Portland State UNIVERSITY

MASEEH COLLEGE OF ENGINEERING

PROJECT SUMMARY & REPORT

A Block of Code

Senior Capstone Project

Team Members:
Nathan BRYANT
Daniel FRISTER
Tyler HART
Jacob MICIKIEWICZ
Greg STROMIRE

Erebus Labs:
Dr. Mike BOROWCZAK
University of Wyoming
DR. Andrea BURROWS
PSU Advisor:
Roy KRAVITZ

'

June 10, 2015

Contents

1 Abstract

We prototyped a low-cost, tactile teaching aide capable of helping a wide range of K-12 students explore the concepts of computer programming. Using a set of physically manipulatable blocks, students can perform basic programming operations and interact with the programs they create. Verification of valid code structures provide immediate feedback for rapid learning and control of simple and fun outputs connected to the system guide students toward a goal for their projects.

Each of our blocks is a 4-layer PCB, enclosed in an acrylic box, with an ATtiny461a microcontroller and 3 LEDs to indicate power, status, and errors. A 1MΩ potentiometer, with an attached acrylic wheel, is connected to the microcontroller's ADC to act as a selector for a block's particular function, or token. Each block communicates over a limited SPI bus to two adjacent blocks to determine topology, and communicates back to a main processor over a TWI global bus. Our main processor board is a BeagleBone Black with a common Debian distribution running a Python-based lexer, Bison-based parser, and C-based interpreter to run our simplified grammar. The results are supplied to an output pipeline to drive modular components such as an LCD screen or an RGB LED, as well as send feedback directly to the blocks via their onboard LEDs.

The end product is a set of blocks that introduce computer programming to students for whom traditional teaching methods are ineffective, or would otherwise have no exposure, and encourages all students to pursue education in STEM.

2 Introduction

2.1 Motivation

Introducing students to programming and computer science is vital in our increasingly connected world. Yet the tools and educational methods to familiarize young minds to STEM subjects have not kept pace with the technologies themselves. The typical CS learning experience in front of a computer screen only engages a select few students with a compatible learning style, and often schools struggle to afford the necessary resources and computer labs. Our goal was to create an interactive learning tool that can engage a wider range of students for presenting computer science concepts.

2.2 Proposal

The package will be a set of twenty to thirty cubes at around two inches across on a side. The set will function as its own interface device, where the topological arrangement of the elements (blocks) will determine the program. The individual blocks must be identifiable as belonging to a specific programming-construct group, such as a simplified programming grammar. At a minimum these groups must include; numbers, variables, operators, and controls.

Additionally, where necessary, a block should be capable of indicating what value or function it has been assigned. Blocks should be easy to connect, allowing users to experiment with layouts, and also provide area-specific error feedback and thereby increasing understanding and limiting confusion.

Finally, the set should be open-source and sufficiently accessible, both by cost and teaching use, as to promote universal access and encourage development and expansion by the general public.

Project Requrements Sheet

Sponsor Requirements	Engineering Requirements	Justification
2	Device must use open source software.	Project should be expandable by others after the team finishes.
6	Device must have low power consumption.	Circuits will be enclosed inside a sealed shell, and should have a long enough operation on a single charge for a class session.
7,10	Device must be easily portable.	Target audience for the product is young to middle aged children.
7,10	Power will be provided by external power or rechargeable battery pack.	Device parts will be enclosed inside blocks, and user will not have to open any block to operate device.
8,10	Block function selection must be clearly visible.	Users cannot properly operate or build programs without knowing what each block represents.
10	Each block must be able to indicate proper operation and placement.	Users will need feedback while developing code to learn from their mistakes.
4,9	Essential programming elements must be represented in system by a block.	Functional programs require standard programing elements
4,9,10	Must compile and run simple programs.	Creating programs is the main function of the system, and necessary for instruction.
10	Device must produce an output based on the program compiled.	Users must have a useable/knowable result.
1,2,3	Circuits must be built from common components.	Custom ordered parts raise unit price and prevent product from being rebuilt without redesign.
1	Part selection for devices will be aimed at extended prices.	Final product will involve a larger number of blocks, and extended prices will be a better representation of actual production costs.

Sponsor Requirements

1	Low hardware production costs.
2	Open Source Hardware Design & Board (can use "closed source" components: ASICs, uC, etc.)
3	Must Have a Multi-Chip solution – e.g. no single SoC;
4	Fundamental grammar functioning, assignments, and binary operators.
5	Open software repository.
6	Low power operation.
7	Built in power source or power pack.
8	Blocks should have multiple possible functions.
9	Control structure blocks.
10	System must be accesible to novice coders, and is intended for use in a class room, by children of approximatley 10-14 years of age.

3 Project Planning

We identified three main elements for our system design: Physical Construction, Block Communication, and Program Interpretation. This enabled us to research and develop each subsystem in parallel with regular inter-group updates to stay unified on task. We also set several early milestones to ensure successful integration and to allow for multiple, iterative design revisions.

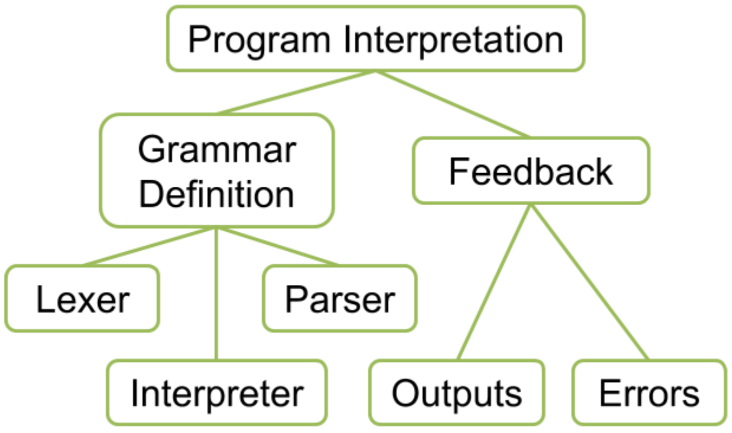


Fig 1

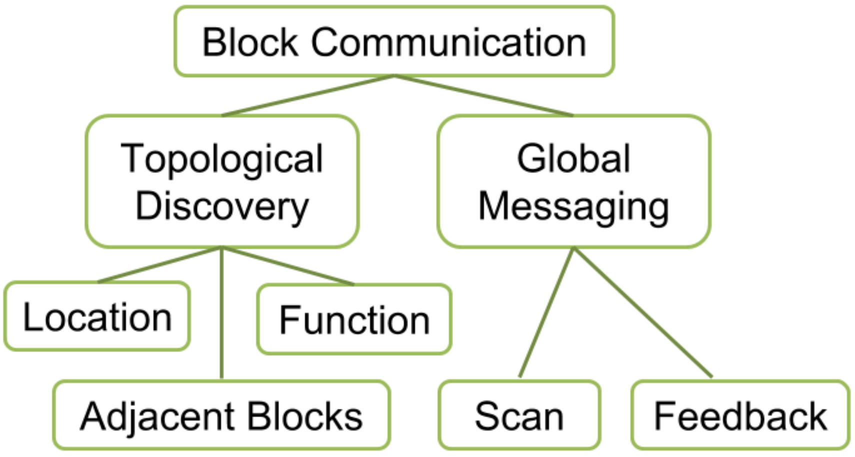


Fig 2

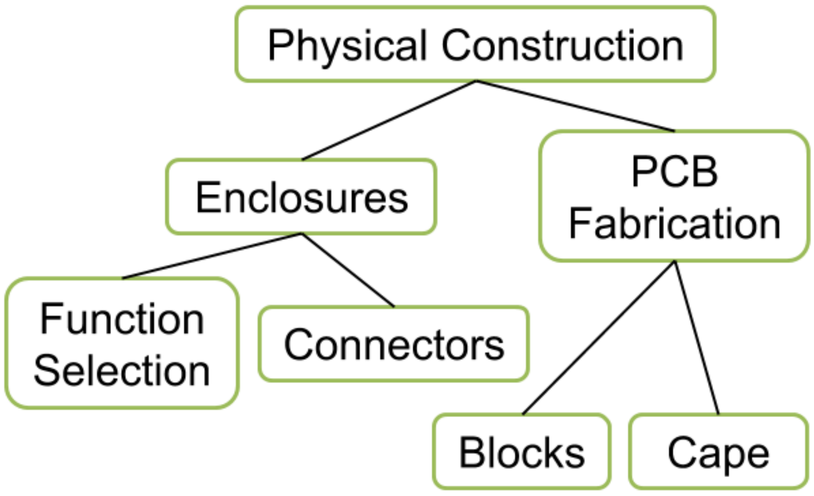
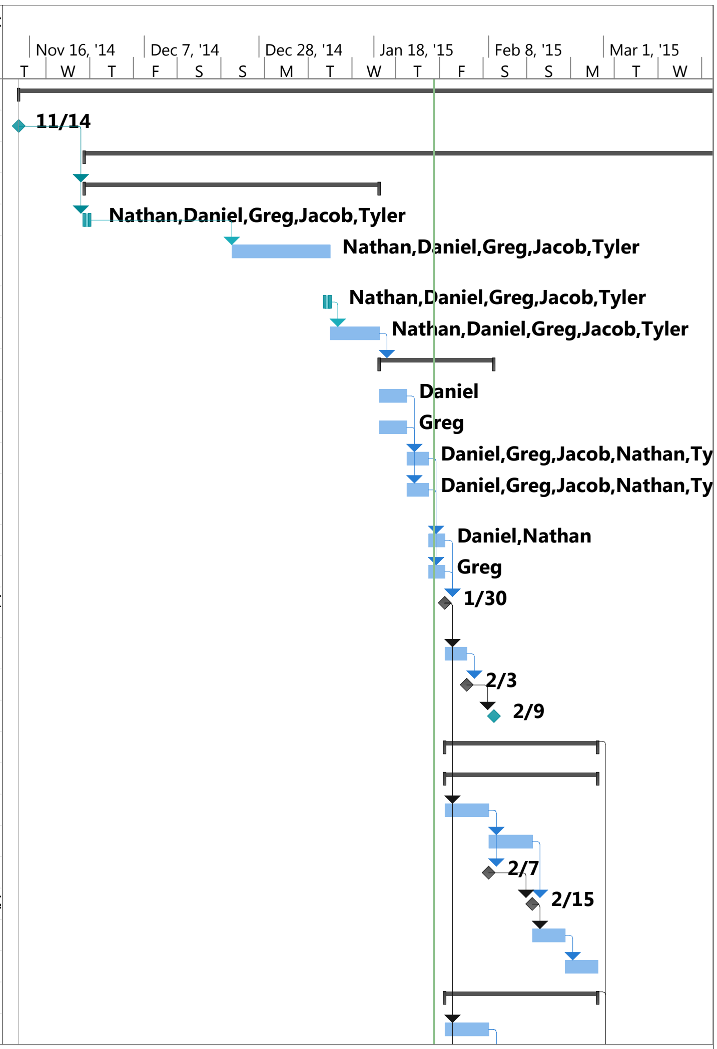


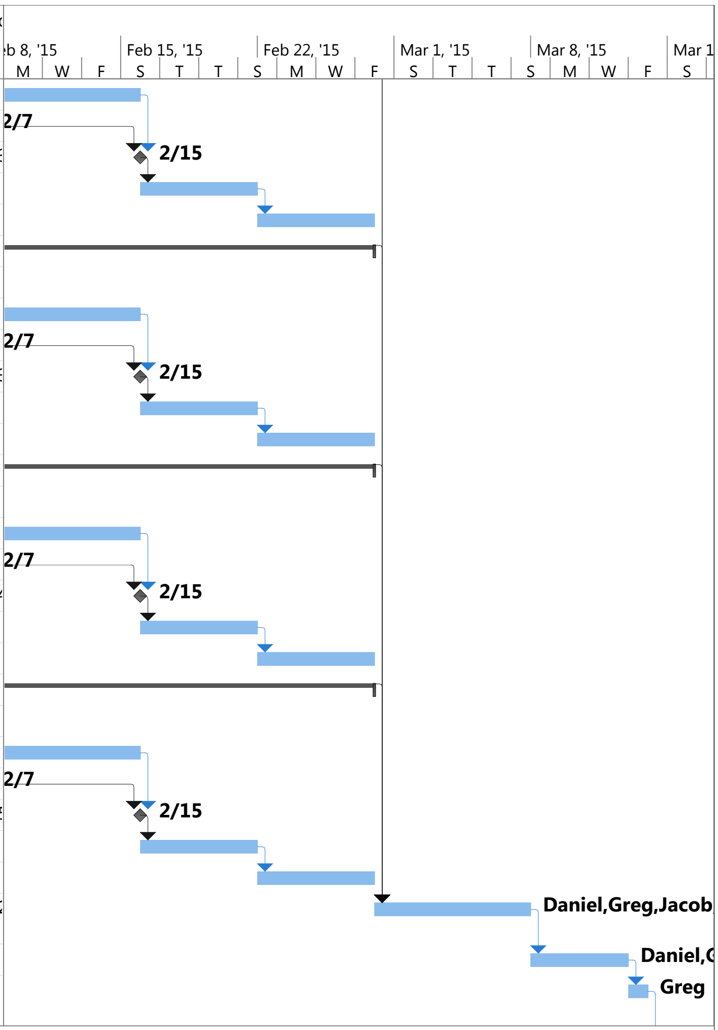
Fig 3

With these divisions the work was scheduled based on skill, interest and load leveling.

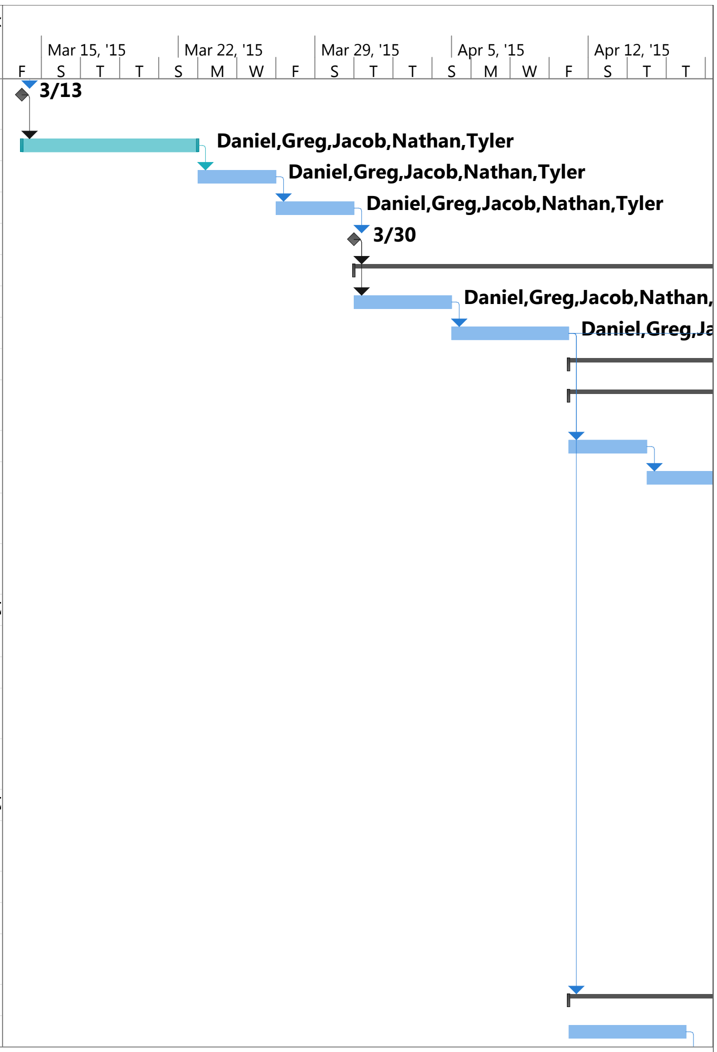
ID	Task Name	Duration	Start	Finish	Prec
1	**A Block of Code**	**211 days**	**Fri 11/14/14**	**Fri 6/12/15**	
2	Capstones Assigned	0 days	Fri 11/14/14	Fri 11/14/14	
3	**Complete Rev 1**	**125 days**	**Wed 11/26/14**	**Mon 3/30/15**	
4	**Generate Concepts**	**54 days**	**Wed 11/26/14**	**Sun 1/18/15**	**2**
5	Meet with Advisor	1 day	Wed 11/26/14	Wed 11/26/14	2
6	Document Ideas for 5 Design Areas	18 days	Tue 12/23/14	Fri 1/9/15	5
7	Meet with Sponsors	1 day	Fri 1/9/15	Fri 1/9/15	
8	Decide on Design Direction	9 days	Sat 1/10/15	Sun 1/18/15	7
9	**Propose Project**	**21 days**	**Mon 1/19/15**	**Mon 2/9/15**	**8**
10	Create Proposal Draft	5 days	Mon 1/19/15	Fri 1/23/15	
11	Create Project Schedule Dra	5 days	Mon 1/19/15	Fri 1/23/15	
12	Review Proposal with team	4 days	Sat 1/24/15	Tue 1/27/15	10
13	Review Project Schedule with team	4 days	Sat 1/24/15	Tue 1/27/15	11
14	Revise Proposal	3 days	Wed 1/28/15	Fri 1/30/15	12
15	Revise Project Schedule	3 days	Wed 1/28/15	Fri 1/30/15	13
16	Present Drafts to Advisor for Feedback	0 days	Fri 1/30/15	Fri 1/30/15	14,1
17	Revise Drafts for Submission	4 days	Sat 1/31/15	Tue 2/3/15	16
18	Submit Docs to Sponsors	0 days	Tue 2/3/15	Tue 2/3/15	17
19	Feedback from Sponsors	0 days	Mon 2/9/15	Mon 2/9/15	18
20	**Prototype 5 Design Areas**	**28 days**	**Sat 1/31/15**	**Fri 2/27/15**	
21	**Prototype Power**	**28 days**	**Sat 1/31/15**	**Fri 2/27/15**	
22	Research Methods	8 days	Sat 1/31/15	Sat 2/7/15	16
23	Research Best Practices	8 days	Sun 2/8/15	Sun 2/15/15	22
24	Order Parts	0 days	Sat 2/7/15	Sat 2/7/15	22
25	Receive Parts	0 days	Sun 2/15/15	Sun 2/15/15	23,2
26	Assemble Proof of Conce	6 days	Mon 2/16/15	Sat 2/21/15	25
27	Test Proof of Concept	6 days	Sun 2/22/15	Fri 2/27/15	26
28	**Prototype Communication**	**28 days**	**Sat 1/31/15**	**Fri 2/27/15**	
29	Research Methods	8 days	Sat 1/31/15	Sat 2/7/15	16



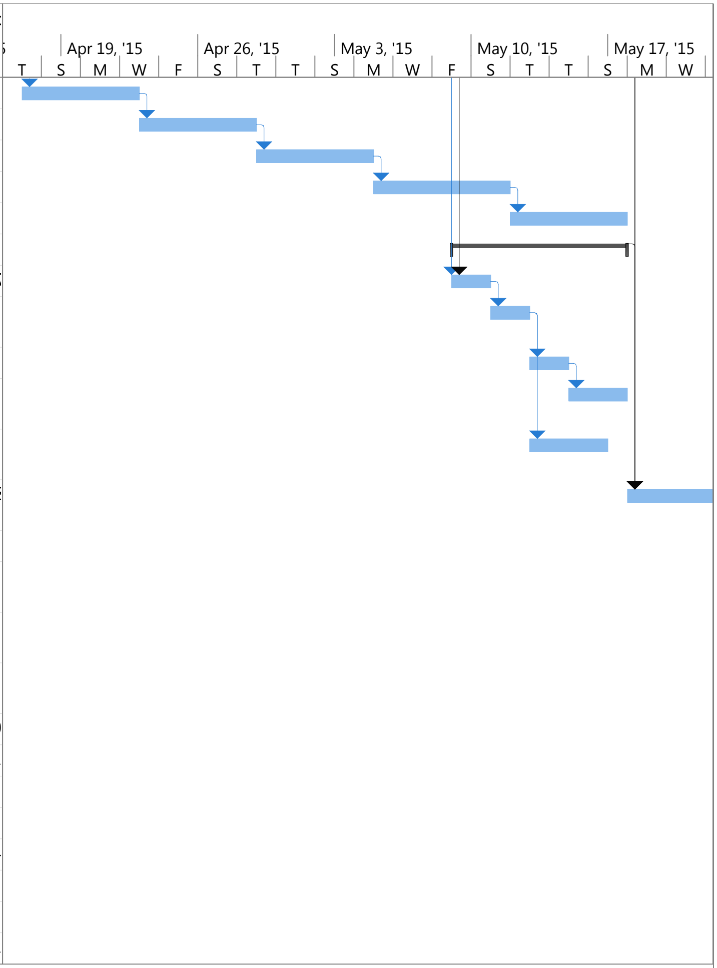
ID	Task Name	Duration	Start	Finish	Prec
30	Research Best Practices	8 days	Sun 2/8/15	Sun 2/15/15	29
31	Order Parts	0 days	Sat 2/7/15	Sat 2/7/15	29
32	Receive Parts	0 days	Sun 2/15/15	Sun 2/15/15	30,3
33	Assemble Proof of Conce	6 days	Mon 2/16/15	Sat 2/21/15	32
34	Test Proof of Concept	6 days	Sun 2/22/15	Fri 2/27/15	33
35	**Prototype Processing**	**28 days**	**Sat 1/31/15**	**Fri 2/27/15**	
36	Research Methods	8 days	Sat 1/31/15	Sat 2/7/15	16
37	Research Best Practices	8 days	Sun 2/8/15	Sun 2/15/15	36
38	Order Parts	0 days	Sat 2/7/15	Sat 2/7/15	36
39	Receive Parts	0 days	Sun 2/15/15	Sun 2/15/15	37,3
40	Assemble Proof of Conce	6 days	Mon 2/16/15	Sat 2/21/15	39
41	Test Proof of Concept	6 days	Sun 2/22/15	Fri 2/27/15	40
42	**Prototype Form Factor**	**28 days**	**Sat 1/31/15**	**Fri 2/27/15**	
43	Research Methods	8 days	Sat 1/31/15	Sat 2/7/15	16
44	Research Best Practices	8 days	Sun 2/8/15	Sun 2/15/15	43
45	Order Parts	0 days	Sat 2/7/15	Sat 2/7/15	43
46	Receive Parts	0 days	Sun 2/15/15	Sun 2/15/15	44,4
47	Assemble Proof of Conce	6 days	Mon 2/16/15	Sat 2/21/15	46
48	Test Proof of Concept	6 days	Sun 2/22/15	Fri 2/27/15	47
49	**Prototype UI/IO**	**28 days**	**Sat 1/31/15**	**Fri 2/27/15**	
50	Research Methods	8 days	Sat 1/31/15	Sat 2/7/15	16
51	Research Best Practices	8 days	Sun 2/8/15	Sun 2/15/15	50
52	Order Parts	0 days	Sat 2/7/15	Sat 2/7/15	50
53	Receive Parts	0 days	Sun 2/15/15	Sun 2/15/15	51,5
54	Assemble Proof of Conce	6 days	Mon 2/16/15	Sat 2/21/15	53
55	Test Proof of Concept	6 days	Sun 2/22/15	Fri 2/27/15	54
56	Integrate Modular Prototypes for Full System	8 days	Sat 2/28/15	Sat 3/7/15	20,2
57	Test Full System	5 days	Sun 3/8/15	Thu 3/12/15	56
58	Record Video of Prototyped System Demonstration	1 day	Fri 3/13/15	Fri 3/13/15	57



ID	Task Name	Duration	Start	Finish	Pred
59	Present Demonstration to Advisor and Sponsors for	0 days	Fri 3/13/15	Fri 3/13/15	58
60	[Break for Finals]	9 days	Sat 3/14/15	Sun 3/22/15	59
61	Revise Designs based on Feedb	4 days	Mon 3/23/15	Thu 3/26/15	60
62	Test Revisions	4 days	Fri 3/27/15	Mon 3/30/15	61
63	Rev 1 Complete	0 days	Mon 3/30/15	Mon 3/30/15	62
64	**Complete Rev 2**	**69 days**	**Tue 3/31/15**	**Sun 6/7/15**	**63**
65	Brainstorm Additional Features	5 days	Tue 3/31/15	Sat 4/4/15	63
66	Prototype Additional Features	6 days	Sun 4/5/15	Fri 4/10/15	65
67	**Complete Board Designs**	**37 days**	**Sat 4/11/15**	**Sun 5/17/15**	
68	**Complete ERC Error-Free Schematics**	**16 days**	**Sat 4/11/15**	**Sun 4/26/15**	
69	Research and Select Part	4 days	Sat 4/11/15	Tue 4/14/15	66
70	Design Schematics	4 days	Wed 4/15/15	Sat 4/18/15	69
71	Review Schematics Internal to Group	4 days	Sun 4/19/15	Wed 4/22/15	70
72	Send Schematics to Third Party for Review	4 days	Sun 4/19/15	Wed 4/22/15	70
73	Revise Schematic	4 days	Thu 4/23/15	Sun 4/26/15	71,7
74	**Complete DRC Error-Free La**	**12 days**	**Mon 4/27/15**	**Fri 5/8/15**	
75	Layout Boards	4 days	Mon 4/27/15	Thu 4/30/15	68
76	Review Layouts Internally to Group	4 days	Fri 5/1/15	Mon 5/4/15	75
77	Send Layouts to Third Party for Review	4 days	Fri 5/1/15	Mon 5/4/15	75
78	Revise Layouts	4 days	Tue 5/5/15	Fri 5/8/15	76,7
79	Print Small Run of Boards in PSU LID	2 days	Sat 5/9/15	Sun 5/10/15	78
80	Assemble Small-Run Boards	2 days	Mon 5/11/15	Tue 5/12/15	79
81	Test Small-Run Boards	2 days	Wed 5/13/15	Thu 5/14/15	80
82	Print and Assemble Remaining Boards	3 days	Fri 5/15/15	Sun 5/17/15	81
83	**Complete Code**	**37 days**	**Sat 4/11/15**	**Sun 5/17/15**	**66**
84	Code Block Identification	6 days	Sat 4/11/15	Thu 4/16/15	



ID	Task Name	Duration	Start	Finish	Prec
85	Code Program Execution	6 days	Fri 4/17/15	Wed 4/22/15	84
86	Code Error Detection	6 days	Thu 4/23/15	Tue 4/28/15	85
87	Code Block Communication	6 days	Wed 4/29/15	Mon 5/4/15	86
88	Test Full Code System	7 days	Tue 5/5/15	Mon 5/11/15	87
89	Revise System Components	6 days	Tue 5/12/15	Sun 5/17/15	88
90	**Complete Enclosures**	**9 days**	**Sat 5/9/15**	**Sun 5/17/15**	
91	Design Enclosures	2 days	Sat 5/9/15	Sun 5/10/15	66,7
92	Print and Assemble Small-Run of Enclosures	2 days	Mon 5/11/15	Tue 5/12/15	91
93	Revise Enclosure Design	2 days	Wed 5/13/15	Thu 5/14/15	92
94	Print and Assemble Remaining Enclosures	3 days	Fri 5/15/15	Sun 5/17/15	93
95	Consider Specialized Output Block Designs	4 days	Wed 5/13/15	Sat 5/16/15	92
96	Integrate Modular Prototypes for Full System	7 days	Mon 5/18/15	Sun 5/24/15	67,8
97	Test Full System	5 days	Mon 5/25/15	Fri 5/29/15	96
98	Record Video of Prototyped System Demonstration	1 day	Sat 5/30/15	Sat 5/30/15	97
99	Present Demonstration to Advisor and Sponsors for	0 days	Sat 5/30/15	Sat 5/30/15	98
100	Revise Designs based on Tests and Feedback	4 days	Sun 5/31/15	Wed 6/3/15	98
101	Test Revisions	4 days	Thu 6/4/15	Sun 6/7/15	100
102	Rev 2 Complete	0 days	Sun 6/7/15	Sun 6/7/15	101
103	**Create Presentation**	**14 days**	**Sat 5/30/15**	**Fri 6/12/15**	
104	Design Poster	4 days	Sat 5/30/15	Tue 6/2/15	97
105	Send Poster off for Print	1 day	Wed 6/3/15	Wed 6/3/15	104
106	Create Presentation Slides	4 days	Sat 5/30/15	Tue 6/2/15	97
107	Write "Curriculum"	4 days	Sun 5/31/15	Wed 6/3/15	98
108	Present	5 days	Mon 6/8/15	Fri 6/12/15	102



4 Methods and Tools

We utilized several methods and tools during all phases of the project, and some were used only for specific phases. We each used a mix of Windows, Macintosh, and Linux workstations throughout the project. We also used the AVR Toolchains (CrossPack, avr-gcc, avrdude) and the AVRISP MKII programmer in development, manufacturing, and testing. The BeagleBone Black was another major part of our full project cycle.

For our development phase, we utilized EagleCAD to design and layout prototype PCBs. We then moved to DesignSpark to allow us to develop 4-layer boards and reduce the surface area for a cheaper design. We employed an Arduino Uno and associated IDE to develop the TWI bus, initially with the smaller but compatible ATtiny85. We also used an Intronix Logicport 34-Channel Logic Analyzer to inspect communications, albeit much later in the process than necessary. We used a 3D printer to prototype a potential selector wheel.

During our manufacturing phase, we used the Portland State University Laboratory for Interconnected Devices to assemble both revisions of our prototype. We used their Torch T200N Reflow Oven to solder our second revision surface mount components. And we used their Full Spectrum Professional Series CO2 20x12 Laser to cut acrylic enclosures and selector wheels. We also used a handful of other Portland State tools such as soldering irons and a dremel.

To test our final boards, we used a microscope to inspect solder connections and then a general purpose multimeter to verify conductivity and isolation. We then used an Arduino Uno's regulated 3.3V output to supply power to a single and then a subset of blocks to confirm connectivity. We also wrote several scripts and test programs to explore the bounds and verify functionality of our set using an Arduino, a BeagleBone Black, and the blocks themselves.

5 Implementation

Our first development goal was a set of blocks that could identify their topology and return their token to a main processor. Though a 'smart-mat'design was considered, we decided it would be too dimensionally restrictive. We chose AVR microcontrollers and DB-9 connectors for their ease and cost. We utilized a localized SPI communication to establish location and a global TWI bus to report the block's assigned value. We then selected a Linux-based single board computer to harness existing language parsing and interpretation tools. The BeagleBone Black provided this at reduced power.

Our second set of development goals aimed to the increase block-set capability, provide portability, and expand the available output options:

- User-selectable subset of functions rather than a single, hard-coded token
- BeagleBone Controller Cape for power distribution and control of a global reset
- Additional modes of output connected to the Controller Cape to support student engagement in learning programming concepts

Each block is controlled by an ATtiny461 microcontroller, connected to two neighbors on a local bus based on SPI, and connected to all other blocks and the BeagleBone Black on a global bus using TWI. Both of these busses, as well as power and ground, are brought onto the block via a standard DB-9 connector. The token selector wheel is mounted on a 1MΩ potentiometer which feeds into the ADC on the microcontroller.

The local bus is started from the BeagleBone and initiates a handshake with the block connected below it. When the handshake completes, a vector is passed indicating the x- and y-coordinates of the sending block. The receiving block increments the vector based on whether the sending block is signaling from the above or the left line of the receiving block, and then stores and sends the incremented vector to the next block. The received vector is then translated into the TWI address on the global bus. The BeagleBone Black is able to query each block which responds with information on its adjacent blocks, its wheel category, and its current ADC value. The BeagleBone Black lexer uses this response data to construct a topology of the current block arrangement. It then passes this information to a parser and interpreter to generate the appropriate output such as driving a LCD screen.

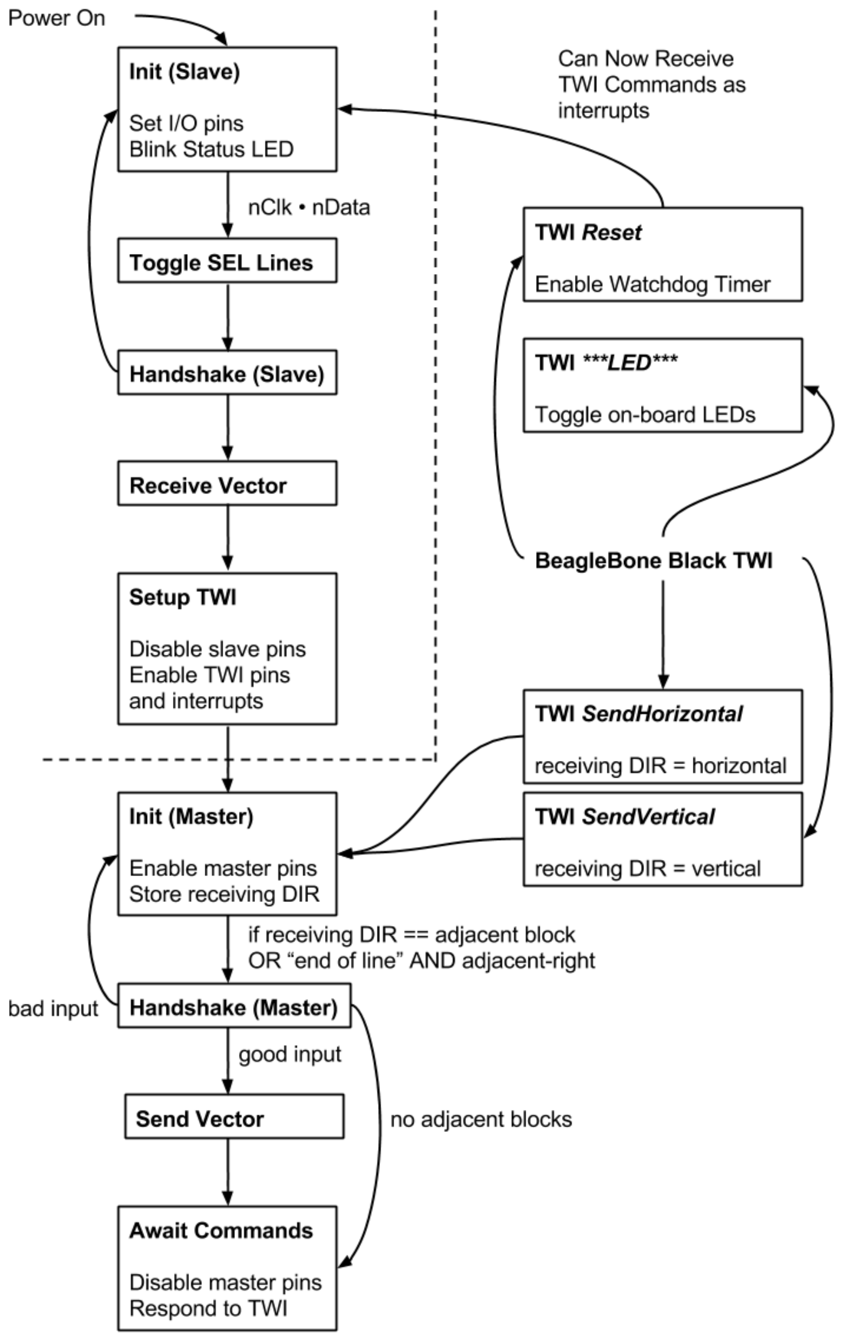
5.1 Block

The primary function of each block is to report back to the main processor board its own user-selected function as well as its adjacent blocks. This allows the main processor to collect the tokens and topological information necessary for constructing the user-generated program. A block progresses linearly through a series of states as outlined in the state diagram, Fig 4, on the following page.

5.1.1 Adjacent Blocks

First, the block will execute an initialization sequence that includes outputting to and reading from its adjacency lines. In typical usage, the blocks are built in a top-down orientation. A pair of slave-select pins, connected to the right and bottom connectors, are designated temporarily as inputs with their internal pull-up resistors enabled. Another pair of slave-select pins, connected to the left and top connectors, are designated temporarily as outputs and immediately pulled low.

For an example, consider the top block in Fig 5. If the block reads its adjacency pins and finds that its h_out line (horizontal output) is HIGH, then it has no block connected to its right (again, in a top-down orientation). If it reads its v_in is LOW, then a block is present below it. These values are stored and supplied to the main processor board in a global bus response (see TWI Commands).



Power On
Init (Slave)
Set I/O pins
Blink Status LED
nClk • nData
Toggle SEL Lines
Handshake (Slave)
Receive Vector
Setup TWI
Disable slave pins
Enable TWI pins
and interrupts
Init (Master)
Enable master pins
Store receiving DIR
if receiving DIR == adjacent block
OR "end of line" AND adjacent-right
bad input
Handshake (Master)
good input
Send Vector
no adjacent blocks
Await Commands
Disable master pins
Respond to TWI
Can Now Receive
TWI Commands as
interrupts
TWI Reset
Enable Watchdog Timer
TWI ***LED***
Toggle on-board LEDs
BeagleBone Black TWI
TWI SendHorizontal
receiving DIR = horizontal
TWI SendVertical
receiving DIR = vertical

Fig 4

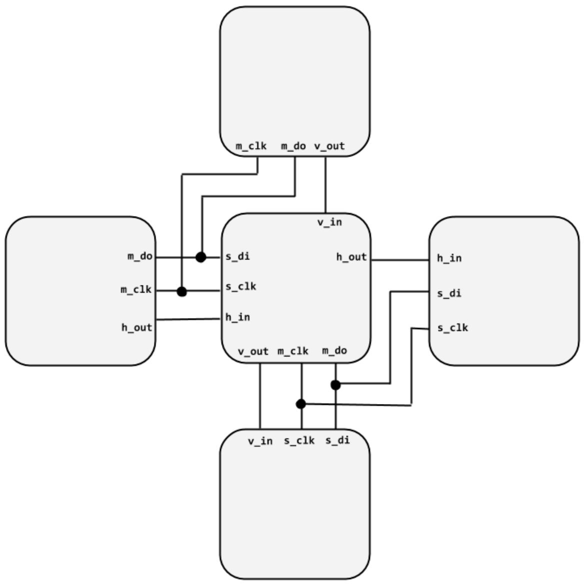
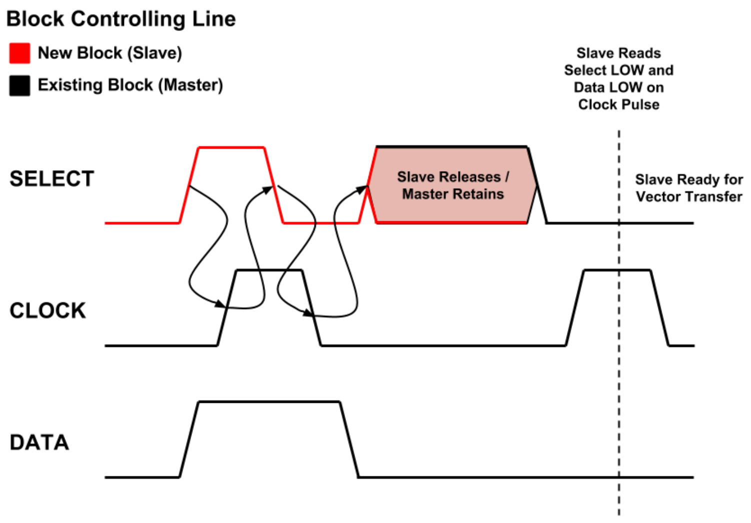


Fig 5

5.1.2 Handshake



Block Controlling Line
🟥 New Block (Slave)
⬛ Existing Block (Master)
Slave Reads Select LOW and Data LOW on Clock Pulse
SELECT
Slave Releases / Master Retains
Slave Ready for Vector Transfer
CLOCK
DATA

Fig 6

After the block is powered on and has executed its initialization sequence, including determining adjacent blocks, it prepares for a custom handshake to initiate a SPI transmission. Shown above is this handshake, where the just-powered-on block is considered the slave, and another block or the BeagleBone Black is considered the master.

1. First, the block will wait for the shared clock and data lines to go LOW. This will signal the block to start toggling its own select lines (v_in and h_in).
2. The master will then respond to the toggle that it sees (either horizontally or vertically) by bringing clock and data HIGH.
3. This triggers the select line LOW again and the clock and data lines again follow.
4. Next, the slave block brings its select line HIGH yet again but then releases control of it by switching the pin to an input and enable its internal pull-up resistor.
5. The master waits a short delay after seeing that rising edge then it changes its corresponding select line from an input to an output and pulls the line LOW.
6. A final clock pulse acts as a verification of the completed sequence before both blocks are prepared for a traditional SPI transmission.

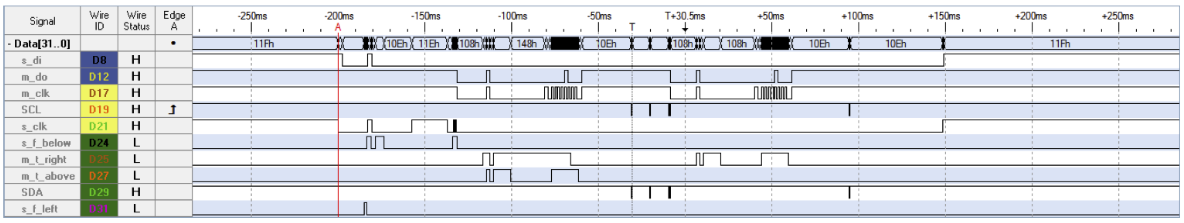
It is possible that an adjacent block could see the same clock and data lines go LOW and in fact complete almost the entire sequence. But the corresponding select line would not be held low on that final clock pulse and thus that block would recognize that the sequence was not meant for it, and reset its state. The same is true for any other sequence of edges that doesn't follow here – it would be considered bad input and cause a reset to initial handshake state.

5.1.3 SPI Transmission

Once a block has completed its handshake, it is prepared to receive data over a three-wire SPI protocol. This means there is no MISO signal involved. After the data is received, the block is able to continue to send the data to adjacent blocks as necessary. It will continue the direction it was originally sent (e.g. if received from above, send below; if received from the left, send right).

It will also recognize when it is considered the 'end of the line', which is considered the block with x-coordinate of 0 and no blocks connected below it. In this special case, it will then send to the right if there is a block present. Finally, a block will send in the vertical or horizontal directions on command from the global TWI bus (see TWI Commands).

Below is a screen capture from a logic analyzer showing 3 different SPI transmissions (in this example, blocks are implemented in a bottom-up orientation). The first transmission is to the current block which is acting as a slave and receiving data. The handshake begins at about the -200ms mark and completes roughly at -135ms mark. The actual transfer is shortly after that. The second handshake begins immediately and two lines do in fact follow the sequence, but only one has its line low when a second clock pulse 'confirms'the exchange. Finally, a TWI command is issued at about the -10ms mark to complete that last SPI transmission.



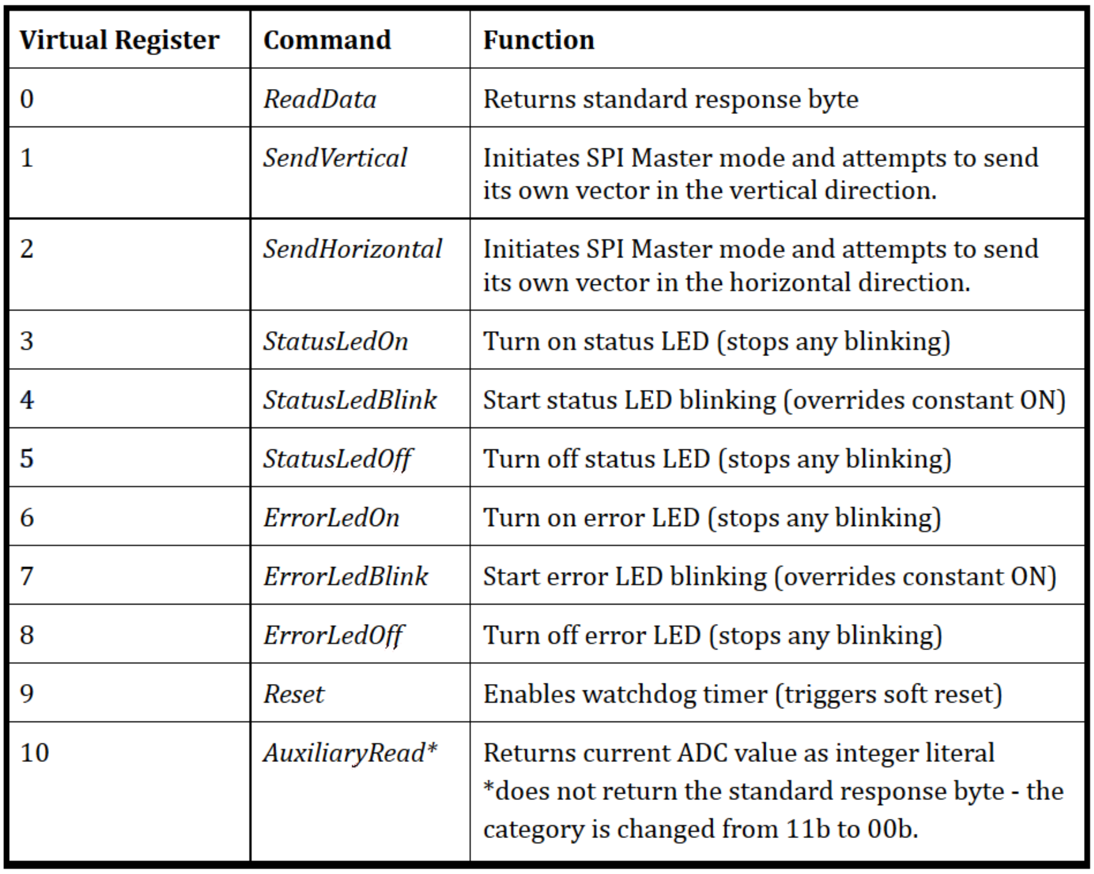
Signal	Wire ID	Wire Status	Edge A
- Data[31..0]			•
s_di	D8	H	
m_do	D12	H	
m_clk	D17	H	
SCL	D19	H	↥
s_clk	D21	H	
s_f_below	D24	L	
m_t_right	D25	L	
m_t_above	D27	L	
SDA	D29	H	
s_f_left	D31	L	

Fig 7

The data transferred is an encoded vector indicated the sending-block's coordinates in the xy-plane of the user-program's topology: the first 3 bits are the x-coordinate, and the next 4 bits are the y-coordinate. The receiving block will add 1 to the x- or y-coordinate it received, depending on from which direction it was sent. If an overflow will occur (e.g. if the block receives a x-coordinate of 7 from the left, it would increment to 8 which would overflow beyond the allotted 3 bits), then the process is aborted and an error indicating LED will blink to indicate the error condition. Otherwise, this encoded and incremented vector becomes the address of this block for a Two-Wire Interface global bus.

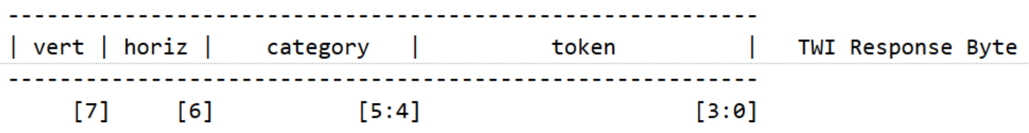
5.1.4 TWI Commands

The TWI interface allows targeted commands to trigger a handful of actions. The BeagleBone Black issues a TWI read request indicating an address (block) and a register (command). These numbers act as virtual registers on the block and are captured to issue tasks in a work queue. Below is a table of all the implemented actions.



Virtual Register	**Command**	**Function**
0	*ReadData*	Returns standard response byte
1	*SendVertical*	Initiates SPI Master mode and attempts to send its own vector in the vertical direction.
2	*SendHorizontal*	Initiates SPI Master mode and attempts to send its own vector in the horizontal direction.
3	*StatusLedOn*	Turn on status LED (stops any blinking)
4	*StatusLedBlink*	Start status LED blinking (overrides constant ON)
5	*StatusLedOff*	Turn off status LED (stops any blinking)
6	*ErrorLedOn*	Turn on error LED (stops any blinking)
7	*ErrorLedBlink*	Start error LED blinking (overrides constant ON)
8	*ErrorLedOff*	Turn off error LED (stops any blinking)
9	*Reset*	Enables watchdog timer (triggers soft reset)
10	*AuxiliaryRead**	Returns current ADC value as integer literal *does not return the standard response byte - the category is changed from 11b to 00b.

Each command (except for number 10) returns a response byte that shows the current status of the addressed block. This byte is described below:



```
------------------------------------------------------------
| vert | horiz |    category    |          token           |   TWI Response Byte
------------------------------------------------------------
    [7]     [6]           [5:4]                        [3:0]
```

The most significant two bits are flags indicating if the current block has adjacent blocks connected in the vertical and horizontal directions respectively. The next two bits identify the category (or wheel) that is used for this block. And finally, the remaining four bits are the most significant 4 bits from the ADC read from the potentiometer. This indicates the user-selected token and, combined with the category identifier, indicates the exact token to be used in the parser.

5.2 Processor Board

5.2.1 Lexer

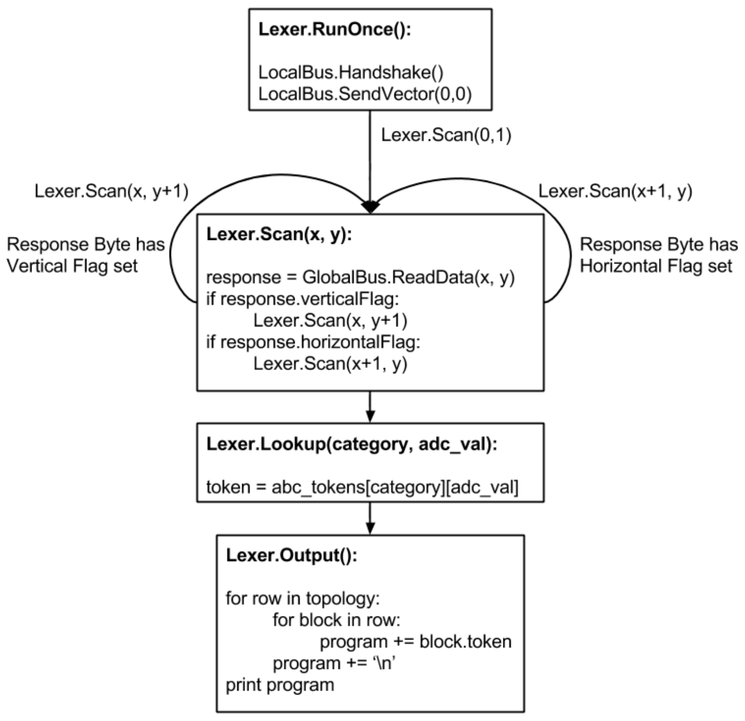


Fig 8

The lexer on the BeagleBone Black utilizes both the local SPI-based bus as well as the global TWI-based bus. The BeagleBone Black initiates a standard handshake, as described above, and sends its own coordinate of (0, 0) to the connected block. This starts a chain of sending vectors to the 'end of the line.'The lexer then uses the TWI to query the first block with a ReadData command, and recursively traverses the topology according to the vertical and horizontal adjacency flags in the response byte.

For each block response it receives with the vertical flag set, it queries the next vertical address until it receives a response without that flag set. It then sends a SendHorizontal command to the next block with a x-coordinate of zero and a horizontal adjacency flag set. The lexer queries that row until it gets to the end and then sends the next SendHorizontal command as necessary. This collects all the blocks (including their categories and ADC values). Each block's ADC value and category is used as keys to a master token lookup table, shown in the table on the following page. The lexer then prints a string constructed from the mapped tokens to stdout which is captured by the parser / interpreter.

ADC Value (4 MSBs)	Values	Operators	Controls	Statements
0	0	+	while	output 1
1	1	+	while	output 1
2	2	-	while	output 1
3	3	/	endwhile	output 2
4	4	*	endwhile	output 2
5	5	^	if	output 3
6	6	^	if	output 3
7	7	%	else	output 3
8	8	=	else	output 4
9	9	!=	endif	output 4
10	.	>	edif	output 5
11	X	>	(	output 5
12	Y	<	(	output 5
13	Z	>=	)	print
14	sum	<=	)	print
15	count	!	)	print

5.2.2 Parser

The parser is implemented using GNU Flex and Bison. Bison is used to allow rapid changes to grammar rules once physical manipulators are tested to give feedback to drive iterative redesign of the language. Flex is used to tokenize literals from the input stream and ease case-insensitivity of the language's keywords.

Tokens

Tokens patterns are expressed as regular expressions and are available in the appendix. Non-tokenized patterns that are in the source code are omitted for brevity. These patterns are used for ignoring certain patterns or implementing parser control patterns.

Grammar

The Bison based parser builds an abstract syntax tree whose structure can be modeled with a 1-to-1 mapping with the grammar rules. The grammar rules are provided in the appendix.

5.2.3 Interpreter

The interpreter receives the root node from the parser, which is a STATEMENT_LIST and descends it directly. The interpreter follows each node in the list and evaluates each node, discarding any return values and performing each node's side effects.

Each operand in the statement in the statement list is recursively evaluated as expressions, until a terminator is reached. Terminators are type NUM and VAR.

5.2.4 Output Pipeline

Outputs from the interpreter are put on stdout in the form:

```
#OUTPUT(CHANNEL, VAL)
```

The orchestration script pipes the output of the interpreter through a pipeline of output modules.

The output modules are written in Python and read from their stdin. They echo everything they read from their stdin to their stdout except for their corresponding #OUTPUT control codes. When the output module reads an output control code, it performs an action.

The following outputs modules were written. Of the output modules written, the LCD Text module and LCD RGB Backlight module were demonstrated.

LCD Text

An LCD attached to the control board is updated with the value given to the corresponding output block when the output statement is executed.

LCD RGB Backlight

The LCD attached to the control board has an RBG LED used as a backlight which is updated with the value given to the corresponding output block when the output statement is executed. The modulo 7 of the value given is truncated to an integer which indexes a color in the following list: [red, orange, yellow, green, blue, indigo, violet].

Piezo Buzzer

A piezo buzzer attached to the control board is buzzed a number of times corresponding to the value given to the output block when the output statement is executed. The number of beeps is ceiling 5.

Relay

A relay attached to the control board is closed for a number of time units corresponding to the value given to the output block when the output statement is executed. A time unit is about a half of a second and is ceiling 5.

Stdout

The value given to the output block when the output statement is executed is printed to stdout in an informative way.

6 Experiments and Testing

Testing of the equipment is broken down into two sections. The first section is hardware tests related to the construction of the blocks and cape. For our hardware test plans we implemented four different test plans. The first hardware test is the connectivity of an individual block. After this is complete the second hardware test is connecting a set of blocks together without connecting power. Connectivity is checked across the global busses to ensure that end to end connections exist throughout the block configuration. The third and fourth hardware tests are connectivity checks on the programmer header and output breakout cape.

The second set of tests were to ensure that the software was loaded correctly and functioning as intended. Software test number one is a simple power on test of an individual block to ensure that it goes to the awaiting address state while not connected to the controller. Following this test a block is connected individually to the controller board to verify that it receives its address and indicates this on the status LED. Once these two tests were complete the final software test is connecting a syntactically correct line of code to ensure the controller and outputs are functioning as intended.

The detailed test plans are attached at the end of the document.

7 Results

52 blocks, 3 programming headers, and 2 capes were made. These can control 4 different outputs with a Turing-complete grammar. Multiple functions are available on each block using 4 different selector wheels. A complete block set includes: 30 blocks with selectable functions, a Controller Cape for BeagleBone Black which supplies power and reset line to networked blocks as well as provides 4 default output components, and a programmer breakout for block updates. It is an open source project, both hardware and software, so all relevant documents, designs, and source code is available in a repository for the public to fork and merge. The project requirements were all met: *see Appendix I for Project Requirements Checklist*

8 Conclusions

8.1 Lessons Learned

Throughout the project there were many areas that could be improved, but two of the design methods stood out when reviewing the project. The first of these is the tasking of multiple individuals or teams to develop ideas in parallel. Starting from the concept generation phase, ideas were developed and evaluated by one individual with only some discussion at group meetings, and often older ideas would be reevaluated with each potential roadblock that appeared. This showed that our group would have benefited from developing multiple ideas until it was clear one would be used in the final revision. The second design method that was apparent after many of the revisions was that we repeatedly expanded our hardware capabilities to meet new needs. It would have been more efficient for the design process had a more complex set of equipment been selected that allowed the design to be finalized, and at that point trim down the hardware to only contain features required.

In addition to improving the team's design process, our project would have run more smoothly if we had allocated equipment earlier rather than later. Testing equipment was requested after discovering it was required, and some time was lost that could have been saved with some planning.

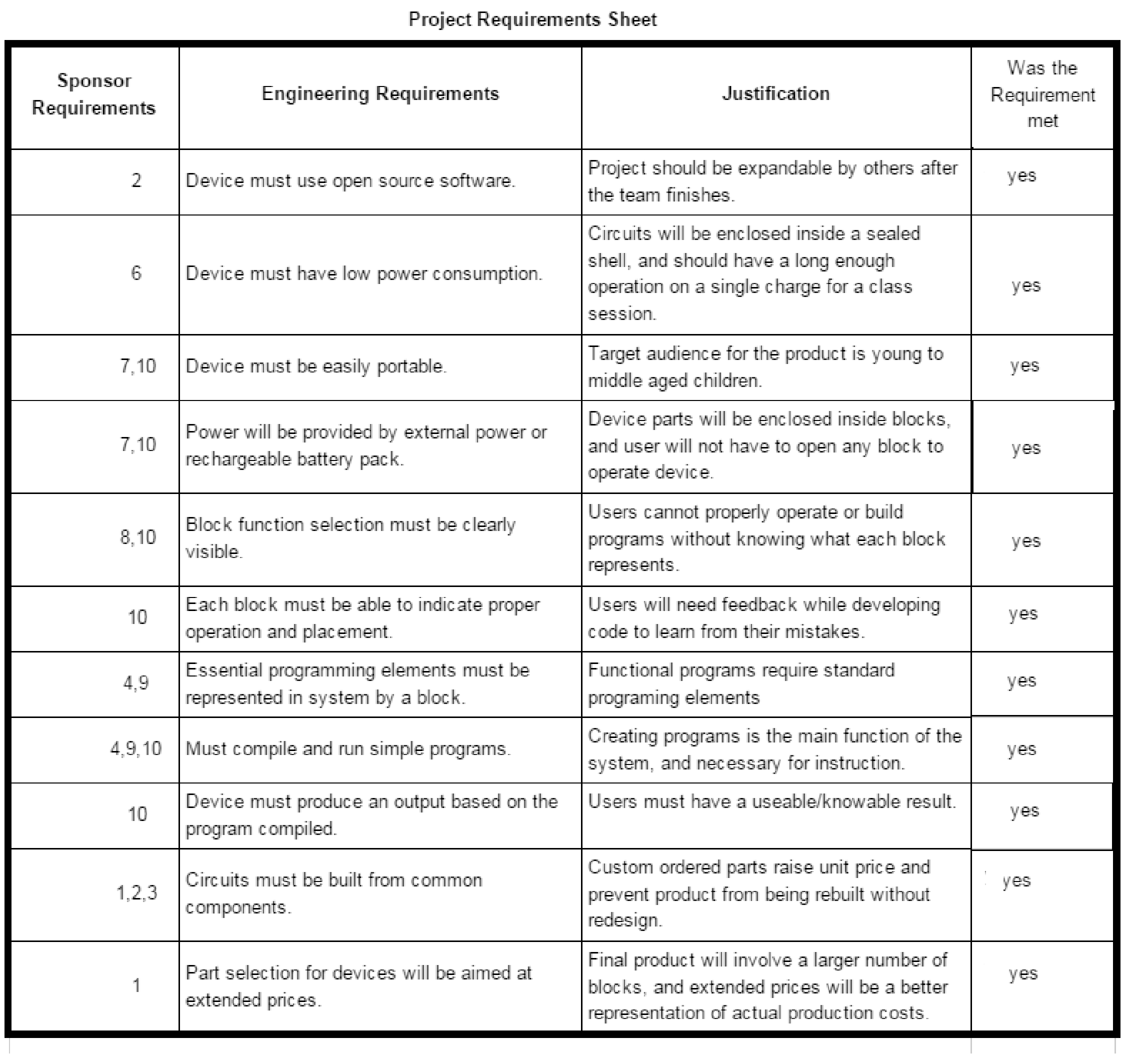
8.2 Future Work

Based on the original project goals and initial concept generation, we propose several additions and refinements that may help the project serve its purpose. The first is reducing the cost-per-block. As the intention is to provide a tool for classroom use, keeping the cost-per-set low will make the project more economically accessible. The largest expense per block is the four connectors and the PCB itself. For reducing connector cost a pin increase to fifteen combined with an increased volume per order will reduce the cost significantly due to the fifteen pin version of the DB connector being a much more common part. The size of the wheels and blocks are limited by the size of the circuit board inside currently, so reducing circuit board size is not an easy modification, but changing the number of boards ordered per batch without a rush on the shipment will lower this expense as well.

The second area of improvement for the blocks is the physical strength of each block of code. During our testing of the BlockBoard v2.2 circuit the stress on one of our connectors broke the solder joint on a support post. While this was partly due to the stiffness of the brand new connectors, we took additional steps to add reinforcement to the blocks before the demonstration. To reduce stress on the connector posts and make the blocks themselves more rigid an expanding glue was added on the seam between the connector and the printed circuit board. To prevent this in future versions a right angle DB connector that more closely mates to the circuit board could be used, as with our DB male connectors a slight gap existed between the housing and the circuit board that allowed the connector to rock back and forth when the connection was made.

The third future goal is to improve the functionality of the block sets. The first method of expanding the set's capabilities is to introduce input blocks to accompany the function select wheel. These blocks could quantize environmental characteristics such as ambient light and temperature to demonstrate programming interaction with the physical world for a more engaging learning experience. The second method of expanding the block set would be to allow for user defined functions. The method of implementation would be to add a user(#) function to the output wheel, and instructing the controller to assign the current configuration of code to that function when it is received. This would allow programs that are not constrained to a limited number of blocks or wheel types, but rather to the memory of the controller.

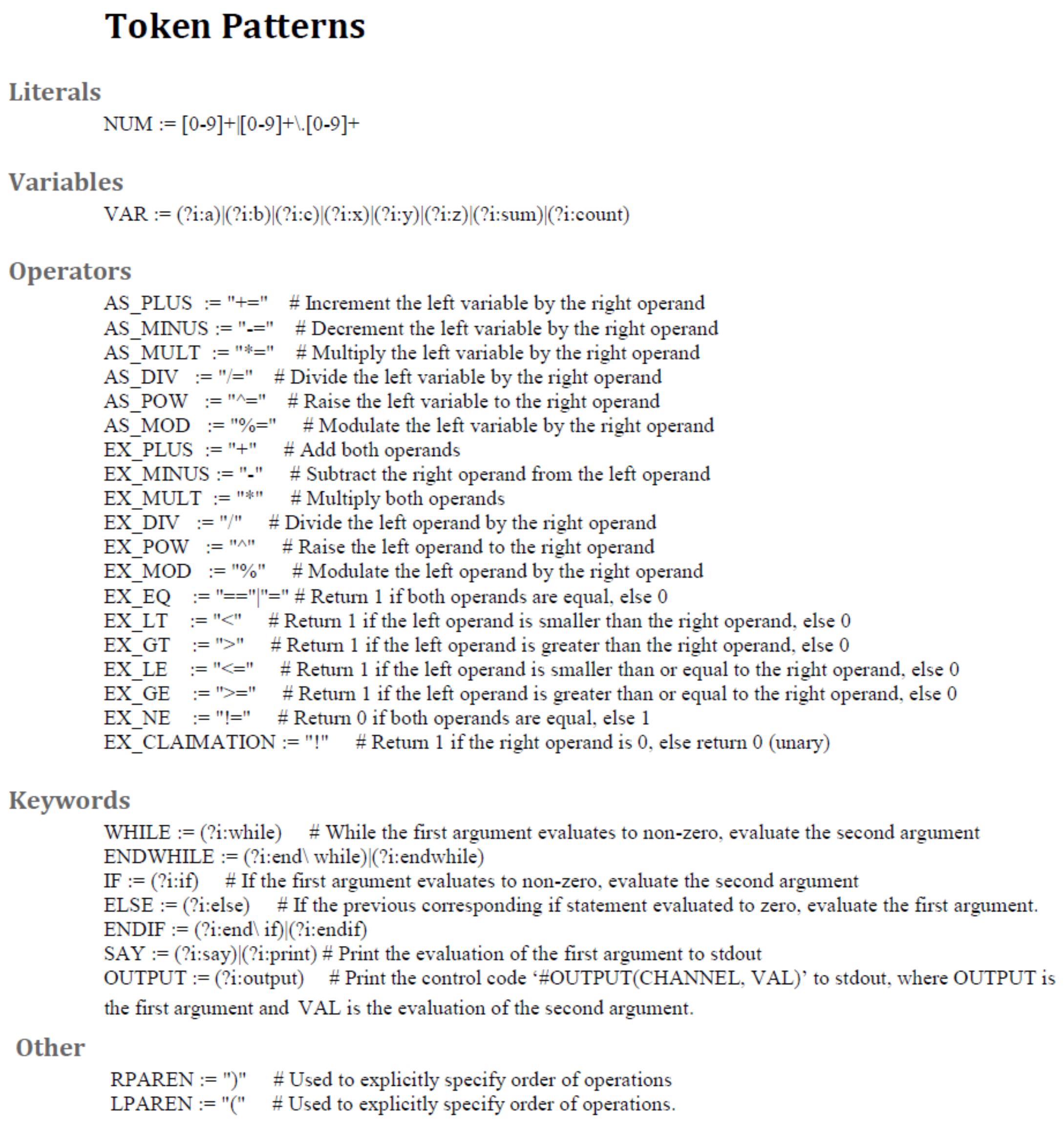
9 Appendix I: Project Checklist



Project Requirements Sheet

Sponsor Requirements	Engineering Requirements	Justification	Was the Requirement met
2	Device must use open source software.	Project should be expandable by others after the team finishes.	yes
6	Device must have low power consumption.	Circuits will be enclosed inside a sealed shell, and should have a long enough operation on a single charge for a class session.	yes
7,10	Device must be easily portable.	Target audience for the product is young to middle aged children.	yes
7,10	Power will be provided by external power or rechargeable battery pack.	Device parts will be enclosed inside blocks, and user will not have to open any block to operate device.	yes
8,10	Block function selection must be clearly visible.	Users cannot properly operate or build programs without knowing what each block represents.	yes
10	Each block must be able to indicate proper operation and placement.	Users will need feedback while developing code to learn from their mistakes.	yes
4,9	Essential programming elements must be represented in system by a block.	Functional programs require standard programing elements	yes
4,9,10	Must compile and run simple programs.	Creating programs is the main function of the system, and necessary for instruction.	yes
10	Device must produce an output based on the program compiled.	Users must have a useable/knowable result.	yes
1,2,3	Circuits must be built from common components.	Custom ordered parts raise unit price and prevent product from being rebuilt without redesign.	yes
1	Part selection for devices will be aimed at extended prices.	Final product will involve a larger number of blocks, and extended prices will be a better representation of actual production costs.	yes

10 Appendix II: Token Patterns



Token Patterns

Literals

NUM := [0-9]+|[0-9]+\.[0-9]+

Variables

VAR := (?i:a)|(?i:b)|(?i:c)|(?i:x)|(?i:y)|(?i:z)|(?i:sum)|(?i:count)

Operators

```
AS_PLUS  := "+="   # Increment the left variable by the right operand
AS_MINUS := "-="   # Decrement the left variable by the right operand
AS_MULT  := "*="   # Multiply the left variable by the right operand
AS_DIV   := "/="   # Divide the left variable by the right operand
AS_POW   := "^="   # Raise the left variable to the right operand
AS_MOD   := "%="   # Modulate the left variable by the right operand
EX_PLUS  := "+"    # Add both operands
EX_MINUS := "-"    # Subtract the right operand from the left operand
EX_MULT  := "*"    # Multiply both operands
EX_DIV   := "/"    # Divide the left operand by the right operand
EX_POW   := "^"    # Raise the left operand to the right operand
EX_MOD   := "%"    # Modulate the left operand by the right operand
EX_EQ    := "=="|"=" # Return 1 if both operands are equal, else 0
EX_LT    := "<"    # Return 1 if the left operand is smaller than the right operand, else 0
EX_GT    := ">"    # Return 1 if the left operand is greater than the right operand, else 0
EX_LE    := "<="   # Return 1 if the left operand is smaller than or equal to the right operand, else 0
EX_GE    := ">="   # Return 1 if the left operand is greater than or equal to the right operand, else 0
EX_NE    := "!="   # Return 0 if both operands are equal, else 1
EX_CLAIMATION := "!"   # Return 1 if the right operand is 0, else return 0 (unary)
```

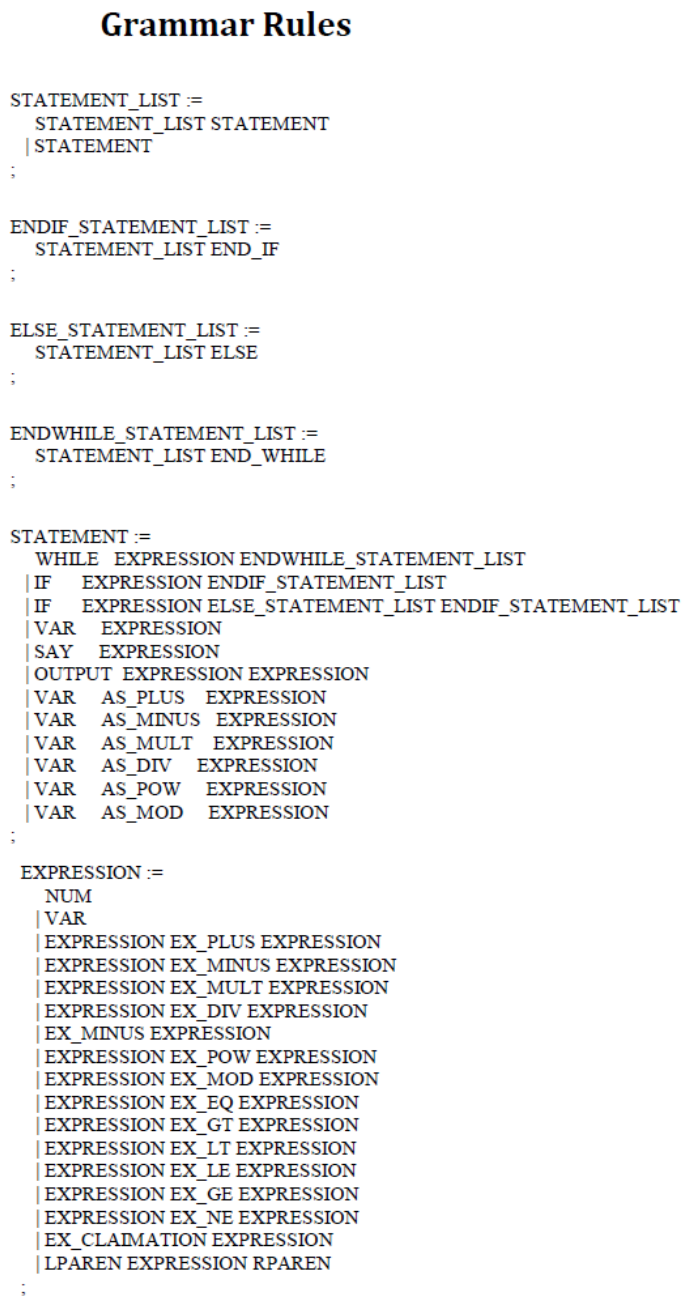
Keywords

```
WHILE := (?i:while)    # While the first argument evaluates to non-zero, evaluate the second argument
ENDWHILE := (?i:end\ while)|(?i:endwhile)
IF := (?i:if)    # If the first argument evaluates to non-zero, evaluate the second argument
ELSE := (?i:else)    # If the previous corresponding if statement evaluated to zero, evaluate the first argument.
ENDIF := (?i:end\ if)|(?i:endif)
SAY := (?i:say)|(?i:print) # Print the evaluation of the first argument to stdout
OUTPUT := (?i:output)    # Print the control code '#OUTPUT(CHANNEL, VAL)' to stdout, where OUTPUT is
the first argument and VAL is the evaluation of the second argument.
```

Other

```
RPAREN := ")"    # Used to explicitly specify order of operations
LPAREN := "("    # Used to explicitly specify order of operations.
```

11 Appendix III Grammar Rules



Grammar Rules

```
STATEMENT_LIST :=
    STATEMENT_LIST STATEMENT
  | STATEMENT
;

ENDIF_STATEMENT_LIST :=
    STATEMENT_LIST END_IF
;

ELSE_STATEMENT_LIST :=
    STATEMENT_LIST ELSE
;

ENDWHILE_STATEMENT_LIST :=
    STATEMENT_LIST END_WHILE
;

STATEMENT :=
    WHILE   EXPRESSION ENDWHILE_STATEMENT_LIST
  | IF      EXPRESSION ENDIF_STATEMENT_LIST
  | IF      EXPRESSION ELSE_STATEMENT_LIST ENDIF_STATEMENT_LIST
  | VAR     EXPRESSION
  | SAY     EXPRESSION
  | OUTPUT  EXPRESSION EXPRESSION
  | VAR     AS_PLUS    EXPRESSION
  | VAR     AS_MINUS   EXPRESSION
  | VAR     AS_MULT    EXPRESSION
  | VAR     AS_DIV     EXPRESSION
  | VAR     AS_POW     EXPRESSION
  | VAR     AS_MOD     EXPRESSION
;

 EXPRESSION :=
     NUM
   | VAR
   | EXPRESSION EX_PLUS EXPRESSION
   | EXPRESSION EX_MINUS EXPRESSION
   | EXPRESSION EX_MULT EXPRESSION
   | EXPRESSION EX_DIV EXPRESSION
   | EX_MINUS EXPRESSION
   | EXPRESSION EX_POW EXPRESSION
   | EXPRESSION EX_MOD EXPRESSION
   | EXPRESSION EX_EQ EXPRESSION
   | EXPRESSION EX_GT EXPRESSION
   | EXPRESSION EX_LT EXPRESSION
   | EXPRESSION EX_LE EXPRESSION
   | EXPRESSION EX_GE EXPRESSION
   | EXPRESSION EX_NE EXPRESSION
   | EX_CLAIMATION EXPRESSION
   | LPAREN EXPRESSION RPAREN
 ;
```

12 Appendix IV: Test Plans

Test Writer: Daniel Frister			
Test Case Name:	Block of Code Connector Check	**Test ID:**	ABC-Hardware-01
Description:	Verify electrical connectivity of all processor pins and connectors.	**Type:**	Black box
Tester Information:			
Name of Tester:	Daniel Frister and Greg Stromire	**Date:**	5/28/2015
Harware Ver:	BlockBoard v2.2	**Time:**	2:00 PM
Setup:	A fully populated individual Block with no external connections. A digital multimeter will be required.		

Ste	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Measure from processor pin 1 to each connector pin 9.	Electrical connection present.	Pass			
2	Measure from processor Pin 2 to left connector pin 3.	Electrical connection present.	Pass			
3	Measure from processor pin 3 to each connector pin 8.	Electrical connection present.	Pass			
4	Measure from processor pin 4 to bottom connector pin 3.	Electrical connection present.	Pass			
5	Measure from processor pin 5 to each connector pin 7.	Electrical connection present.	Pass			

Test Writer: Daniel Frister						
Test Case Name:		Block of Code Global Bus Check			**Test ID:**	ABC-Hardware-02
Description:		Verify global bus connectivity when in operating configuration.			**Type:**	Black box
Tester Information:						
Name of Tester:		Daniel Frister			**Date:**	5/29/2015
Harware Ver:		BlockBoard v2.2			**Time:**	4:00 PM
Setup:		A fully populated set of block boards. A digital multimeter will be required.				
Ste	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Connect all boards using top and bottom connectors.	Connectors should seat completely.	Pass			
2	Measure between bottom connector pin 1 on bottom board and top connector pin 1 on top board.	Electrical connection present.	Pass			
3	Measure between bottom connector pin 6 on bottom board and top connector pin 6 on top board.	Electrical connection present.	Pass			
4	Measure between bottom connector pin 7 on bottom board and top connector pin 7 on top board.	Electrical connection present.	Pass			

Test Writer: Daniel Frister				
Test Case Name:	Programmer Breakout Connectivity	**Test ID:**	ABC-Hardware-03
Description:	Verify connectivity of external programmer breakout.	**Type:**	Black box
Tester Information:			
Name of Tester:	Daniel Frister	**Date:**	5/22/2015
Harware Ver:	BlockBoard v2.2	**Time:**	3:00 PM
Setup:	A fully populated programmer breakout. A digital multimeter will be required.		

Ste	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Measure between header pin 1 and connector pin 3.	Electrical connection present.	Pass			
2	Measure between header pin 2 and connector pin 7.	Electrical connection present.	Pass			
3	Measure between header pin 3 and connector pin 8.	Electrical connection present.	Pass			
4	Measure between header pin 4 and connector pin 9.	Electrical connection present.	Pass			

Test Writer: Daniel Frister					
Test Case Name:	Block of Code Power On Check			**Test ID:**	ABC-Software-01
Description:	Verify individual block power on to idle state.			**Type:**	Black box
Tester Information:					
Name of Tester:	Daniel Frister			**Date:**	6/3/2015
Harware Ver:	BlockBoard v2.2			**Time:**	11:00 AM
Setup:	A fully populated individual Block with no external connections. A DC power supply will be required.				

Ste	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Connect 3.3Vdc to Pin 7 on bottom connector.	No result.	Pass			
2	Connect GND to Pin 7 on bottom connector.	Power LED (Red) turns on.	Pass			
3	Wait ~2 seconds.	Error LED (Yellow) blinks twice, Status LED (Green) starts blinking until power is removed.	Pass			
Overall test result			Pass			

Test Writer: Daniel Frister						
Test Case Name:		Block of Code Function Scan			**Test ID:**	ABC-Software-02
Description:		Verify a single block receives its address and transmits its function to controller.			**Type:**	Black box
Tester Information:						
Name of Tester:		Daniel Frister and Greg Stromire			**Date:**	6/3/2015
Harware Ver:		BlockBoard v2.2			**Time:**	6:00 PM
Setup:		A fully populated individual Block connected to a BeagleBone Black				
Ste	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Connect 3.3Vdc to Pin 7 on bottom connector.	No result.	Pass			
2	Connect GND to Pin 7 on bottom connector.	Power LED (Red) turns on.	Pass			
3	Wait ~2 seconds.	Error LED (Yellow) blinks twice, Status LED (Green) starts blinking until power is removed.	Pass			
4	Send vector w/ BeagleBone Black	Status LED turns solid.	Pass			
5	Send ReadData command	Retuned value matches function wheel selection	Pass			

Test Writer: Greg Stromire				
Test Case Name:	Block of Code ADC Range Check	**Test ID:**	ABC-Software-03
Description:	Poll on-block ADC while spinning different wheels to verify matching tokens to wheel values	**Type:**	Black box
Tester Information:			
Name of Tester:	Greg Stromire	**Date:**	6/1/2015
Harware Ver:	BlockBoard v2.2	**Time:**	11:00 AM
Setup:	A fully populated individual Block with connected to a BeagleBone Black running the "t_adc_range.py" test script from the "test_scripts" directory.		

Step	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Run script with '-v' flag and spin Values wheel on block from extreme to extreme	Tokens indicated on Values wheel match printed tokens on screen	Pass		Tests adc mapping and wheel allocation	
2	Run script with '-a' flag and spin Operators wheel on block from extreme to extreme	Tokens indicated on Operators wheel match printed tokens on screen	Pass		Tests adc mapping and wheel allocation	
3	Run script with '-c' flag and spin Control wheel on block from extreme to extreme	Tokens indicated on Control wheel match printed tokens on screen	Pass		Tests adc mapping and wheel allocation	
4	Run script with '-s' flag and spin Statements wheel on block from extreme to extreme	Tokens indicated on Statements wheel match printed tokens on screen	Pass		Tests adc mapping and wheel allocation	
Overall test result			Pass			

Test Writer: Greg Stromire					
Test Case Name:		Block of Code TWI Address Check		**Test ID:**	ABC-Software-04
Description:		Read data, change TWI address, repeat for addresses 0-127.		**Type:**	Black box
Tester Information:					
Name of Tester:		Greg Stromire		**Date:**	6/1/2015
Harware Ver:		BlockBoard v2.2		**Time:**	12:00 PM
Setup:		A fully populated individual Block with a known, hard-coded TWI address, connected to a BeagleBone Black running the "t_i2c_address_space.py" test script from the "test_scripts" directory. Make sure the 3 i2c port is enabled: 'echo BB-I2C1 > /sys/devices/bone_capemgr.9/slots'			

Step	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Run script with block connected to BeagleBone Black	Block should return valid result until addresses above 127.	Pass			
Overall test result			Pass			

Test Writer: Greg Stromire					
Test Case Name:	Block of Code TWI Command Check			**Test ID:**	ABC-Software-05
Description:	Confirms TWI Commands trigger on-board actions.			**Type:**	Black box
Tester Information:					
Name of Tester:	Greg Stromire			**Date:**	6/1/2015
Harware Ver:	BlockBoard v2.2			**Time:**	1:00 PM
Setup:	A fully populated individual Block with a known, hard-coded TWI address, connected to a BeagleBone Black running a Python session while importing the ABC_Global_Bus class.				

Step	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Send each supported TWI command to the block.	Block should blink correct LED, reset, or perform other actions as indicated by the command	Pass			
Overall test result			Pass			