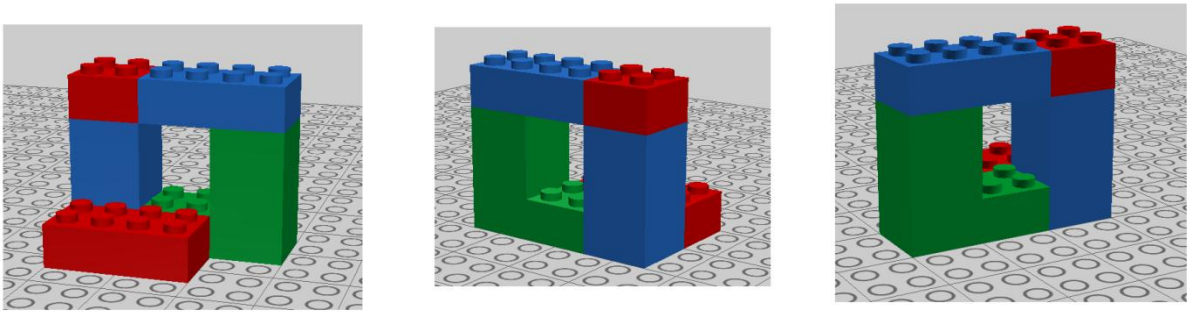


Figure 7: Different Views of Test Structure 3

4 Design Approaches

After much brainstorming and researching on prior art, design approaches from a broad spectrum came into mind. They are the following: resistor network, vision markers, shift register, distributed network routing. Each one of them was analyzed against the product requirements covered in section 3 for strengths and weaknesses until finally one was selected.

4.1 Resistor Network

This approach uses a resistor network to determine the block locations. The blocks are placed on a specially designed surface, the workbench, as shown below.

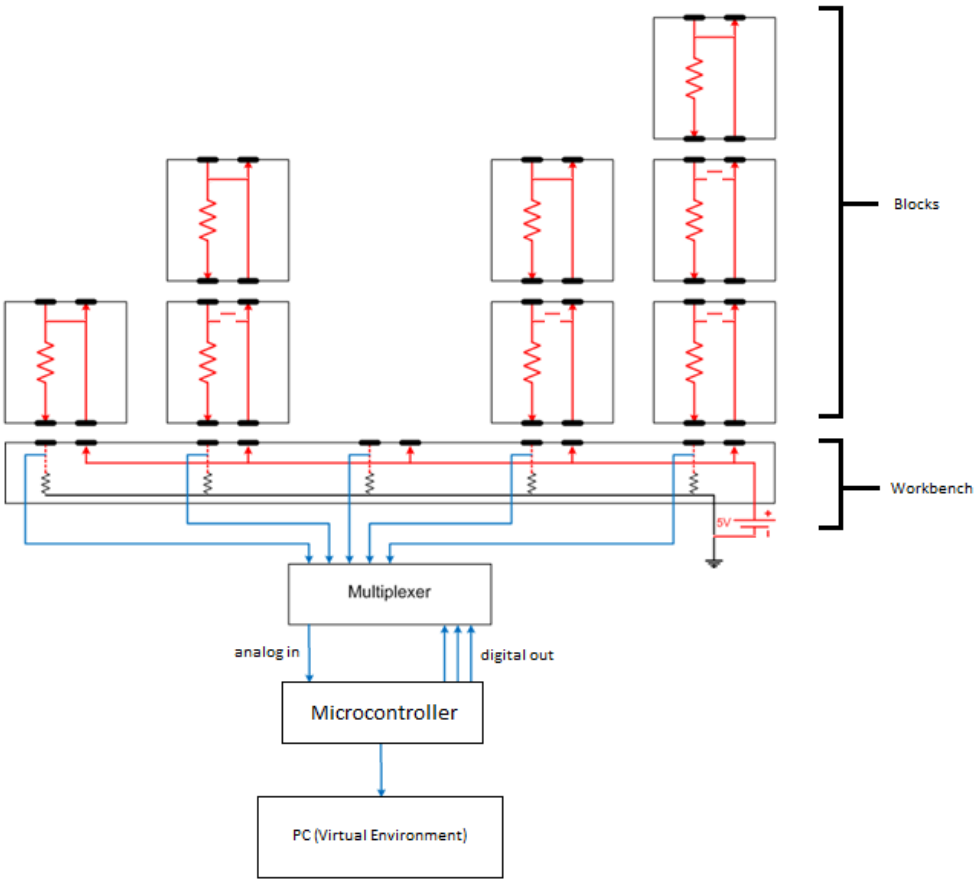


Figure 8: System Diagram of the Resistor Network Approach

The blocks consist of a resistor and a switch. The switch in a block is always closed when there is nothing connected to the top of it. However, when a block is connected to the top of another block, the switch in the latter will open. This is illustrated below. As you can see, when the top block is stacked on top, the switch in the middle block opens.

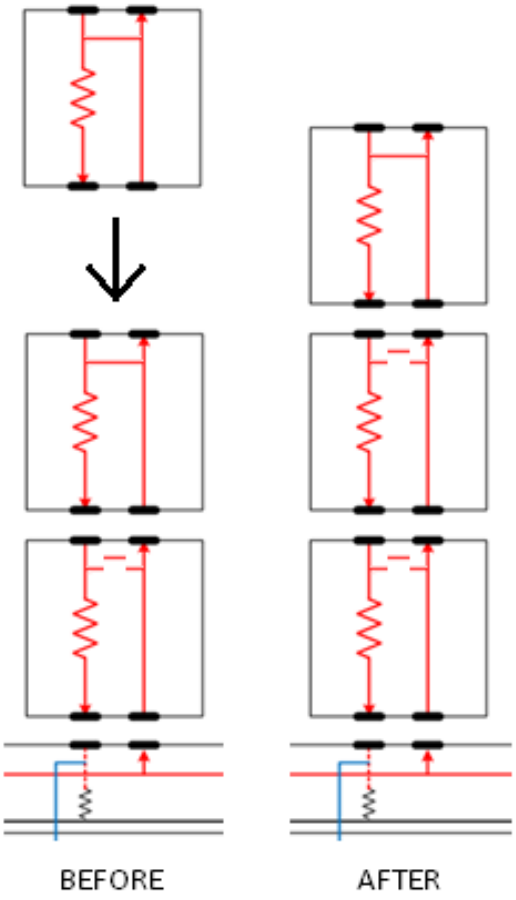


Figure 9: Stacking a Block

Notice how each stack of blocks is a series of resistors. When a stack of blocks is placed on the workbench, the series of resistor will be a complete circuit with a V_{dd} = 5V voltage source. Furthermore, the voltage across the last resistor (the resistor embedded in the workbench) will have a voltage given by this equation:

$$\text{voltage across last resistor} = \frac{V_{dd}}{1 + \text{number of blocks in stack}}$$

The microcontroller in conjunction with the multiplexor embedded in the workbench will read the voltage across the last resistor of each stack, use that information to calculate the number of blocks in

those stacks and send the result to the PC where it would be used to render the structure in the virtual environment.

The problem with this design is that there can only be one type of block. Having different blocks with different resistors stacked together will allow the system to determine what blocks may be present in the stack, but it will not allow the system to figure out the order in which the blocks are stacked. A way to remedy this is to make it so that the user must place the blocks one at a time so that the software can use the changes in the voltage to figure out what block with what resistance was just placed down. However, this means that when there are multiple users playing with the DigiBlox, it is not possible for them to work on separate parts of a construction and then bring back their work to the workbench. This is not an acceptable constraint for the user, especially for users that are children.

The main problem with this design is that there is a limit to how many blocks can be stacked before the voltage across the last resistor becomes so small that it becomes difficult for the software to accurately determine the number of blocks in the stack. Hence, this design was scrapped entirely.

4.2 Vision Markers

This approach is similar to the Tern in that it utilizes a piece of technology intended for augmented reality applications or other image processing systems. The vision markers are a part of the ARToolKit that is freely available from the Human Interface Technology research group at the University of Washington.[10]

The ARToolKit uses computer vision algorithms to calculate the real camera position and orientation relative to physical markers in real time. In other words, markers can be placed on physical objects and a computer with a camera will be able to determine the location of the marker as long as the camera is in view of it. In the image below, for example, the woman is wearing a head-mounted display with a built

[10] (AR ToolKit)

in camera and interface to a computer. She is holding a marker. When she looks at the marker, the computer feeds a computer-generated cartoon character into her display so that it appears as if the cartoon is directly on top of the marker no matter what orientation or distance the marker is held at. This is because the AR ToolKit tells the computer where the marker is located so that the computer knows where to generate the cartoon.

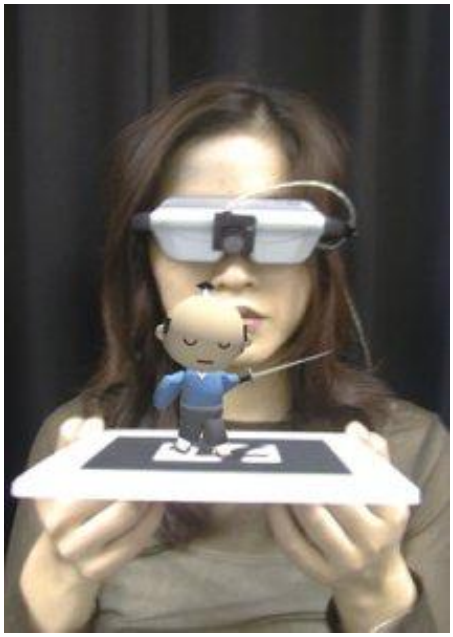


Figure 10: AR ToolKit used for Augmented Reality

This concept could be applied to the DigiBlox as well because the software allows for multiple markers at a given time. Multiple markers allow for each block type to have its own marker. However, this entails that there must be cameras pointing at all sides (top, bottom, left, right, front and back) of the structure as it is built and the positions of the cameras must be carefully placed with respect to one another. Also, the camera must be in view of the structure at every step of the building process; if the cameras fail to capture a step in the building process where a block gets enclosed by other blocks, the enclosed block will not appear in the final rendition in the virtual environment. The errors that this design can result in

render it undesirable. Besides, having a multiple camera setup is not exactly an ideal play area for children, especially when there are many children working together to build a structure.

4.3 Shift Register

The shift register approach is like a hybrid of the resistor network approach and the distributed network routing approach. Like the resistor network approach, the shift register approach utilizes a workbench that consists of a matrix of connectors allowing it to connect to the bottom of stacks of blocks placed on top of it. Each stack of blocks acts like a shift register but instead of shifting a data bit down the stack to the workbench, it is shifting information regarding the type of block.

This means that the workbench will know the order at which the blocks are stacked in for each stack. And since it also knows where on the workbench the stacks exist, it can give that information to the PC where it will be used to render the structure in a virtual environment. So, in a way, it is like the distributed network approach in that each block is sending information to the computer down a network of blocks, except that there are multiple networks and each network follows a linear path.

This design approach, unlike the resistor network approach, allows for different kinds of blocks and allows users to build large stacks of blocks. Additionally, the design approach is not as cumbersome as the multiple camera setup of the vision marker approach. The best aspect of it, however, is that the microprocessor embedded in each block that does the shifting of information only needs a fixed amount of memory because as new information comes in, the old ones go out. This memory constraint was the problem with distributed network routing approach but the shift register approach manages to overcome this problem.

By combining the resistor network approach with the distributed network approach, the result is a design that seems to be the most promising of all the approaches considered. As a result, it is the design

approach that will be used to realize the DigiBlox. The proceeding chapter will describe this approach in detail.

The DigiBlox playset consists of three main modules: the blocks, the workbench, and the virtual environment. The workbench contains a microcontroller, a power supply and a connection to the PC (where the virtual environment is generated) while the blocks only have a microcontroller embedded in them. When the blocks are assembled together on the workbench, a network of connections will be established between the blocks and the workbench allowing the workbench to power up the blocks so that they would transmit their information to the workbench via this network of connections. The workbench will then compile all the information it received and transmit them to the PC where a representation of the assembled blocks will be built in the virtual environment.

Below shows the system diagram of the DigiBlox playset illustrating how the blocks, the workbench and the virtual environment all integrate together. Detailed description of how each module works is explained in the next section.

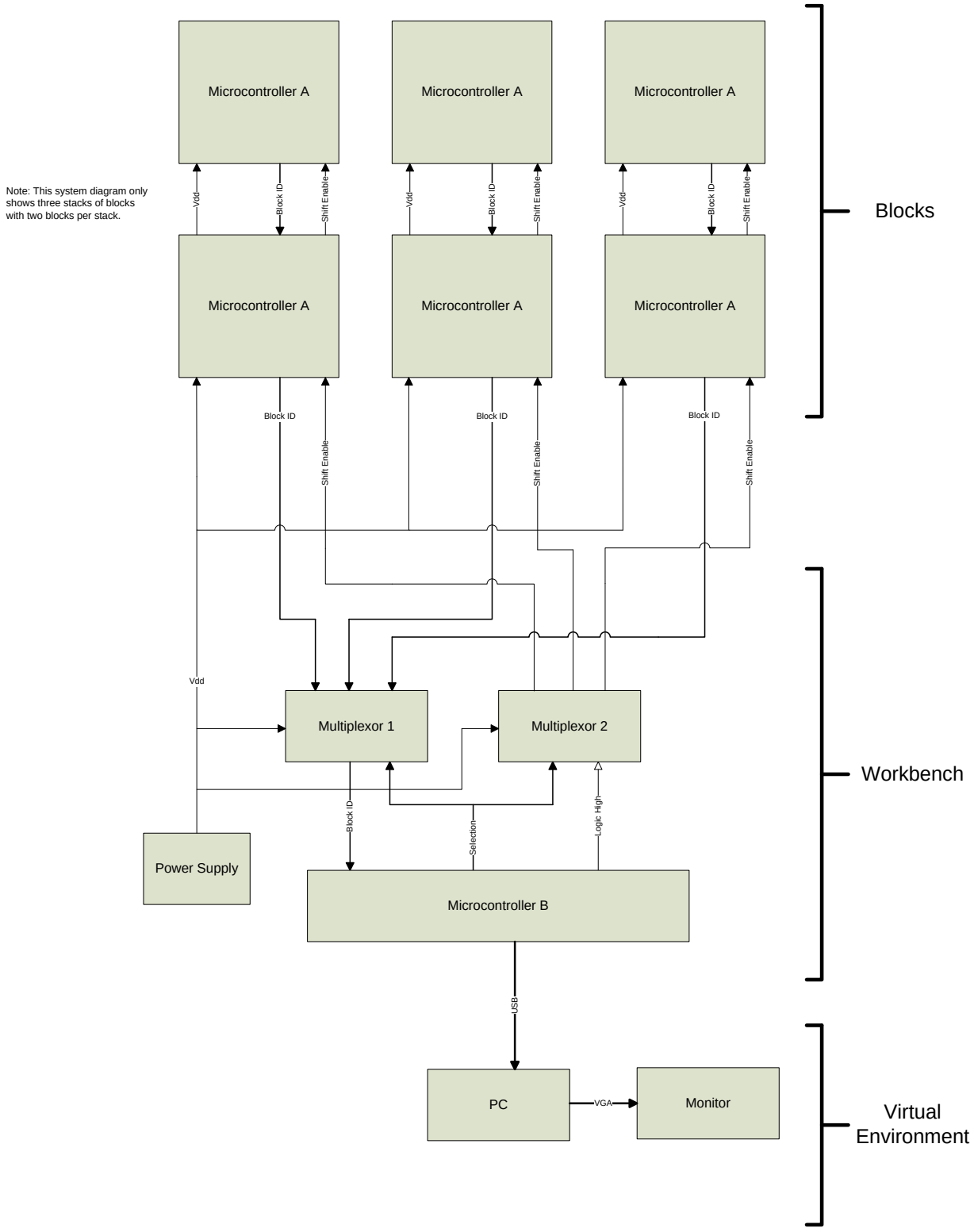


Figure 11: Complete System Diagram of the DigiBlox Playset

The blocks' main component is the microcontroller. Below is the system diagram of a single block.

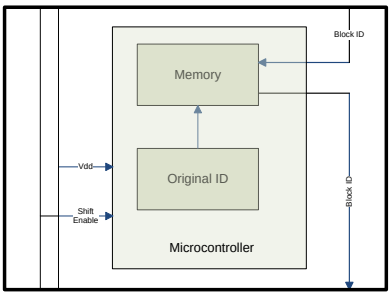


Figure 12: System Diagram of a Single Block

The blocks come in different types with an ID associated for each type; for example, red colored blocks will have "red" as their ID and blue colored blocks will have "blue" as their ID. When the blocks are stacked on top of one another on the workbench, their V_{dd}, *shift enable* and *block ID* lines will all get connected as shown below. This means that when the workbench provide the bottom block power via the V_{dd} line, all the other blocks will also get power.

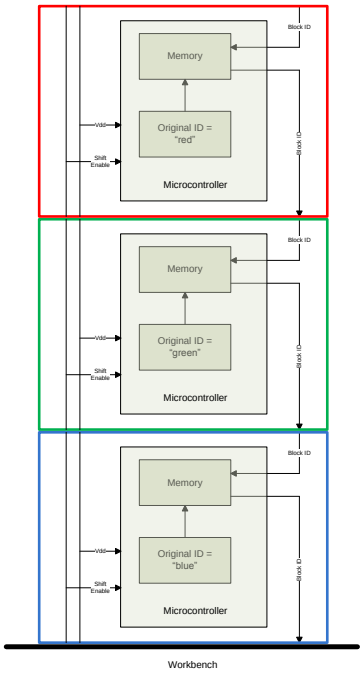
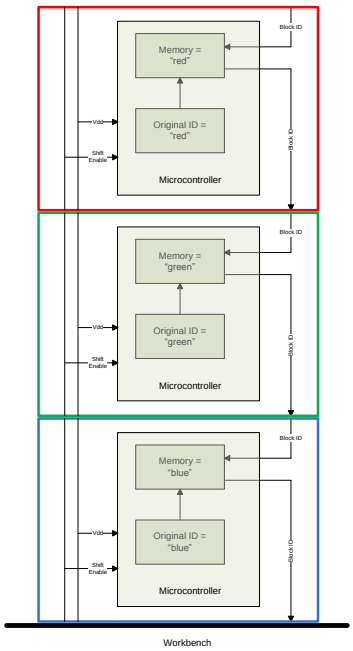


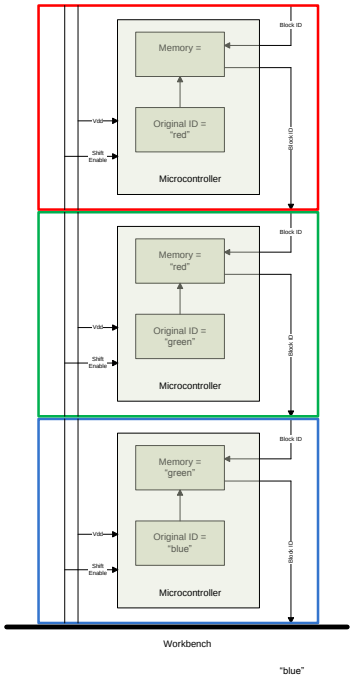
Figure 13: A Stack of Blocks on the Workbench

Furthermore, when the workbench set the *shift enable* line to active high, it will prompt the blocks to start shifting their IDs through the *block ID* line. For example, in the configuration above, where a red, green and blue block are stacked on top of each other in that order on the workbench, when the workbench sets the *shift enable* to high, the following will occur in sequence:

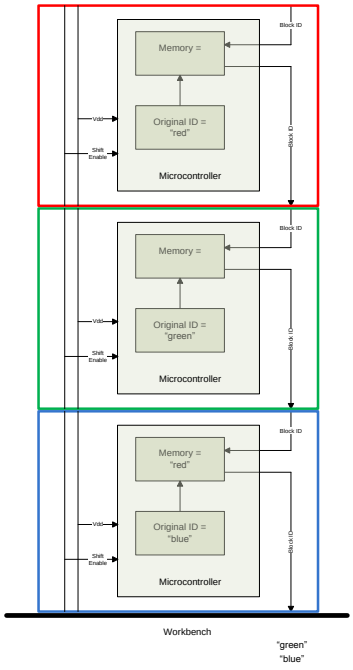
1. All blocks will make a copy of their IDs in their temporary memory to prepare for the proceeding shifting process as shown on the right.



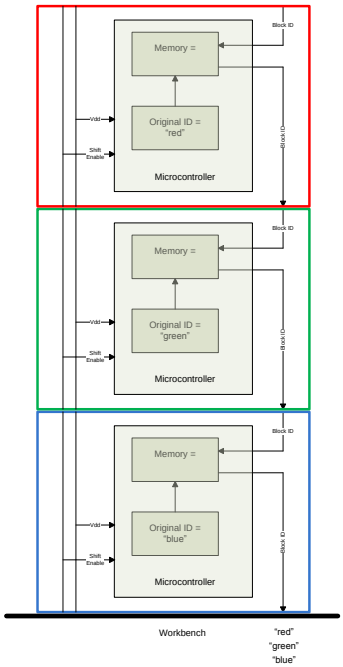
2. The shifting process now begins: The blue block will send “blue” to the workbench, the green block will send “green” to blue block and the red block will send “red” to the green block as shown on the right.



3. The blue will send “green” to the workbench and the green block will send “red” to the blue block as shown on the right.



4. The blue block will send “red” to the workbench as shown in the right.



Notice that the workbench receives the ID of each block in order of the way it was stacked. This will be used later on by the software to determine what block goes where when rendering the blocks in a virtual environment.

The workbench is a flat surface consisting of a matrix of cells outlining where blocks could be placed. It will look like the image below.

Figure 14: Concept Image of the Workbench

Each cell has two output pins and one input pin. The output pins are the V_{dd} and the *shift enable*, and the input pin is the *block ID*. The workbench also has a microcontroller and multiplexors connected in the manner shown below.

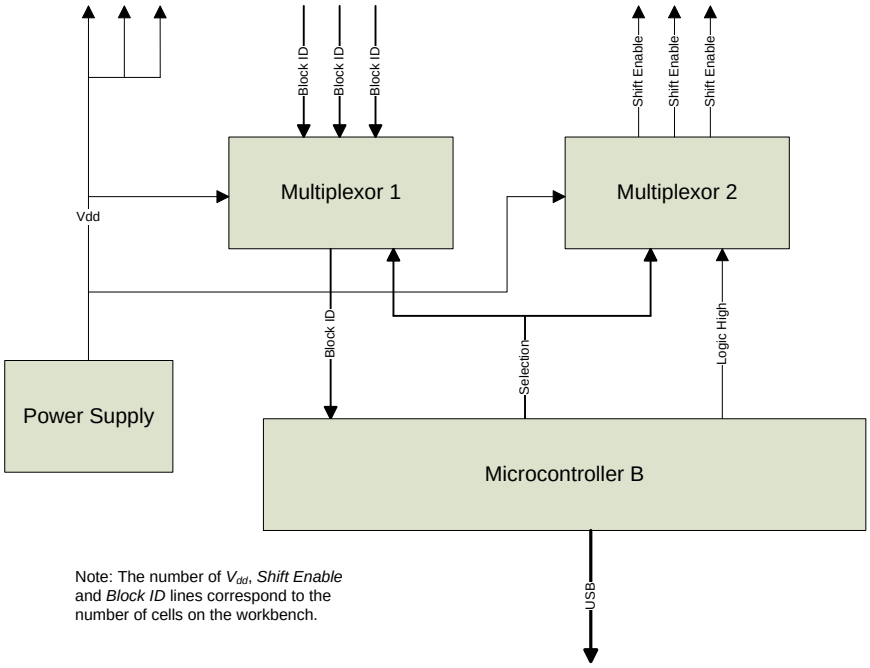


Figure 15: System Diagram for the Workbench

The idea is to make the microcontroller, with the help of the multiplexors, retrieve the IDs of blocks stacked on a cell for each cell. It does this by setting the *shift enable* for the cell to high so that the stacked blocks on that cell can begin shifting their IDs to the workbench. Once that is done, it will move on to the next cell and so on and so forth until eventually it will know the stacking configuration of each cell. It will aggregate all this information and transmit it to the PC where the information will be used to render a virtual representation of the blocks in a virtual environment.

This approach begs the question, “How can you build a bridge?” A bridge is a structure where there is nothing under certain blocks. Since this approach is based on the idea of blocks being stacked, how do you insert a gap between blocks in a stack? The solution to this is to have an “invisible” block. The

invisible block is a transparent block in the real world but is an unoccupied space in the virtual world. The problem with this solution is that it really does not allow children to play with their physical creations. For example, children would not be able to have toy cars passing under the bridge they built because there are those “invisible” blocks in the way.

4.4 Distributed Network Routing

This approach treats the blocks as a distributed network routing system where the assembled blocks are a network of nodes, one of which is connected to the PC. Each block has a microprocessor that allows it to communicate with neighboring blocks. Since all the blocks are connected to one another, this network of connections allows each block to communicate with blocks beyond its immediate vicinity. An algorithm would then be implemented that will have all the microprocessors of each block work together to map its assembly configuration and then send that information to the PC. The PC will then be able to render the assembled blocks as a 3D model in a virtual environment.

To explain how the algorithm works, suppose the blocks are connected in the manner shown in Figure 16 with the gray circle representing the block that is connected to the Brain Block (assume the Brain Block is connected immediately below the gray circle). The letter within each block indicates blocks of different types where the red, green and blue colored blocks corresponds to A, B and C.

Important Note: Figure 16 does not show physical connections; it is merely a representation.

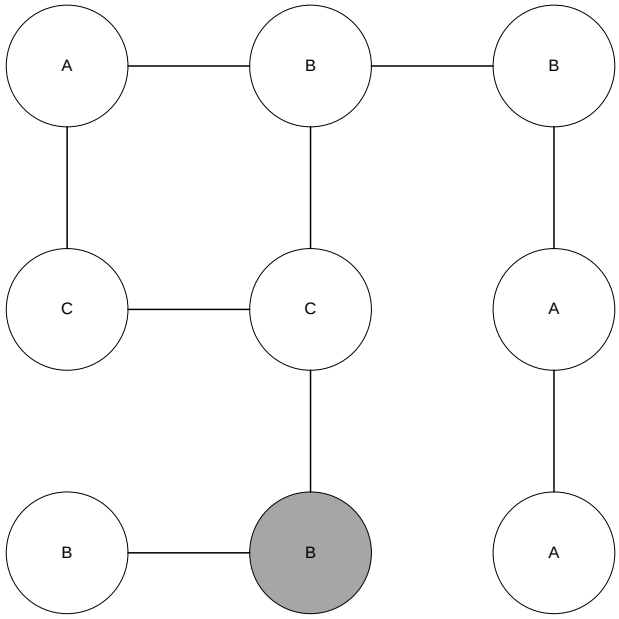


Figure 16: Connected Blocks Represented as a Graph

The DigiBlox uses a Breadth-First Search (BFS) algorithm to build a tree with vertices reachable from the root vertex. In non-graph theory terms, the algorithm establishes a flow of information between blocks so that every block has a unique path to send information to the block represented by the gray circle (the block immediately connected to the Brain Block which is not shown in the figures).

Running BFS on the construction in Figure 16 will build the tree shown in Figure 17 where the gray vertex is the root of the tree (or the block that is immediately connected to the Brain Block which again is not shown in figures) and hereby will be called the root block.

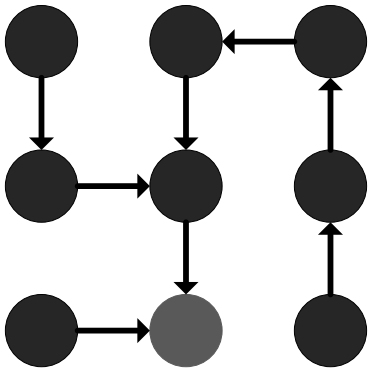


Figure 17: Constructed Tree after Running BFS

After the tree has been established, each block has a unique path to send information to the root block. In Figure 18, a specific block sends a packet down the tree. The packet initially contains just the type of the block, but as the packet traverses down the tree, it accumulates information regarding the direction of its traversal. It will then reach the root block, the root of the tree, where it will be sent to the PC via the Brain Block. By using the information contained in the packet, the PC can retrace the route the packet took and the type of block the packet came from. In Figure 18, for example, the PC receives the packet containing **C** ► ▼ ▼. Reversing the directions and order of the arrows within the packet will give **C** ▲ ▲ ◄. This means that a block of type **C** will be found up ▲, up ▲and left ◄ from the root block. Of course, this means that all the blocks must have the same physical orientation meaning that if each block were to have faces labeled 1 through 6, faces of the same label must point in the same direction. A forcing function must be implemented in the connecting mechanism to physically restrict uses from assembling the blocks in an incorrect orientation.

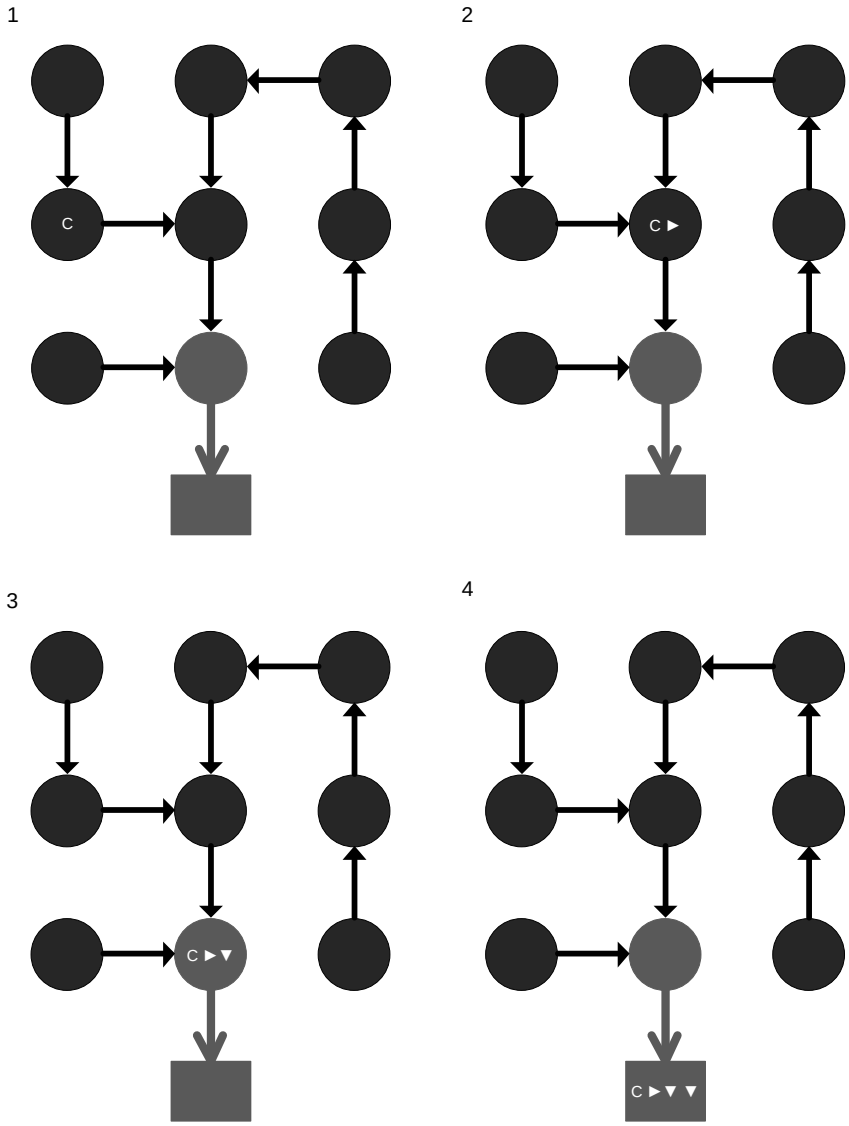


Figure 18: Information Flow from Block to Brain Block

If each block sends packets like this to the PC, then the PC would know what blocks go where and thus would be able to recreate the assembled blocks in a virtual environment. This is how the DigiBlox works!

Of course the DigiBlox is entirely dependent on the BFS algorithm's ability to construct the tree in the first place.

The inputs of the BFS are the graph $G = (V, E)$ (where V and E are vertices and edges in G respectively) and the vertex s (where $s \in V[G]$). In the context of the DigiBlox, G is the graph of the assembled blocks where the vertices V are the blocks, the edges E are the connections and the vertex s is the root block or, more specifically, the selected root of the tree to be grown from the graph G.

BFS works by exploring adjacent vertices starting from the root vertex s. It will keep track of its progress by coloring the vertices: all vertices start out white but once it gets discovered, it will get colored gray or even black if all vertices adjacent to it have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree with u being the parent of v. In terms of the DigiBlox, this means that packets from v will flow from u. Since a vertex is discovered at most once, it has at most one parent. All this is visually demonstrated in Figure 20 and the pseudocode in Figure 19 clearly describes the BFS procedure.

In describing BFS in psuedocode form, let me preface with the following:

- The color of each vertex $u \in V$ is stored in the variable $color[u]$, and the parent of u is stored in the variable $\pi[u]$. If u has no parent (for example, if $u = s$ or u has not been discovered), then $\pi[u]$ = NIL. (However, in the actual implementation of BFS in the DigiBlox, $\pi[u]$ contains the direction of the parent of u relative to u.)

- The distance from the source s to vertex u computed by the algorithm is stored in $d[u]$. (However, $d[u]$ has no use to BFS in the DigiBlox; it is only used for the algorithm's proof of correctness covered in chapter 6.1.)

- BFS also uses a first-in, first-out queue Q to manage the set of gray vertices. ENQUEUE(Q,u) means to put u into the queue Q and DEQUEUE(Q,u) means to remove u from the queue Q.

```
BFS(G, s)
1  for each vertex u ∈ V[G] - {s}
2     do color[u] ← WHITE
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← GRAY
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← Ø
9  ENQUEUE(Q, s)
10 while Q ≠ Ø
11    do u ← DEQUEUE(Q)
12       for each v ∈ Adj[u]
13          do if color[v] = WHITE
14                then color[v] ← GRAY
15                     d[v] ← d[u] + 1
16                     π[v] ← u
17                     ENQUEUE(Q, v)
18       color[u] ← BLACK
```

Figure 19: BFS Procedure in Psuedocode

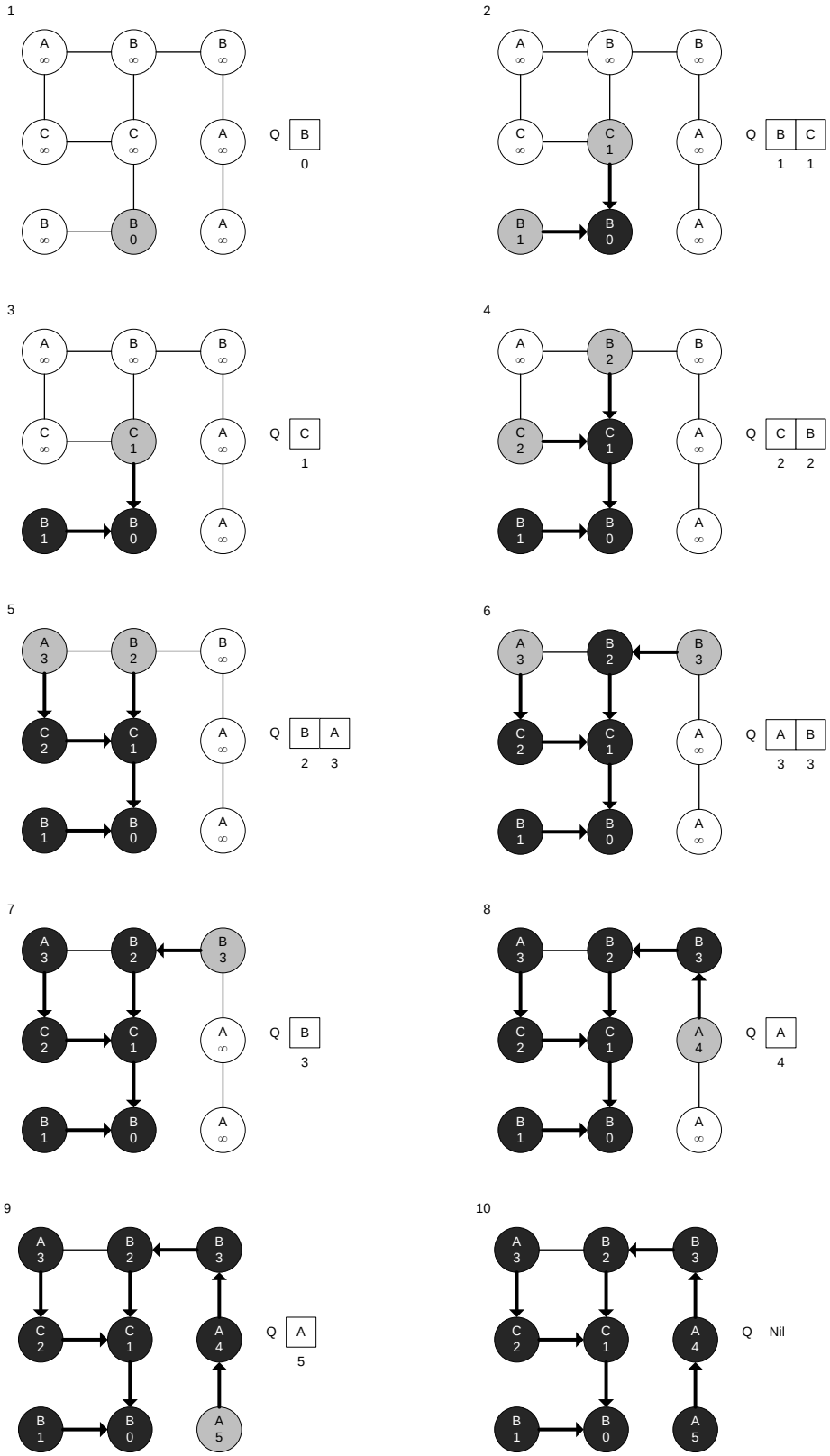


Figure 20: BFS in Action

This distributed network routing approach allow for an infinite number of types of blocks unlike the resistor network approach. It also avoids any extra cumbersome hardware like the cameras in the vision marker approach and the workbench in the resistor network approach. The only extra hardware is a single block that connects to the PC. The best part about this approach is that it is extremely robust in that it allows more freedom for users in creating structures. The six connectable sides and the fact that there are no workbenches or “invisible” blocks to consider gives rise to this freedom for the user.

One problem with the algorithm that was previously mentioned is that it requires the blocks to have the same orientation. However, the robustness of the algorithm makes this problem an acceptable trade-off.

Another problem with the algorithm is that the space complexity is not constant. See Figure 18 again and notice how the packet grows in size the more nodes it traverses. This indicates that the space complexity of the algorithm is proportional to the number of nodes (i.e. the number of blocks used in a structure). This means that this design will challenge low-memory microprocessors. However, this does not mean that a 15 block playset could not be built.

5 MQP Specification

The goal of this MQP is to demonstrate a proof-of-concept of the algorithm (covered in 4.4) in a multi-microprocessor system. The algorithm calculates the assembly configuration of interconnected microprocessors. This system of interconnected microprocessors will be developed in the context of the DigiBlox playset: a toy set consisting of “Lego” blocks that have the ability to transmit its assembly configuration to a piece of software which would then render the assembled blocks in a virtual environment. With that said, the following additional specifications should be met to fully demonstrate this proof-of-concept:

1. There should be three different types of blocks: red, green and blue.
2. Five blocks of each type should be available for use.
3. Each block should physically be a cube-shaped structure that can be grasped by one hand.
4. It should be quick and easy to assemble and dissemble the blocks given the constraint of the algorithm.
5. The test structures in Figure 5, Figure 6 and Figure 7 should be able to be built and rendered. (As explained in section 3, these test structures are complex enough to represent a wide-variety of structures.)
6. The virtual environment should be able to clearly showcase the assembled blocks.

6 Design, Implementation and Testing

The DigiBlox consists of the Basic Blocks (which will be typically referred to as just "blocks"), the Brain Block (which can only connect to one block) and the Virtual Environment. All these modules will be working together to execute the algorithm that determines the assembly configuration of the assembled blocks.

6.1 Proving the Algorithm

Before implementing the algorithm, a formal proof is needed to ensure that the algorithm works for all assembly configurations. The following is the mathematical proof of the algorithm verified by Prof. Mani Murali, Assistant Professor at Worcester Polytechnic Institute and an instructor of its algorithms course:

Assume, for the purpose of contradiction, that some vertex receives a d value not equal to its shortest path distance. Let v be the vertex with minimum $\delta(s, v)$ that receives such an incorrect d value; clearly $v \neq s$. By Lemma 1, $d[v] \geq \delta(s, v)$, and thus we have that $d[v] > \delta(s, v)$. Vertex v must be reachable from s, for if it is not, then $\delta(s, v) = \infty \geq d[v]$.[11]

Lemma 1

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $d[v]$ computed by BFS satisfies $d[v] \geq \delta(s, v)$.

Proof

We use induction on the number of ENQUEUE operations. Our inductive hypothesis is that $d[v] \geq \delta(s, v)$ for all $v \in V$.

[11] (Cormen, Leiserson, Rivest, & Stein, 2001)

The basis of the induction is the situation immediately after s is enqueued in line 9 of BFS (Figure 19). The inductive hypothesis holds here, because $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$.

For the inductive step, consider a white vertex v that is discovered during the search from a vertex u. The inductive hypothesis implies that $d[u] \geq \delta(s, u)$. From the assignment performed by line 15 and from Lemma 2, we obtain

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v). \end{aligned}$$

Vertex v is then enqueued, and it is never enqueued again because it is also grayed and the **then** clause of lines 14–17 is executed only for white vertices. Thus, the value of $d[v]$ never changes again, and the inductive hypothesis is maintained.

Lemma 2

Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Proof

If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v), and thus the inequality holds. If u is not reachable from s, then $\delta(s, u) = \infty$, and the inequality holds.

6.2 Block Diagram

The block diagram in Figure 21 shows a DigiBlox playset consisting of nine blocks attached together. The bottom center block is the block that connects to the Brain Block which is the only block that interfaces with the PC. The blocks communicate with each other through a bus driven by a clock signal generated by the brain block to maintain synchronization. Although the block diagram only shows each block having four buses, the actual blocks will have six buses altogether, one for each side.

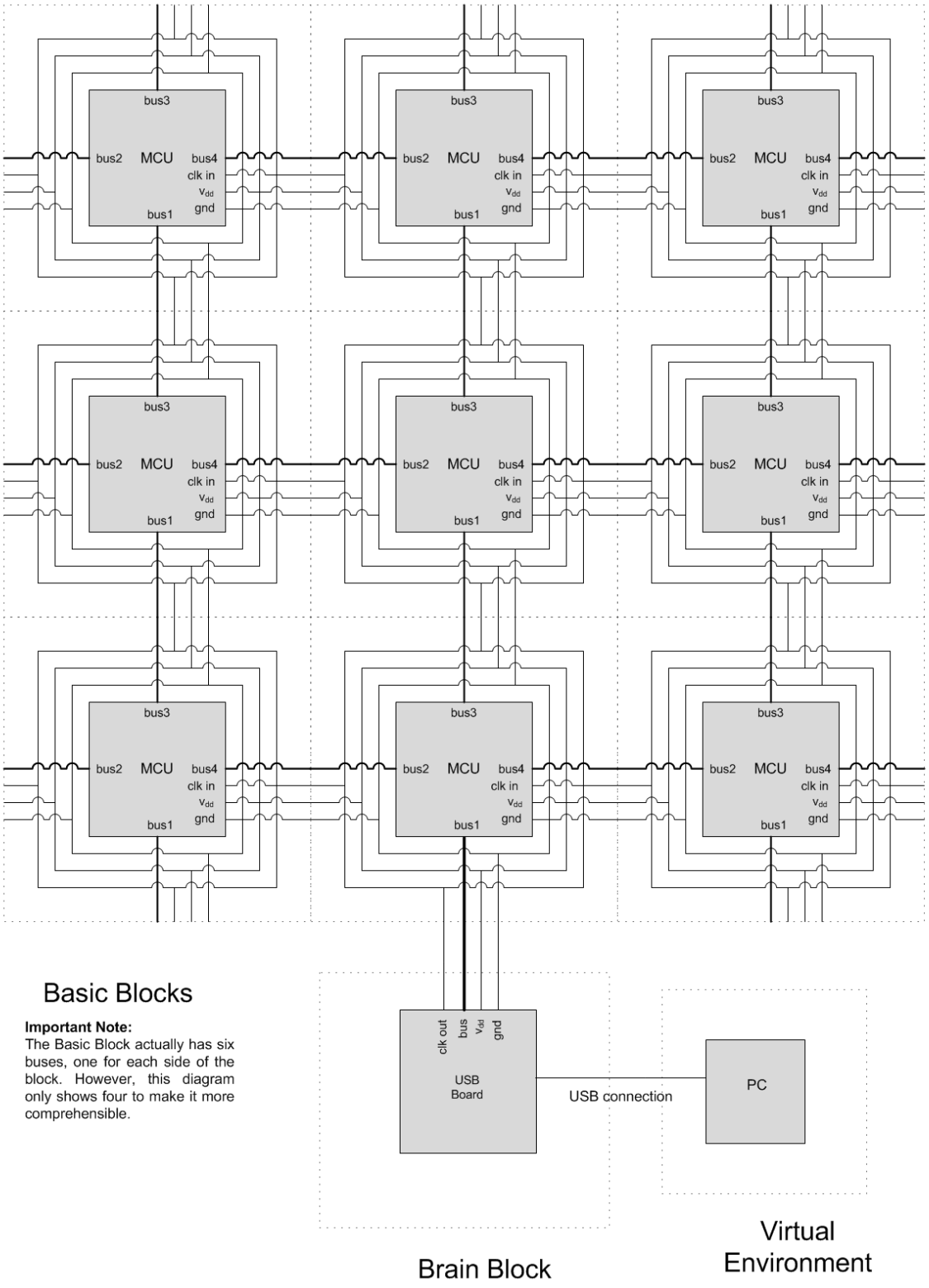


Figure 21: Top-Level Block Diagram

6.3 Basic Blocks

The Basic Block is just one microcontroller known as the MSP430F2013 as shown in Figure 22. There are no additional components and the selected microcontroller is from the Value Line series from Texas Instruments thereby making each block as inexpensive as possible. This is ideal since a DigiBlox structure will be using as many as 15 of these blocks. The selected MSP430 also has bidirectional pin capabilities which make it appropriate for the bidirectional nature of the connections involved in the algorithm.

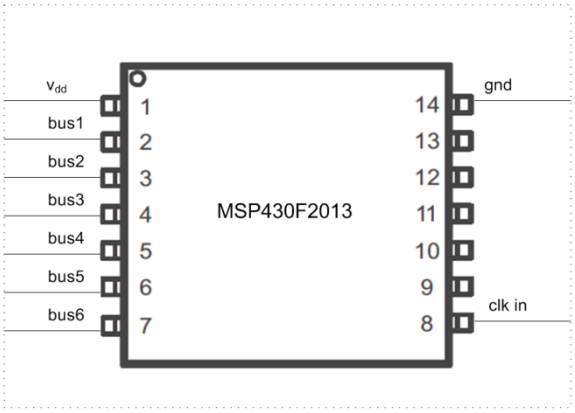


Figure 22: Schematic of the Basic Block

Figure 23 is a photograph of one of the MSP430F2013 used and the MSP-EXP430G2 Development Board that uploads program code into it. IAR Embedded Workbench was the integrated development environment used to write the code.

Figure 23: MSP430 and Development Board

6.3.1 Build Tree

The assembled blocks will first need to build a tree to determine the direction of the flow of information. As you can see in Figure 17, each block has an arrow indicating the direction at which to send data. Establishing the direction of the arrow for each block is the key to building this tree.

The code snippet in Figure 24 is the Main of the microcontroller code and the flowchart in Figure 25 describes the entire code snippet visually. Majority of the code snippet of Main consists of instructions on how the microcontroller implements the part where the blocks build the tree. Notice how Lines 60 to 93 loops forever (at least until the microcontroller loses power).

```
50 void main(void)
51 {
52
53   WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
54
55   char status = BIT0; // Initialize status to blank (001)
56
57   appendByte <<= 1; // Needed to prepare the append byte
58
59   while(1)
60   {
61     if (status & BIT0) // If status is blank (001), then enter
62     {
63       P1DIR &= ~(BIT5|BIT4|BIT3|BIT2|BIT1|BIT0); // Set P1.0-5 to input direction
64       for (char i=1;i^BIT6;i=i<<1) // Cycle through P1.0-5
65       {
66         if (i & P1IN) // If signal found in P1.0-5, then enter
67         {
68           direction = i; // Set direction to where signal was found
69           addDirectionToAppendByte(direction); // Add direction to append byte
70           status = BIT1; // Set status to gray (0010)
71           i=BIT5; // Set i to the end to exit the for loop
72         }
73       }
74     }
75
76     else if (status & BIT1) // If status is blank (010), then enter
77     {
78       P1DIR = ~direction; // Set P1.0-5 to output except for the direction bus
79       P1OUT = ~direction; // Send signal to the selected buses
80       status = BIT2; // Set status to black (0100)
81       swDelay(100000); // This delay is needed to ensure that contingent blocks reads sent signals
82     }
83
84     else if (status & BIT2) // If status is blank (100), then enter
85     {
86       P1DIR = direction; // Set output for parent pin and input for child pins
87       P1IE |= BIT6;                           // P1.6 interrupt enabled
88       P1IES |= BIT6;                          // P1.6 Hi/lo edge
89       P1IFG &= ~BIT6;                         // P1.6 IFG cleared
90
91       _BIS_SR(LPM4_bits + GIE);               // Enter LPM4 w/interrupt
92     }
93   }
94 }
```

Figure 24: Code Snippet of Main

A “starting” signal (a simple logic high) will be generated from the Brain Block to a bus of the Basic Block that it is connected to. Each Basic Block starts off with its status variable equal to 001. This means that it would always enter the if statement from Line 61 to 74 until its status changes. Embedded in this if statement is Line 63 which sets all the buses of Basic Blocks as inputs; therefore all Basic Blocks starts off

ready to receive the “starting” signal and cycles between all its buses (as you can see from the for loop in Line 64 to 72) so that when it detects the signal from a particular bus, it would set its direction variable to that which corresponds to it (as shown in Line 68). Upon detecting this signal, it would also change its status to 010 so that it would enter the if statement from Line 76 to 82 instead of the prior if statement thereby ceasing its search for the signal in case it receives that signal again. This new if statement is where the block sets its buses other than the one that received the signal to output (see Line 78) and regenerates that signal to those buses (see Line 79) so that the blocks that it is connected to will similarly receive that signal to repeat the whole process. After doing that it would once again change its status variable (see Line 80) to 100 so that it would start entering the if statement from Line 84 to 92 and stop entering the prior ones. This if statement is where the Send/Receive Bits process covered in section 6.3.2 occurs. It is where the block sends data in the direction according to its newly set direction variable and receives data from the remaining buses.

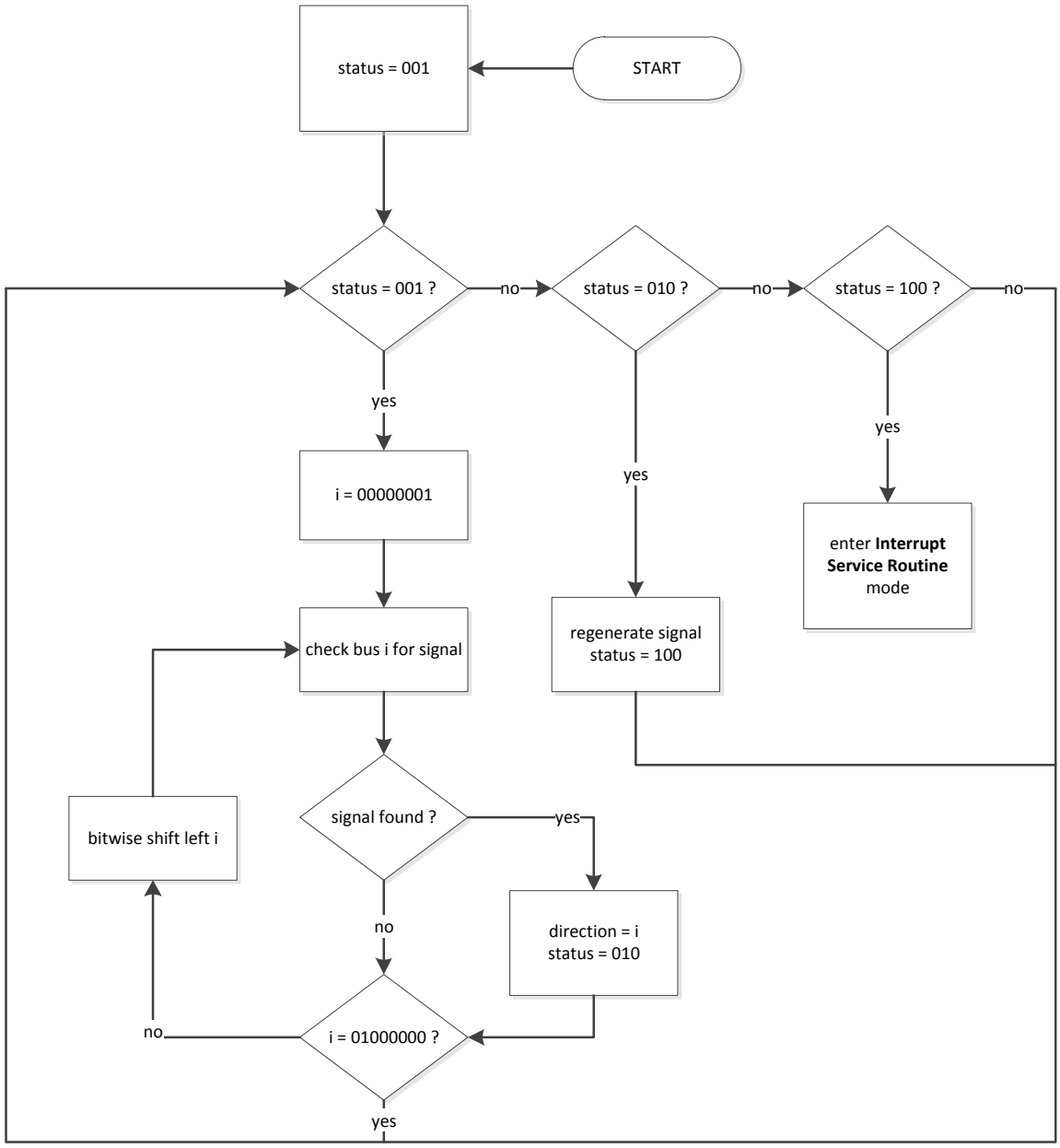


Figure 25: Flowchart of Main

In summary, the Brain Block generates the "starting" signal to the block that it is connected to. That block receives the signal, sets its direction variable to correspond to where the signal came from, regenerates the signal to neighboring blocks, stops looking for the signal and start sending data to a

connecting block according to its direction variable. The neighboring blocks will then receive the regenerated signal and repeat this entire process.

To test if the blocks can successfully build the tree, several microcontrollers had some of their buses connected together to represent a scenario of an assembly configuration. The image in Figure 26 shows these connections established using the black jumper wires. The blue wires and green wires respectively provide the power (3.3V) and ground supplied by an external board (called the Arduino USB Board) to the microcontrollers. The yellow wire is the source of the "starting" signal (3.3V logic high) which was again provided by the external board.

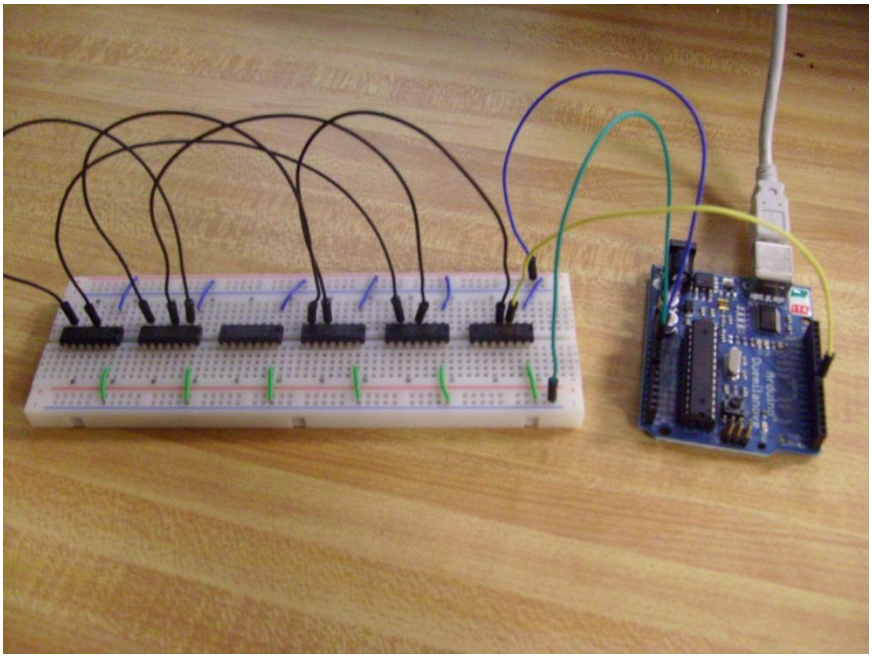


Figure 26: Testing the Build Tree

Changing the assembly configuration was a matter of rewiring the black wires and yellow wire to different buses of the microcontrollers. Checking what the direction variable equal to for each microcontroller was a matter of checking the logic value of pin 8, 9 and 10 using a logic analyzer since

the logic value of these pins corresponds to a certain value for the direction variable as shown in Table 2. Note that these pins were configured that way only for this test and not for the actual final code of the project.

Table 2: Expected Test Results

direction variable	Pin 8	Pin 9	Pin 10
00000001	0	0	0
00000010	0	0	1
00000100	0	1	0
00001000	0	1	1
00010000	1	0	0
00100000	1	0	1

A myriad of assembly configurations (at least 20) involving 1 to 6 blocks were tested successfully.

6.3.2 Send/Receive Bits

After establishing the direction each Basic Block is to send data, the next step is to actually send the data.

Once the Basic Block has established its direction variable and changes its status to 100, it would start entering the if statement in Line 84 to 90 as shown in the code snippet in Figure 24. Using the direction variable, the block will set the bus that it is to send data through to output and the buses that it expects to receive data from to input as shown in Line 86. Then it would enter the Interrupt Service Routine mode as shown in Lines 87 to 91.

The Interrupt Service Routine is the key to synchronize the blocks when they transfer data between each other to the clock signal provided by the Brain Block. At the falling edge of that clock signal, the blocks will enter the Interrupt Service Routine, which basically means it would execute the sequence of instructions in the code snippet in Figure 27 every time it detects a falling edge of the clock signal.

```
 96 // Port 1 interrupt service routine
 97 #pragma vector=PORT1_VECTOR
 98 __interrupt void Port_1(void)
 99 {
100   sendData(counter, direction); // Send a bit to parent block
101
102   readData(selector0, side0, BIT0); // Read a bit from bus1
103   readData(selector1, side1, BIT1); // Read a bit from bus2
104   readData(selector2, side2, BIT2); // Read a bit from bus3
105   readData(selector3, side3, BIT3); // Read a bit from bus4
106   readData(selector4, side4, BIT4); // Read a bit from bus5
107   readData(selector5, side5, BIT5); // Read a bit from bus6
108   // Note: The bus from parent block will always be a 0 bit
109
110   P1IFG &= ~BIT6;                          // P1.6 IFG cleared
111 }
```

Figure 27: Cope Snippet of Interrupt Service Routine

Since all the blocks enter this Interrupt Service Routine at the same time due to them using the same clock signal, this means that when one block sends a bit, the receiving block is ready to receive the bit. Line 100 is where the block sends a bit and lines 102 to 107 are where the block looks for a sent a bit. On the next falling edge of the clock signal, the Interrupt Service Routine is executed again so that a new bit is sent and a new bit is received if any. Of course this means that the wavelength of the clock signal must be sufficiently long enough so that the next falling edge is not encountered before the tasks in the Interrupt Service Routine is completed, nor should it be too long that it would contribute to latency. This issue will be explored and resolved in section 6.4.1.

This proceeding part describes the test that was done to see if the microcontrollers could transfer bits from Basic Block to Basic Block and from Basic Block to Brain Block simultaneously with a clock signal generated by the Brain Block.

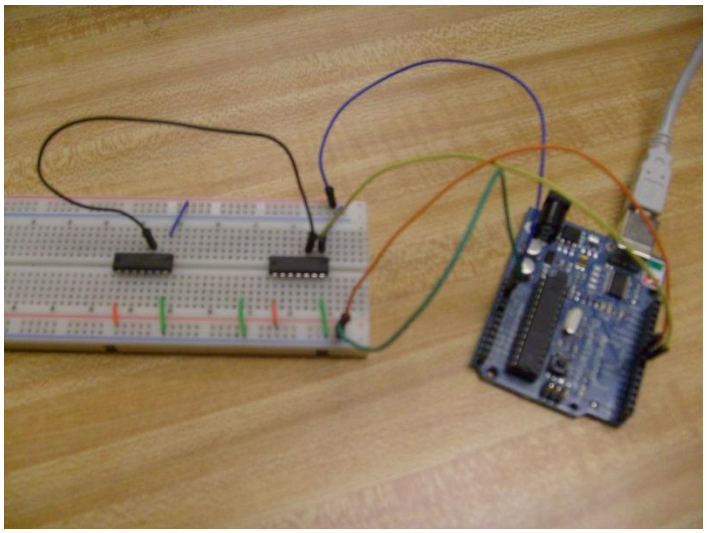


Figure 28: Testing the Send/Receive Bits

The test basically uses two Basic Blocks and the Brain Block connected to a PC. Each Basic Block has power, ground and a clock signal supplied by the Brain Block, which is the Arduino USB Board shown at the far right of the picture in Figure 28. The first block is connected to the second block which is also connected to the Brain Block. The first block will be sending a bit sequence that alternates between 0 and 1. The second block should receive the bits and send them to the Brain Block. The Brain Block should finally send those bits to display on the serial monitor of the PC.

This test was successful since the serial monitor of the PC continuously displayed “01010101...” and so on and so forth.

6.3.3 Data Manipulation

The PC in the example in Figure 29 should receive this information:

1111111100010010111111110001001000100100111111110001001000010010111111110001001000010010000100100010011000000000… (and more 0’s until the DigiBlox is powered off).

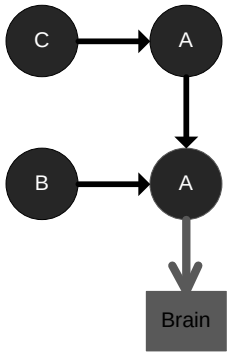


Figure 29: Example Configuration

Symbolically, these bits represent the following: $↑A$↑A←B$↑A↑A$↑A↑A←C!

To break it down further:

11111111	0001	0010	11111111	0001	0010	0010	0100	11111111	0001	0010	0001	0010	11111111	0001	0010	0001	0010	0010	0110	00000000
$	↑	A	$	↑	A	←	B	$	↑	A	↑	A	$	↑	A	↑	A	←	C	!

This is all the information the PC needs to render the block configuration in Figure 29. $ is the header indicating the start of a new packet. Each packet contains information regarding the type and location of one of the block. The first packet is ↑A which means that there is an A block up (↑) from the Brain Block. The second packet is ↑A←B which means that there is a B block, determined by the last letter in the packet, up and left (↑←) from the Brain Block. The third packet is $↑A↑A which means that there is an A block up and up (↑↑) from the Brain Block. The fourth and final packet is $↑A↑A←C which means that there is a C block up, up and left (↑↑←) from the Brain Block. The exclamation mark means that there are no more packets. The table below shows what sequence of bits corresponds to what symbolic value. Note that even though some of the symbols have the same bit sequence, they are

expected at different places relative to the most recent header; therefore the similar bit sequences would not cause a misreading.

Table 3: Bit Sequence Definitions

Symbol	Bit Sequence	Description
$	11111111	Header
A	0010	Block A
B	0100	Block B
C	0110	Block C
↑	0001	Up
←	0010	Left
X	0011	Forward
→	0100	Right
O	0101	Backward
↓	0110	Down
!	00000000	Ender

Now that the bit sequence has been defined, the next step is to code how a block is to handle all the bits it receives from other blocks and what to do with that information before it sends the bits to the next block. Remember that it needs to send a bit as it receives a bit from each bus.

To describe this data manipulation process, let's use the example in Figure 29. Once again, the block connected to the Brain Block needs to send the following bit sequence:

$↑A$↑A←B$↑A↑A$↑A↑A←C!

This bit sequence is an amalgamation of bits received from contiguous blocks and an additional bit sequence known as the appendByte. When a bit is received from a bus, say bus1, it is stored in a bit of a byte making up the array side1. Similarly, a bit received from bus2 will be stored in a bit of a byte making up the array side2. This is the same all the way until bus6 which corresponds to array side6. The code snippet of the function that reads a bit from the selected bus and places it in the appropriate array is shown in Figure 30.

```
122 void readData(char selector[], char side[], char BIT) // Read bits from the bus into respective side[]
123 {
124   static int headerStatus = 0;
125   static int enderStatus = 0;
126
127   if (BIT & P1IN) // Searching for header
128     if (headerStatus < 7)
129       headerStatus++;
130   else
131     if (headerStatus < 7)
132       headerStatus = 0;
133
134   if (headerStatus == 8) // Header found
135     if (BIT & P1IN)
136       changeBits(selector, side, 1); // Change the selected bit in side[] to 1
137     else
138       changeBits(selector, side, 0); // Change the selected bit in side[] to 0
139     incrementSelector(selector);
140
141   if (BIT & P1IN == 0) // Searching for ender
142     if (enderStatus < 7)
143       enderStatus++;
144   else
145     if (enderStatus < 7)
146       headerStatus = 0;
147 }
```

Figure 30: Code Snippet for Read Data

Now back to the example configuration of Figure 29, the block connected to the Brain Block would receive and store the following bits in its arrays (side1 to side6, all of which are of size 120):

side1	0	1	2	3	4	5	…	119
byte value	11111111	00010010	11111111	00010010	00100110	00000000	…	00000000
	$	↑ A	$	↑ A	← C	!		

side2	0	1	2	3	4	5	…	119
byte value	11111111	00100100	00000000	00000000	00000000	00000000	…	00000000
	$	← B						

side3	0	1	2	3	4	5	…	119
byte value	00000000	00000000	00000000	00000000	00000000	00000000	…	00000000

side4	0	1	2	3	4	5	…	119
byte value	00000000	00000000	00000000	00000000	00000000	00000000	…	00000000

side5	0	1	2	3	4	5	…	119
byte value	00000000	00000000	00000000	00000000	00000000	00000000	…	00000000

side6	0	1	2	3	4	5	…	119
byte value	00000000	00000000	00000000	00000000	00000000	00000000	…	00000000

The appendByte, which originally only contained information regarding the block type and then later changed to include the direction (see Line 69 of Figure 24), would have this value:

appendByte	0
byte value	00010010
	↑ A

As the block receives and stores the bits into the arrays, the blocks simultaneously retrieve bits from the arrays and appendByte to create a sequence of bits it needs to send as shown in Figure 31. The block

looks for headers in each array. When a header is found, the sequence of bits between it and the next header (or ender) will be a packet. This packet needs to be appended with a new header and the appendByte before it could be sent out. So in the example in Figure 31, the packet ↑A←C was found in array side1. Before this packet is sent out, the bit sequence $↑A needs to appended to the front of it to make the new packet $↑A ↑A←C. That's just one of the four packets being sent out in the example in Figure 31. The other packets being sent out are $↑A, $↑A ←B and $↑A ↑A.

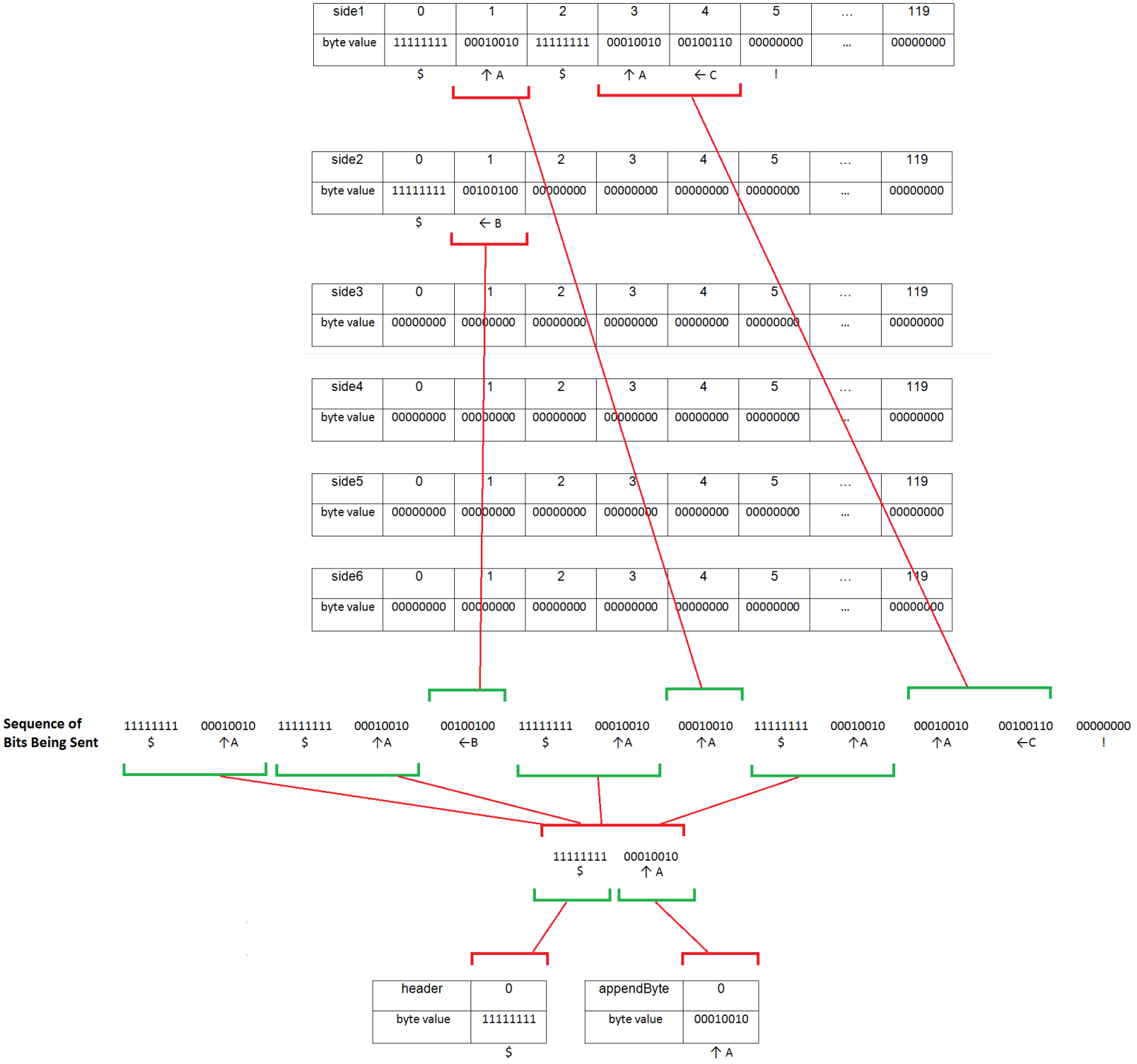


Figure 31: Example of Data Manipulation

That basically describes the data manipulation process. The complexity of the data manipulation process arises from the fact that the microcontroller has limited memory. It would be significantly easier code wise to store the information of the bits as a queue data structure of integer type, but that requires a significant amount of memory since *one bit of a receive/sent data will actually take more than six bytes when stored*! This is because 1) each piece of data in a queue data structure needs a pointer which takes up two to four bytes of memory space and 2) even though integer type is easier to code with, it alone takes up four bytes.

The way the data manipulation was handled in this project paid careful attention to memory constraints. The code made use of arrays of type char with each bit of the char data (that's two bytes per char data) representing the bit of the sent/receive data. This means that *one bit of a receive/sent data will take exactly one bit when stored*!

Notice that each array has 120 data elements. This is not an arbitrary number. According to calculations, if there are 14 blocks all together, the maximum number of elements that one array can have when the blocks are assembled is 120.

The assembly configuration that will cause this maximum to be reached would be 14 blocks connected in a snake-esque fashion like the one below with an additional Brain Block connected at an end.

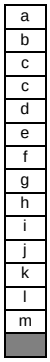


Figure 32: Snake-esque Configuration

The block immediately connected to the Brain Block will encounter an array that is filled with

number of elements in the packet to determine Block a + number of elements in the packet to determine Block b + … + number of elements in the packet to determine Block l + number of elements in the packet to determine Block m + element for ender = 15 + 14 + … + 2 + 1 = 120 *elements*

Hence, the array sizes are set to 120 instead of some arbitrary size which could lead to too little space for the DigiBlox to work or too much space causing the code to fail to compile.

The design of the data manipulation code seems appropriate for the constraints of the MSP430F2013.

This code also marks the end of the code for the Basic Block.

6.3.4 Mechanical Construction

This section describes the mechanical design of the blocks. Each block should physically be a cube-shaped structure that can be grasped with one hand and should also be quick and easy to connect and disconnect from other blocks.

Each block will be made of three of the structure similar to the one in Figure 33 assembled in the way shown in Figure 34 using readily available angle brackets.

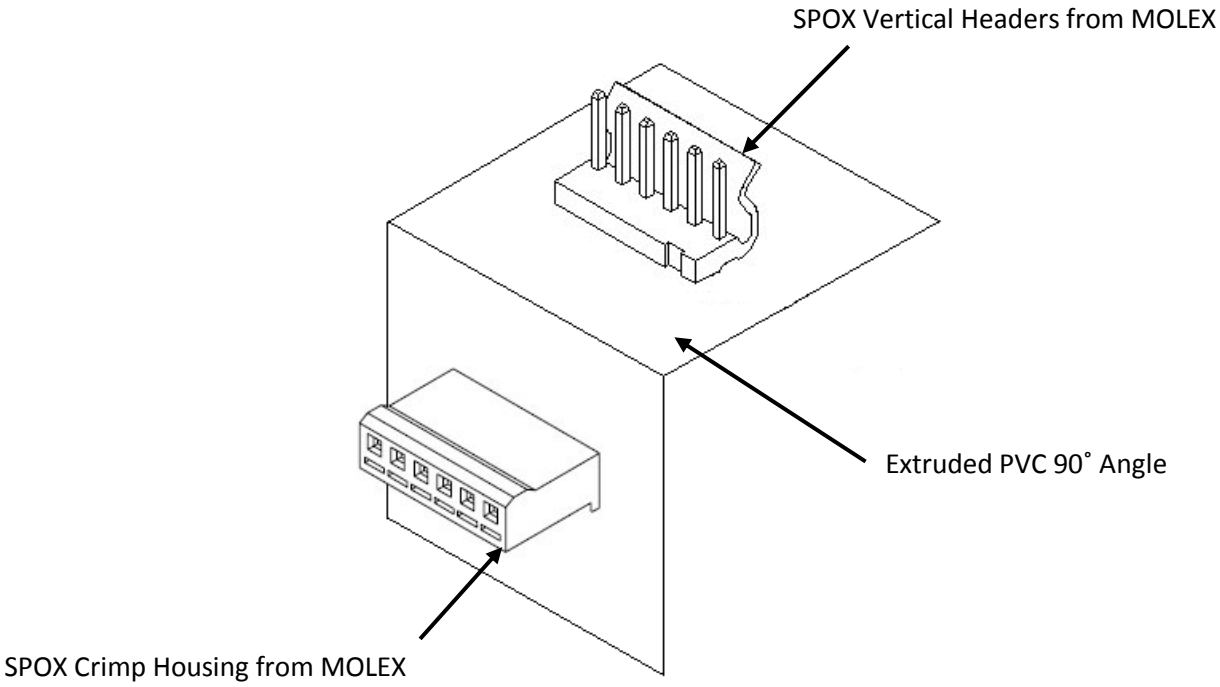


Figure 33: A Third of a Block

The SPOX connectors are resistance-hold connectors which mean that you could easily connect and disconnect them making them ideal for this project. PVC was chosen because they are light and easy to machine compared to metal and wood. They are also sufficiently strong and very durable.

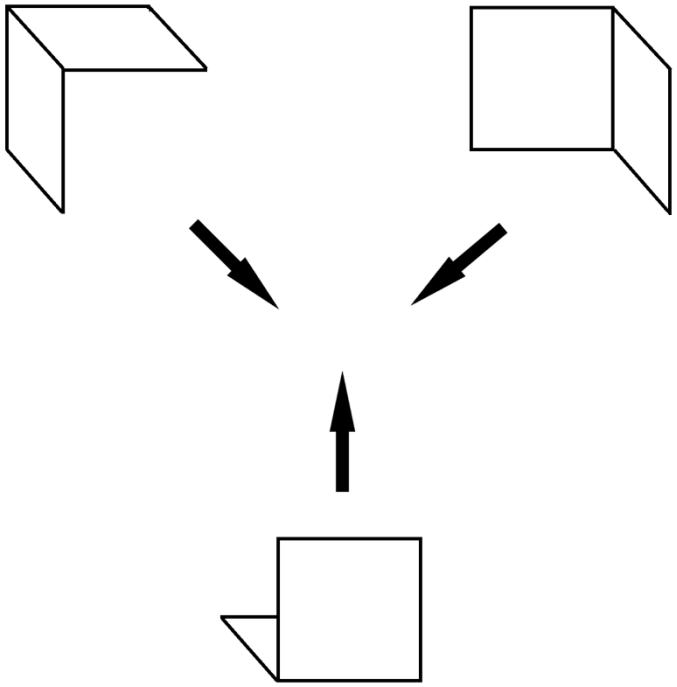


Figure 34: Assembled to Make a Cube

The three pieces assembled together in Figure 34 are not the same, that is, they are not exactly as shown in Figure 33, but they only differ in the number of pins of the connector. The drawing shown in Figure 35 roughly illustrates the complete block after the three pieces have been assembled together.

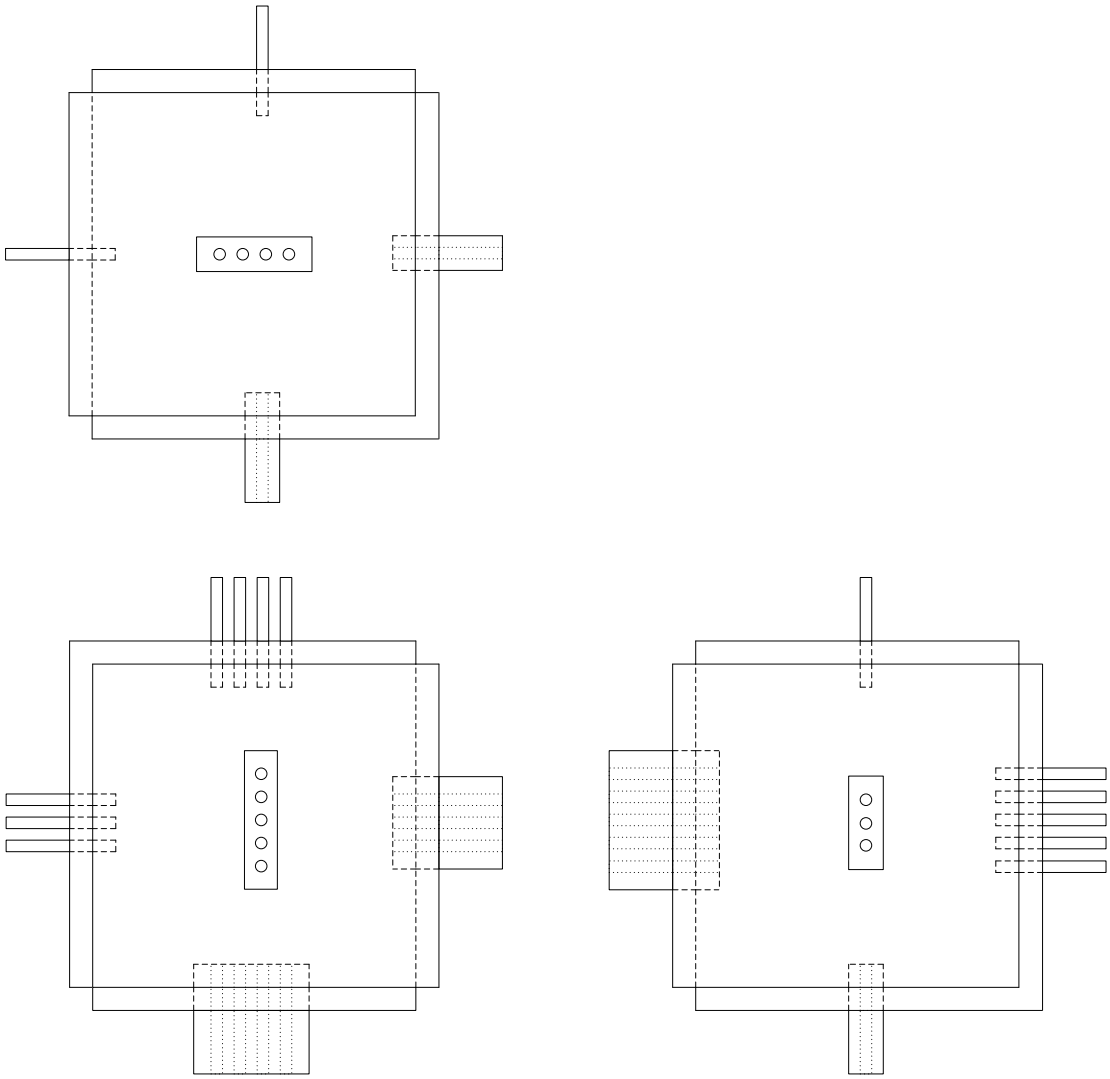


Figure 35: Front, Side and Top View of a Block

A block will basically be a cube consisting of a 4-pin male connector on one side with a 4-pin female connector on the opposite side, a 5-pin male connector on another side with a 5-pin female connector on the opposite side, and a 6-pin male connector on the remaining sides with a 6-pin female connector on the opposite side. Although some of the sides have up to 6 pins, only 4 of the pins are actually in use (ground, V_{dd}, the bus line and the clock signal). The mismatch number of pins would mean that only specific sides of two blocks can connect to each other thereby providing the forcing function required for the algorithm to work.

Each block will house the microcontroller chip sitting on a 14-pin IC socket. Wires from the leads of the IC socket would be soldered to the appropriate pins of the SPOX connectors. Since the IC socket will be the piece permanently attached to the rest of the parts, the MSP430 could therefore be easily replaced. The image below shows the inside of the block with the connectors attached.

Figure 36: Photograph of a Basic Block

Table 4 lists all the parts needed for the mechanical assembly of the DigiBlox.

Table 4: Parts List for Mechanical Design

Manufacturer	Part Description	Part Number	Qty
Molex	3.96mm (.156") Pitch SPOX™ Crimp Housing, Female, With Friction Ramp, 4 Circuits	0009501041	14
Molex	3.96mm (.156") Pitch SPOX™ Crimp Housing, Female, With Friction Ramp, 5 Circuits	0009501051	14
Molex	3.96mm (.156") Pitch SPOX™ Crimp Housing, Female, With Friction Ramp, 6 Circuits	0009501061	14
Molex	3.96mm (.156") Pitch SPOX™ and KK® Wire-to-Board Header, Vertical, with Friction Lock, 3 Circuits	0009652048	14
Molex	3.96mm (.156") Pitch SPOX™ and KK® Wire-to-Board Header, Vertical, with Friction Lock, 4 Circuits	0009652058	14
Molex	3.96mm (.156") Pitch SPOX™ and KK® Wire-to-Board Header, Vertical, with Friction Lock, 5 Circuits	0009652068	14
U.S. Plastic Corps.	2"x2"x1/4" Gray PVC-1 Extruded Angle	45031	14
Amplifier Parts	14-Pin DIP IC Socket	9000026	14

6.4 Brain Block

The function of the Brain Block is to supply the clock signal, power and ground to all the other blocks, receive data from only one other block (whichever block the user connects the Brain Block to) and send data to the PC via USB. The schematic for the Brain Block is shown in Figure 37.

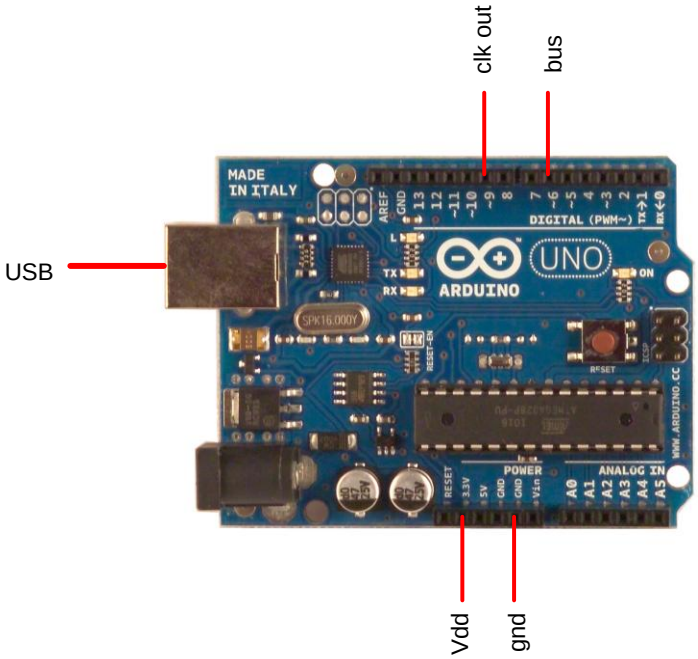


Figure 37: Schematic of Brain Block

The Brain Block is just the Arduino USB Board which is a popular development board for USB interfaces and therefore is suitable for the USB aspect of this project. The development board also comes with its own proprietary integrated development tool called the Arduino Software Suite.

6.4.1 Generate Clock Signal

Although the Interrupt Service Routine described rigorously in section 6.3.2 begins on the falling edge of an external clock, the code within the Interrupt Service Routine itself runs on an internal one.

The selected internal clock was the f_{MCLK} according to line 53 of the code snippet in Figure 24. The frequency of this clock is 1MHz according to the datasheet snippet in Table 5 circled in red.

Table 5: Datasheet Snippet of MSP4302013

PARAMETER		TEST CONDITIONS	T_A	VCC	MIN	TYP	MAX	UNIT
$I_{AM, 1MHz}$	Active mode (AM) current (1MHz)	$f_{DCO} = f_{MCLK} = f_{SMCLK} = 1MHz$, $f_{ACLK} = 32,768Hz$, Program executes in flash, BCSCTL1 = CALBC1_1MHZ, DCOCTL = CALDCO_1MHZ, CPUOFF = 0, SCG0 = 0, SCG1 = 0, OSCOFF = 0		2.2 V		220	270	µA
				3 V		300	370	

Using the timing tool available in IAR Embedded Workbench, the code within the Interrupt Service Routine uses approximately 2220 clock cycles from f_{MCLK}. This means that the clock to be generated from the Brain Block must be at least 2220 / 1M = 2ms in wavelength.

```
const int clkPin = 9;
const int busPin = 6;

void setup() {
  Serial.begin(9600); // Standard bit rate for USB
  pinMode(clkPin,OUTPUT); // Set clock pin as output
  pinMode(busPin,INPUT); // Set bus pin as input
  digitalWrite(clkPin, HIGH); // Clock starts at high
}

void loop() { // Loop forever
  digitalWrite(clkPin, LOW); // Falling edge of clock
  delayMicroseconds(3000);        // wait for 3 milliseconds
  bitReceived = digitalRead(busPin);

  if (bitReceived == HIGH) { // Send '1' to PC if received 1
    Serial.print('1');
  }
  else if (bitReceived == LOW)  { // Send '0' to PC if received 0
    Serial.print('0');
  }
  digitalWrite(clkPin, HIGH); // Rising edge of clock
}
```

Figure 38: Code Snippet for Brain Block

The code snippet in Figure 38 shows that the clock signal to be generated would be 3ms where the extra millisecond was added in case of system uncertainties. Oscilloscope reading of the clock signal showed negligible discrepancy in the wavelength which means that the clock signal is reliable enough.

6.4.2 Send/Receive Bits

The Brain Block only connects to one other block therefore only has one bus other than its USB connection to the PC.

The Arduino is primarily a USB interface therefore it has predefined libraries specific for USB communication: both for the PC end and the board end. The code snippet circled in red in Figure 39 basically shows that the Arduino will send a '1' ASCII character to the PC whenever it receives a 1 bit from the connected block or a '0' ASCII character if it receives a 0 bit instead.

```
const int clkPin = 9;
const int busPin = 6;

void setup() {
  Serial.begin(9600); // Standard bit rate for USB
  pinMode(clkPin,OUTPUT); // Set clock pin as output
  pinMode(busPin,INPUT); // Set bus pin as input
  digitalWrite(clkPin, HIGH); // Clock starts at high
}

void loop() { // Loop forever
  digitalWrite(clkPin, LOW); // Falling edge of clock
  delayMicroseconds(3000);        // wait for 3 milliseconds
  bitReceived = digitalRead(busPin);

  if (bitReceived == HIGH) { // Send '1' to PC if received 1
    Serial.print('1');
  }
  else if (bitReceived == LOW)  { // Send '0' to PC if received 0
    Serial.print('0');
  }
  digitalWrite(clkPin, HIGH); // Rising edge of clock
}
```

Figure 39: Code Snippet for Brain Block with Highlights

6.5 Virtual Environment

As mentioned, the PC needs to be able to use the information sent by the blocks to render the assembled blocks in the Virtual Environment. Furthermore, the Virtual Environment needs to be able to clearly showcase the rendered blocks. The Virtual Environment was created in Unity, an integrated authoring tool commonly used for game development. It has tools that can be used to generate polygonal worlds with advance camera controls, model texturing and peripheral device support; thus it is suited for the purpose of this project.

A preview of the world that was created out of blocks of different colors is shown in Figure 40.

Figure 40: Early Version of Virtual Environment

The code snippet below takes preexisting assets already available in Unity libraries such as 3D blocks, assign them colors and place them in positions determined by values in the serial port of a USB. This shows that this early version of the Virtual Environment can retrieve information from devices connected via USB and use it to alter objects in the virtual world.

```
using UnityEngine;
using System.Collections;

public class LevelCreationScript : MonoBehaviour {

    private Controllers controllerScript;
    private int[] levelState;
    private bool levelChanged = false;
    public Transform[] platforms;


    void Start () {
        controllerScript = (Controllers)gameObject.GetComponent("Controllers");
        levelState = new int[8];
        for(int i = 0; i < 8; i++)
            levelState[i] = 0;
    }


    void FixedUpdate () {
        for( int i = 0; i < 8; i++)
        {
            if(controllerScript.pos[i] != levelState[i])
                levelChanged = true;
        }
        if(levelChanged)
        {
            for(int i = 0; i < 8; i++)
            {
                if(controllerScript.pos[i] != levelState[i])
                {
                    GameObject col = GameObject.FindWithTag("col" + i);
                    if(col != null)
                    {
                        if(controllerScript.pos[i] != 0)
                        {
                            GameObject plat = GameObject.Instantiate(platforms[ controllerScript.pos[i] - 1 ].gameObject,
                                col.transform.position, Quaternion.identity) as GameObject;
                            plat.transform.parent = col.transform;
                            plat.transform.Translate(0, (controllerScript.pos[i] - 1) * 10 + 10, 0);
                        }

                        else
                            col.transform.DetachChildren();
                    }
                }
            }
            levelChanged = false;
            for(int k = 0; k < 8; k++)
                levelState[k] = controllerScript.pos[k];
        }
    }
}
```

Figure 41: Script Used to Generate Blocks in Virtual Environment

7 Project Results

To demonstrate the proof-of-concept, the test structures in Figure 5, Figure 6 and Figure 7 should be able to be built in the real world and rendered in the virtual world. Unfortunately, there is a consistent error in the sequence of bits being transferred from the blocks to the PC.

In the test configuration shown in Figure 42, the PC should receive the following sequence of bits:

11111111	0001	0010	11111111	0001	0010	0010	0100	11111111	0001	0010	0001	0010	11111111	0001	0010	0001	0010	0010	0110	00000000
$	↑	A	$	↑	A	←	B	$	↑	A	↑	A	$	↑	A	↑	A	←	C	!

However, the bits highlighted in yellow are consistently incorrect. After testing many different assembly configurations, a patterned emerged suggesting that the two least significant bit of the byte that represent the directions, except for the one immediately after a header, are always wrong.

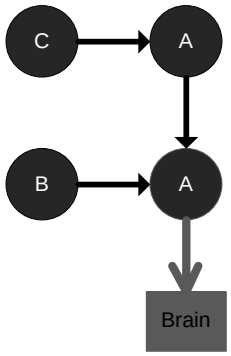


Figure 42: Simple Test Configuration

The consistent nature of the error suggests that the cause of the problem is not an electrical one, but a software one – there is a bug in the code. Since the data manipulation part of the code (covered in section 6.3.3) is the only untested part of the code, it is likely to be the part of the code that carries the bug. The fact that the error is to do with bit sequences also indicates this.

The complexity of the data manipulation part of the code is inherently necessary as it was coded in a way that is optimized to use as little memory as possible. Using a microcontroller that has more memory or installing more RAM into the current microcontroller would make this part of the code less complex. But it would bring up the price of each block significantly.

8 Budget Results

Table 6 lists the all the parts used in this project. The total money expected to be spent for this project was $64.44; however, the actual money spent was a little less than $90 due to extra parts purchased in case of part failure or damage.

Table 6: Complete Parts List

Manufacturer	Part Description	Part Number	Qty	Price
Molex	3.96mm (.156") Pitch SPOX™ Crimp Housing, Female, With Friction Ramp, 4 Circuits	0009501041	14	$0.20 each
Molex	3.96mm (.156") Pitch SPOX™ Crimp Housing, Female, With Friction Ramp, 5 Circuits	0009501051	14	$0.20 each
Molex	3.96mm (.156") Pitch SPOX™ Crimp Housing, Female, With Friction Ramp, 6 Circuits	0009501061	14	$0.20 each
Molex	3.96mm (.156") Pitch SPOX™ and KK® Wire-to-Board Header, Vertical, with Friction Lock, 3 Circuits	0009652048	14	$0.18 each
Molex	3.96mm (.156") Pitch SPOX™ and KK® Wire-to-Board Header, Vertical, with Friction Lock, 4 Circuits	0009652058	14	$0.18 each
Molex	3.96mm (.156") Pitch SPOX™ and KK® Wire-to-Board Header, Vertical, with Friction Lock, 5 Circuits	0009652068	14	$0.18 each
U.S. Plastic Corps.	2"x2"x1/4" Gray PVC-1 Extruded Angle	45031	14	$12.80 total
Amplifier Parts	14-Pin DIP IC Socket	9000026	14	$0.22 each
Texas Instruments	TI LaunchPad Development Board	MSP-EXP430G2	1	$4.30 each
Texas Instruments	Microcontroller – Value Line Series, DIP	MSP430F2013	14	$0.45 each
Arduino	Arduino Uno – USB Board with Software Suite	n/a	1	$22 each
Unity	Unity 3.1 – Integrated Authoring Tool	n/a	1	WPI carries license
IAR Systems	IAR Embedded Workbench – Integrated Development Environment	n/a	1	used free version
TOTAL				64.44

9 Conclusion

This major qualifying project set out to show how the DigiBlox would work. An algorithm that utilizes the network of connections of the assembled blocks to determine its assembly configuration was developed and mathematically proven. The implementation of this algorithm, however, yielded mixed results.

The first part of the algorithm was to build a tree where the nodes of the tree were the blocks themselves. This was done successfully. The next part of the algorithm required each node (a.k.a. block) to retrieve data from all its child nodes, manipulate the data, and then send it to its parent. The retrieval and sending of data worked fine, but the data manipulation part had a problem. The code for the data manipulation was discovered to have a bug. The code was optimized to ensure effective memory usage which was important since the DigiBlox used low-memory microprocessors; however, this significantly increased the complexity of the code making it difficult to determine the location of the bug. Having more memory to allow easier coding was suggested.

Other than having access to more memory, future work for the DigiBlox could include having blocks of different shapes instead of just different colors. Actual Lego blocks come in many different shapes so this addition would make playing with the blocks more interesting.

10 Bibliography

AR ToolKit. (n.d.). Retrieved 10 29, 2010, from Human Interface Technology Laboratory: http://www.hitl.washington.edu/artoolkit/

Bluejay, M. (n.d.). *How Much Electricity do Household Items Use?* Retrieved 10 29, 2010, from Micheal Bluejay: http://michaelbluejay.com/electricity/howmuch.html

Boden, M. (2004). *The Creative Mind.* Routledge.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms.* Cambridge, MA: The MIT Press.

Dimensions of a Standard Lego Brick. (n.d.). Retrieved 10 29, 2010, from Dimensions Guide: http://www.dimensionsguide.com/dimensions-of-a-standard-lego-brick/

Horn, M. (2007). Tangible Programming in the Classroom with Tern. *Conference on Human Factors in Computing Systems* (pp. 1965-1970). San Jose, California, USA: Associaton for Computing Machinery.

Hsu, J. (2000). *Electric Current Needed to Kill a Human*. Retrieved 10 29, 2010, from Hypertextbook: http://hypertextbook.com/facts/2000/JackHsu.shtml

Kitamura, Y. (2001). Real-time 3D Interaction with ActiveCube. *Conference on Human Factors in Computing Systems* (pp. 355-356). Association of Comupting Machinery.

Noble, J. (2009). Programming Interactivity. O'Reilly Media.

Shneiderman, B. (1998). *Designing the User Interface - Strategies for Effective Human-Computer Interaction* (3 ed.). Addison-Wesley.

Smith, A. C. (2007). Using Magnets in Physical Blocks that Behave as Programming Objects. *Tangible, Embedded and Embodied Interaction*, (pp. 147-150). Los Angeles, USA.

11 Appendices

11.1 Source Code of Basic Block

```
//*******************************************************************************
//  DigiBlox
//  Anas Maghfur
//*******************************************************************************

#include  <msp430x20x3.h>
#define MAX_POS_SEL 120
#define MAX_BIT_SEL 7
#define MAX_SIDE_SEL 5
#define TYPE 00000001 // red=00000001 green=00000010 blue=00000011

//***************************Function Prototypes********************************

void swDelay(unsigned int max_cnt); // Software delay
void readData(char selector[], char side[], char BIT);
void sendData(char counter[], char direction);
void changeBits(char selector[], char side[], char newVal);
void incrementSelector(char selector[]);
void incrementCounter(char counter[]);
char retrievedBitByCounter(char counter[]);
char retrieveBitFromByte(char byte, char bitPos);
void addDirectionToAppendByte(char direction);

//***************************Global Variables***********************************

char counter[3] = {0,0,MAX_BIT_SEL};

char appendByte = TYPE;

char selector0[2] = {1,MAX_BIT_SEL};
char selector1[2] = {1,MAX_BIT_SEL};
char selector2[2] = {1,MAX_BIT_SEL};
char selector3[2] = {1,MAX_BIT_SEL};
char selector4[2] = {1,MAX_BIT_SEL};
char selector5[2] = {1,MAX_BIT_SEL};

// buffers for each side
char side0[MAX_POS_SEL+1];
char side1[MAX_POS_SEL+1];
char side2[MAX_POS_SEL+1];
char side3[MAX_POS_SEL+1];
char side4[MAX_POS_SEL+1];
char side5[MAX_POS_SEL+1];

char direction = 0;


//*********************************Main*****************************************

void main(void)
```

```
{

  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

  char status = BIT0; // Initialize status to blank (001)

  appendByte <<= 1; // Needed to prepare the append byte

  while(1)
  {
    if (status & BIT0) // If status is blank (001), then enter
    {
      P1DIR &= ~(BIT5|BIT4|BIT3|BIT2|BIT1|BIT0); // Set P1.0-5 to input direction
      for (char i=1;i^BIT6;i=i<<1) // Cycle through P1.0-5
      {
        if (i & P1IN) // If signal found in P1.0-5, then enter
        {
          direction = i; // Set direction to where signal was found
          addDirectionToAppendByte(direction); // Add direction to append byte
          status = BIT1; // Set status to gray (0010)
          i=BIT5; // Set i to the end to exit the for loop
        }
      }
    }

    else if (status & BIT1) // If status is blank (010), then enter
    {
      P1DIR = ~direction; // Set P1.0-5 to output except for the direction bus
      P1OUT = ~direction; // Send signal to the selected buses
      status = BIT2; // Set status to black (0100)
      swDelay(100000); // This delay is needed to ensure that contingent blocks reads sent signals
    }

    else if (status & BIT2) // If status is blank (100), then enter
    {
      P1DIR = direction; // Set output for parent pin and input for child pins
      P1IE |= BIT6;                       // P1.6 interrupt enabled
      P1IES |= BIT6;                      // P1.6 Hi/lo edge
      P1IFG &= ~BIT6;                     // P1.6 IFG cleared

      _BIS_SR(LPM4_bits + GIE);           // Enter LPM4 w/interrupt
    }
  }
}

// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
  sendData(counter, direction); // Send a bit to parent block

  readData(selector0, side0, BIT0); // Read a bit from bus1
  readData(selector1, side1, BIT1); // Read a bit from bus2
  readData(selector2, side2, BIT2); // Read a bit from bus3
  readData(selector3, side3, BIT3); // Read a bit from bus4
  readData(selector4, side4, BIT4); // Read a bit from bus5
  readData(selector5, side5, BIT5); // Read a bit from bus6
  // Note: The bus from parent block will always be a 0 bit

  P1IFG &= ~BIT6;                     // P1.6 IFG cleared
}

//*******************************Functions************************************

void swDelay(unsigned int max_cnt) // Software delay
{
  unsigned int cnt=0;
  while (cnt < max_cnt)
    cnt++;
}
```

```
void readData(char selector[], char side[], char BIT) // Read bits from the bus into respective side[]
{
  static int headerStatus = 0;
  static int enderStatus = 0;

  if (BIT & P1IN) // Searching for header
    if (headerStatus < 7)
      headerStatus++;
  else
    if (headerStatus < 7)
      headerStatus = 0;

  if (headerStatus == 8) // Header found
    if (BIT & P1IN)
      changeBits(selector, side, 1); // Change the selected bit in side[] to 1
    else
      changeBits(selector, side, 0); // Change the selected bit in side[] to 0
    incrementSelector(selector);

  if (BIT & P1IN == 0) // Searching for ender
    if (enderStatus < 7)
      enderStatus++;
  else
    if (enderStatus < 7)
      headerStatus = 0;
}

void sendData(char counter[], char direction) // Retrieve bit from buffer and send to parent
{
  static int sendStatus = 0;
  char temp;

  char retrievedBit = retrievedBitByCounter(counter);

  if (sendStatus <= 7)
  {
    if (retrievedBit == 1)
    {
      sendStatus++;
      P1OUT = direction;
    }
    else
    {
      sendStatus = 0;
      P1OUT = 0;
    }
  }
  else if (sendStatus > 7 && sendStatus <= 15)
  {
    switch(sendStatus)
    {
      case 8:
        temp = 7;
      case 9:
        temp = 6;
      case 10:
        temp = 5;
      case 11:
        temp = 4;
      case 12:
        temp = 3;
      case 13:
        temp = 2;;
      case 14:
        temp = 1;
      case 15:
        temp = 0;
    }
    if (retrieveBitFromByte(appendByte, temp) == 1)
```

```
      P1OUT = direction;
    else
      P1OUT = 0;
    sendStatus++;
  }
  else
  {
    if (retrievedBit == 1)
      P1OUT = direction;
    else
      P1OUT = 0;
    incrementCounter(counter);
  }
}

void changeBits(char selector[], char side[], char newVal) // Use this to change the value of bits in chosen side
{
    if (newVal == 1)
    {
      switch(selector[1])
      {
        case 0:
          side[selector[0]] |= BIT0;
        case 1:
          side[selector[0]] |= BIT1;
        case 2:
          side[selector[0]] |= BIT2;
        case 3:
          side[selector[0]] |= BIT3;
        case 4:
          side[selector[0]] |= BIT4;
        case 5:
          side[selector[0]] |= BIT5;
        case 6:
          side[selector[0]] |= BIT6;
        case 7:
          side[selector[0]] |= BIT7;
      }
    }
    else
    {
      switch(selector[1])
      {
        case 0:
          side[selector[0]] ^= BIT0;
        case 1:
          side[selector[0]] ^= BIT1;
        case 2:
          side[selector[0]] ^= BIT2;
        case 3:
          side[selector[0]] ^= BIT3;
        case 4:
          side[selector[0]] ^= BIT4;
        case 5:
          side[selector[0]] ^= BIT5;
        case 6:
          side[selector[0]] ^= BIT6;
        case 7:
          side[selector[0]] ^= BIT7;
      }
    }
}

void incrementSelector(char selector[]) // Increment the selectors
{
  if (selector[0] <= MAX_POS_SEL) // Increment the position select until MAX_POS_SEL
    selector[0]++;

  if (selector[1] > 0) // Decrement bit select until 0 then reset back to its maximum value and start over
    selector[1]--;
```

```
  else
    selector[1] = MAX_BIT_SEL;
}

void incrementCounter(char counter[]) // Increment the counter which keeps track of the bit to send
{
  if (counter[2] == 0)
  {
    counter[2] = MAX_BIT_SEL;
    counter[1]++;
  }
  else
    counter[2]--;

  if (counter[1] == MAX_POS_SEL)
  {
    counter[1] = 0;
    counter[0]++;
  }
  else
    counter[2]++;

  if (counter[0] == MAX_SIDE_SEL)
    counter[0] = 0;
  else
    counter[0]++;
}

char retrievedBitByCounter(char counter[]) // Uses the counter variable to retrieve the correct bit from a byte
{
  switch(counter[0])
  {
    case 0:
      return retrieveBitFromByte(side0[counter[1]], counter[2]);
    case 1:
      return retrieveBitFromByte(side1[counter[1]], counter[2]);
    case 2:
      return retrieveBitFromByte(side2[counter[1]], counter[2]);
    case 3:
      return retrieveBitFromByte(side3[counter[1]], counter[2]);
    case 4:
      return retrieveBitFromByte(side4[counter[1]], counter[2]);
    case 5:
      return retrieveBitFromByte(side5[counter[1]], counter[2]);
  }
}

char retrieveBitFromByte(char byte, char bitPos) // Retrieves the selected bit from a byte
{
  int bitVal;

  switch(bitPos)
  {
    case 0:
      bitVal = byte & BIT0;
    case 1:
      bitVal = byte & BIT1;
    case 2:
      bitVal = byte & BIT2;
    case 3:
      bitVal = byte & BIT3;
    case 4:
      bitVal = byte & BIT4;
    case 5:
      bitVal = byte & BIT5;
    case 6:
      bitVal = byte & BIT6;
    case 7:
      bitVal = byte & BIT7;
  }
```

```
  if (bitVal == 0)
    return 0;
  else
    return 1;
}

void addDirectionToAppendByte(char direction) // Add the direction bit sequence to the append byte
{
  switch(direction)
  {
    case 00000001:
      appendByte |= BIT4;
    case 00000010:
      appendByte |= BIT5;
    case 00000100:
      appendByte |= (BIT5|BIT4);
    case 00001000:
      appendByte |= BIT6;
    case 00010000:
      appendByte |= (BIT6|BIT4);
    case 00100000:
      appendByte |= (BIT6|BIT5);
  }
}
```

11.2 Source Code of Brain Block

BrainBlock

```
const int clkPin = 9;
const int busPin = 6;

void setup() {
  Serial.begin(9600); // Standard bit rate for USB
  pinMode(clkPin,OUTPUT); // Set clock pin as output
  pinMode(busPin,INPUT); // Set bus pin as input
  digitalWrite(clkPin, HIGH); // Clock starts at high
}

void loop() { // Loop forever
  digitalWrite(clkPin, LOW); // Falling edge of clock
  delayMicroseconds(3000);        // wait for 3 milliseconds
  bitReceived = digitalRead(busPin);

  if (bitReceived == HIGH) { // Send '1' to PC if received 1
    Serial.print('1');
  }
  else if (bitReceived == LOW)  { // Send '0' to PC if received 0
    Serial.print('0');
  }
  digitalWrite(clkPin, HIGH); // Rising edge of clock
}
```

11.3 Source Code for Virtual Environment

```
using UnityEngine;
using System.Collections;

public class LevelCreationScript : MonoBehaviour {

    private Controllers controllerScript;
    private int[] levelState;
    private bool levelChanged = false;
    public Transform[] platforms;


    void Start () {
        controllerScript = (Controllers)gameObject.GetComponent("Controllers");
        levelState = new int[8];
        for(int i = 0; i < 8; i++)
            levelState[i] = 0;
    }


    void FixedUpdate () {
        for( int i = 0; i < 8; i++)
        {
            if(controllerScript.pos[i] != levelState[i])
                levelChanged = true;
        }
        if(levelChanged)
        {
            for(int i = 0; i < 8; i++)
            {
                if(controllerScript.pos[i] != levelState[i])
                {
                    GameObject col = GameObject.FindWithTag("col" + i);
                    if(col != null)
                    {
                        if(controllerScript.pos[i] != 0)
                        {
                            GameObject plat = GameObject.Instantiate(platforms[ controllerScript.pos[i] - 1 ].gameObject,
                                col.transform.position, Quaternion.identity) as GameObject;
                            plat.transform.parent = col.transform;
                            plat.transform.Translate(0, (controllerScript.pos[i] - 1) * 10 + 10, 0);
                        }

                        else
                            col.transform.DetachChildren();
                    }
                }
            }
            levelChanged = false;
            for(int k = 0; k < 8; k++)
                levelState[k] = controllerScript.pos[k];
        }
    }
}
```