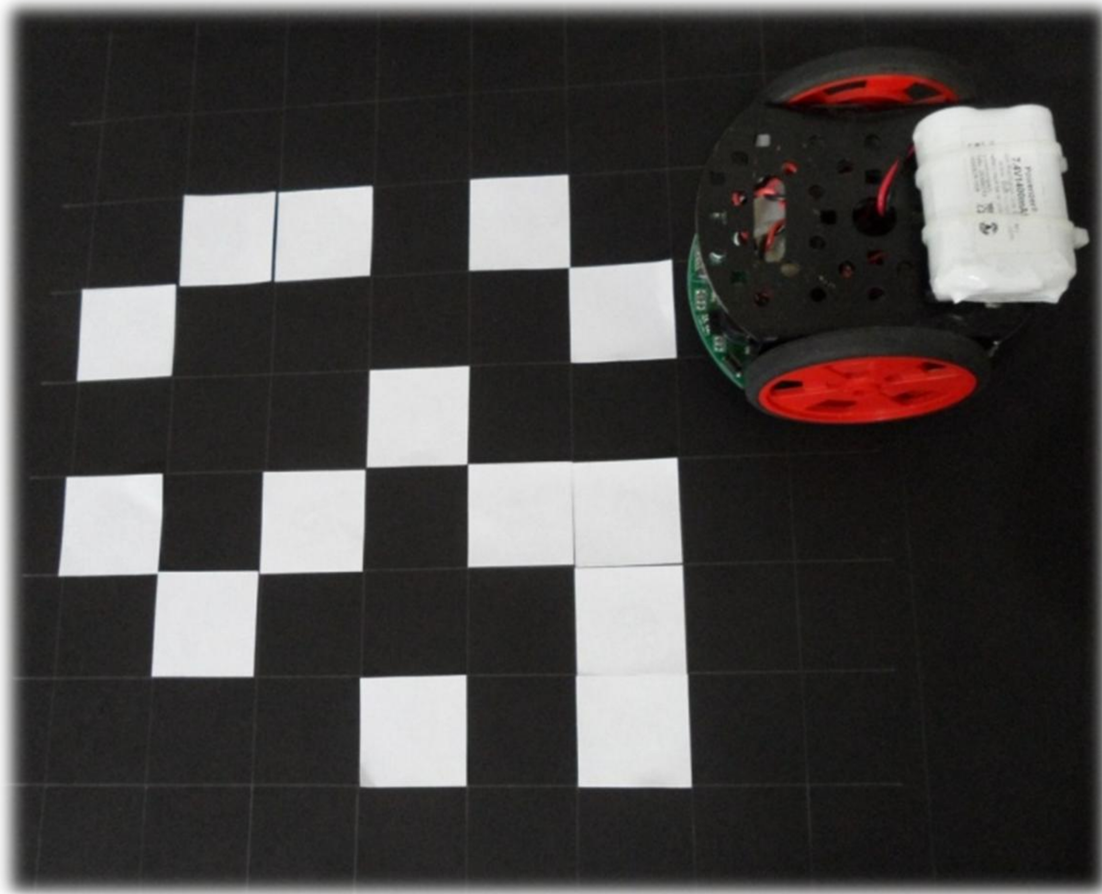
Activity Name:	The Magic Robot: Introducing Error Control Concept with Robots
Student Level:	Year 12
Satisfied Achievement Standards:	2.44 (Error Control Coding), 2.45 (Most of), 2.46 (Most of). Details inside.
Apparatus:	12Blocks programming environment, a TBot robot, black & white adhesive squares (sticky notes or fridge magnetic cards would be fine)



Contents

Summary: 3

I. Motivation: 3

II. Mind storming with students 4

III. Activity Description 5

IV. List of Areas of Achievement Standards Covered 5

V. About 12Blocks Language 6

VI. About the TBot: 6

VII. Warming UP 7

VIII. Writing the Program (With Details of the Standards mappings): 9

Phase 1: Moving the TBot across the squares: 9

Phase 2: Storing and using the data to do the magic work: 14

The Main Program Module: 21

IX. Notes 23

X. Acknowledgment 24

XI. Complete Code in a Picture 25

Summary:

By completing this activity, students should gain the knowledge and understanding of computer science concepts and programming experience required by the New Zealand Digital Technologies Achievement Standards of 2.45, 2.46, and parts of 2.44. It aims to deliver this knowledge to students through an engaging and entertaining experience.

This activity is basically another version of the popular activity 'Card Flip Magic' developed by Tim Bell and featured on csunplugged website. The interesting difference here is that we are going to have a robot performing the role of the magician. After randomly arranging the squares into a 6 by 6 matrix-like block, students would flip a particular square/card from their own choice. The robot then will walk over each square and find out the particular card that was flipped.

This activity guide assumes minimal prior knowledge of programming; hence it should be accessible to a wider range of teachers and used confidently to run this activity in their classes.

I. Motivation:

Teachers may preferably start this activity with their students by watching the video demonstration of the "Card Flip Magic" game that is featured on csunplugged.org. A good next step then would be to watch a video of this activity, which is essentially the same game except that the magician role is played by a robot. The game is expected to arouse students' curiosity in the science behind this trick, motivating them with the desire to learn more about the concept behind it. This game will let them experience a real world and hands-on example for how computers are able to perform intelligent tasks.

Teachers at this point are expected to engage in a motivating and inspiring discussion with their students and ultimately unfolding the concept of error detection and correction in computer science.

A raw and premature demonstration video of this activity can be watched on YouTube here:

http://www.youtube.com/watch?v=ivMt745iMJw [1]

[1]: a brief discussion of this video can be found at the end of this document.

II. Mind storming with students

After the students have been introduced to the concept of error control in computer science, teachers should now try to get into a discussion with them on what they think it would take to replicate the above robot game.

Students are expected to ultimately reach the consensus that they will need to solve two main distinct problems to recreate that activity. The first is to get the robot to walk over the whole matrix of squares (or blocks) passing over each individual square in order to read it. The second problem is to collect the data (whites/blacks) in a proper way in order to interpret them at the end to find out the solution (the flipped card).

There are obviously different approaches to handle each of those two problems; some will be easier than the rest. For example, to have the robot read all the squares, we could make it go over each row and then each column such that once it reaches the end of a column or row, it manoeuvres and turns back to start reading the next one. An easier approach though (and the one we chose and that appears in the demonstration video) would be to make it simply go over each row and once it reaches the end of a row, it just moves backward and manoeuvres slightly to prepare to go over the next row. Moving over each column would be redundant. While we do this when solving the game in our head, computers would not need this step since the data is already there; they only need to extrapolate it. (This in fact would be a good point to engage with students into a discussion on how computers can be more efficient than our brains at solving some spectrum of tasks). Yet another approach might be having the robot following a line beside each row such that it passes over the squares at the same time.

Similarly, the way the data is stored and interpreted can be achieved in different ways. An easy initial approach might simply be to use a separate array for each row of squares. This will probably make it easier for the students. Teachers might however want to have their students use a single two dimensional array for extra educational benefit. Unfortunately, 12Blocks currently does not support two dimensional arrays, but hopefully it will do in a future version.

Teachers do not have to adopt the same approaches that we have adopted here. After going through this guide and seeing how our approach was implemented, it should be fairly easy to adapt this guide to a different approach of, for instance, moving the robot. Whatever approach is adopted, teachers need to take into account the fact that the robot will be very sensitive to the surface type and even the slightest bump would affect its accuracy in moving on the intended path. This was the reason why we preferred to simply stick white paper squares on top of a black surface to reduce bumps and friction as much as possible. The kind of surface that the sheet is placed on would also dramatically affect both the speed and accuracy of the robot's movements.

III. Activity Description

In this activity students will be using the 12Blocks programming language to write the program that makes the robot do the magic work. While at first instant it may look to be a complicated work, 12Blocks in fact makes it quite simple and easy to control the TBot robot. Robots are always an exciting thing for kids to work with and the students will in fact be absolutely fascinated and hooked up when they see how easy it is to control the TBot with the 12Blocks language. Movements, sounds, lights, and sensor reading can all be done with basically a drag and drop of a single block of code.

This activity will be implemented in two phases. In the first phase, students will need to build the code that gets the robot moving across all the squares to read them. The other phase will be to write the code to interpret the collected data and do the necessary algorithmic work to find out the solution, i.e. the flipped card or the parity bit.

The first phase will involve a considerable time of trails and errors to get the robot to move precisely as required. This however will greatly depend on the approach chosen to implement this phase. In fact, 12Blocks should be updated soon to include a readily implemented code block for following a line *(see note [2] at the end of this document)*. So once it is available, this block can make the implementation of this phase extremely easy.

IV. List of Areas of Achievement Standards Covered

The activity will (or can) cover the following areas of the achievement standards (2.44, 2.45, and 2.46) as shown in detail below:

Area Covered	Standard	Notes
Understanding of error control coding	2.44	Achievement
Discussing how a widely used technology is enabled by error control coding	2.44	Achievement with Merit
Selecting & using appropriate data types	2.45	Achievement
Specifying variables for holding information	2.45	Achievement
Accessing & using data in indexed sequential data structures (arrays)	2.45 & 2.46	Achievement
Constructing a modular algorithmic structure where at least the top level module contains calls to other modules	2.45	A/M/E
A modular algorithmic structure where modules constitutes a well structured logical decomposition of a	2.45	Achievement with Excellence

task		
A task sufficiently rich to allow the student to meet the standard	2.46	A/M/E
Usage of expressions, iteration and selection control structures	2.46	A/M/E
Construction & calling of multiple programmer defined functions	2.46	A/M/E
Documenting the program (optional)	2.46	Achievement with Merit
Obtaining & using input data from external source	2.46	A/M/E
Testing the program with sample expected inputs	2.45 & 2.46	Achievement
Comprehensive testing of a program	2.46	Achievement with Excellence
Writing a well structured, maintainable, commented program with explanatory variable names and named modules/functions	2.46	Achievement with Excellence

Note: A/M/E stands for requirement by all three achievement levels.

V. About the 12Blocks Language

12Blocks is a visual and easy to use drop-and-drag programming language that is highly suitable for educational purposes, especially at introductory levels. It is mainly targeted at programming robots and microcontrollers and supports various types of devices. The best way to have a quick overview of this language is to visit its official website at: http://12blocks.com. The features page (http://12blocks.com/features.php) is a good place to get started. Tutorials and sample programs can also be found on the main site.

VI. About the TBot:

The TBot is a multipurpose educational robot that was designed to make learning both fun and exciting. It is suitable for a wide range of experiments and activities right from kindergartens up to university students. It has many features like motors control, full-colour LED lights, powerful sensors, encoders, sound playing, synthesizing, recording, plus many other features. Most interesting is that it can be programmed with the simple drag and drop 12Blocks language making it accessible to a wider range of users. For more details and a full list of features please visit: http://onerobot.org/overview.pdf. For orders and enquiries please visit the main official site at: http://onerobot.org

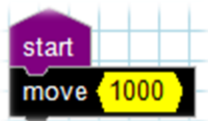
VII. Warming UP

Before students are asked to embark on writing the program, they should preferably spend some time getting familiar with the capabilities of the TBot and the 12Blocks programming language.

For example, students can initially start to try to move the robot in a straight line, forward and backward. Then, they try getting it to move in a certain path with turns and curves.

Getting the TBot to Move:

With 12Blocks, getting the TBot to move forward is as simple as this:

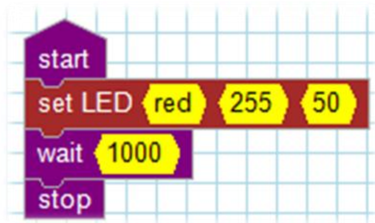


The picture below shows how to get it to move forward for half a second then turn right for 0.2 seconds, and then move backward:

As you may have guessed it, a minus sign is used to move backward. A minus sign is also used to turn left. The input cells accept time in milliseconds.

Exploring Colours with LED light

The TBot has a full-colour LED light that can be easily controlled to produce various kinds of colours and hues. The picture below shows an example of how it can be easily done in 12Blocks:

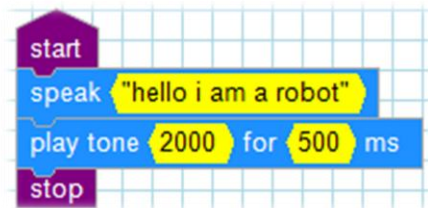


The first input cell in the '**set LED**' code block is used to select the colour name while the two other cells are used to control saturation and luminosity, respectively. A wait block is used to keep the LED emitting light for one second.

Getting the TBot to Speak:

12Blocks has various blocks to play sounds in different ways. The TBot has a built-in speaker and with 12Blocks, students can easily play wav files, sound effect files, as well as synthesized song files. They can also use the built-in microphone to implement sound controlled tasks. Other capabilities also exist like producing tones of chosen frequencies.

For example, the following program will make the TBot speak out the sentence "hello I am a robot" and then generate and play a tone of 2000 Hz for half a second:



Students should now be encouraged to go through some of the other code blocks and explore for themselves what they can do with the TBot. They should for example try using some loops, conditional blocks, assigning data to variables, printing numbers and texts to the screen, and reading user input. Tutorials and samples can be found on 12Blocks official site: http://12blocks.com.

12blocks is fairly a simple and straight forward language and very readable as well. Therefore, it would be more educationally rewarding for students if they were allowed to plunge into it without prior demonstrations. Guidance and help should be provided minimally and only when truly required. Care should be taken not to spoil the moment for the student and let them enjoy the bliss and elation of solving a problem themselves. At the same time, it is also important not to allow them to get desperate.

VIII. Writing the Program (With Details of the Standards mappings):

As mentioned earlier, this activity will be implemented in two phases. We will discuss each phase with some detail here. We will build our program with a 6 by 6 matrix of squares in mind. So, as per the game rules (explained on csunplugged), the initial matrix would be only 5 by 5. The 6th row and column would be added at the end by the game coordinator to control the parity bits.

Phase 1: Moving the TBot across the squares:

The example code blocks above showed us how easy it was to build a program in 12Blocks. In the above simple examples, we simply attached the desired block of code to the *'Start'* code block to make the program run directly once it is uploaded to the TBot (achieved by clicking the **Run** menu command). In our case here however, we are developing a relatively more complex program and hence we need to build it in a more professional approach.

We will be using functions (methods) to implement each distinct part of our program.

The functions tab of the 12Blocks programming environment has a block that can be used to define a new function. The picture below shows how it looks like. To define a new function simply drag the top angled block and rename the function with the desired name. Parameters can be declared inside the brackets, separated by commas (without space) if more than one is used.

Since we are building a function for moving the robot forward, we will not need any parameter. We can simply define our function as follows:

After defining the new function, it will appear in the functions section of the library similar to this:

```
myfunction ( ) locals:
return 0
moveForward
```

✓ **Step 1: Moving Forward**

By now students are expected to have experimented enough with moving the TBot. So an initial trial to get the robot to move over six blocks might look something similar to the following:

```
moveForward ( ) locals:
repeat n from 1 to 6 step 1
    move 217
    wait 1000
    play tone 3000 for 250 ms
```

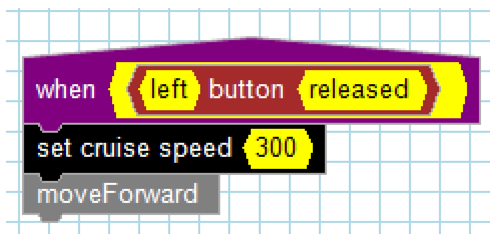
Explanation: A '*repeat*' loop block was used to have the TBot move six times for 217 milliseconds, stopping for a second between each. The number 217 was in our case determined by trial and error and students are expected to do the same to find out how much it would take the TBot to move over each single square. Obviously, this would vary according to the size of the squares or blocks used. The type of the surface of any sheet used and the type of surface that the sheet (if any) is placed on, will also significantly affect this. The speed of the TBot can also be controlled with the *'set cruise speed'* code block found in the *motion* section of the library.

A code block that will play a 'beep' tone after stopping at each square is added to give the program a nice effect of scan-like action. The TBot will actually read off the square colour at that point using its sensors, but that will be implemented later on.

A.S. Relevance[1]:	Construction of named modules **(2.45 A/M/E)**, construction of programmer defined functions/methods **(2.46 A/M/E)**, Usage of iteration control structures (**2.46 A/M/E**)

[1]: The abbreviation A.S. is used to refer to the term *Achievement Standard*.

To test the code above, students can build a simple testing program similar to this:

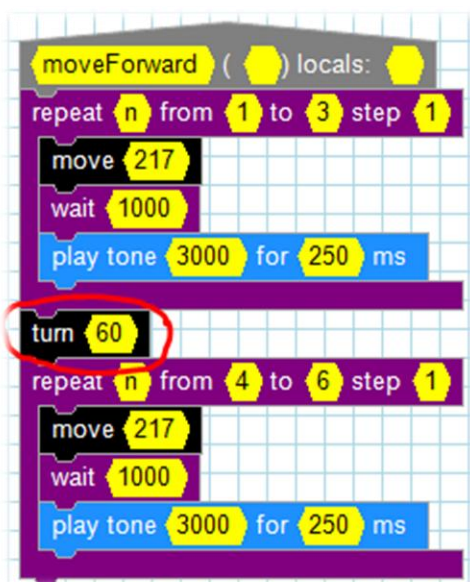


The '**button'** code block allows the user to chose from a drop menu which button (TBot has three buttons) to use and to also choose the state of the button (down, up, pressed, released) for when the action should be triggered. The button block is docked to a '**when**' block to let the program runs when the certain button is pressed and released.

A.S. Relevance:	Testing and debugging a program **(2.45 & 2.46 Achievement)**

✓ Step 2: Correcting the TBot's Skewing

Students might find that the TBot skews to one direction instead of moving in a straight line. This can happen either due to the surface or due to internal mechanics of the TBot. In case corrections were needed then students can apply them by modifying the code to something similar to this:



The trick is to experiment with how much we need to turn the TBot in the opposite direction to correct for the skewing. Students can then add this correction as needed. In our case above, we needed to add the correction after the TBot moves across 3 squares so we simply split up our loop into two and added the correction in between.

Note: The TBot unit that we have experimented with was a handmade prototype and had a tendency to skew left when moving forward. All new TBots will be machine built though and will be more precise and accurate; so students might not need to add any correction code. The new TBots will also have encoders for the wheels which should make any needed correction much easier to implement.

✓ Step 3: Moving Backward

Now that we have moved the TBot forward across a row of six squares, we need to bring it backward in order to manoeuvre it to start moving across the next row of squares. As we have discussed earlier, students can adopt a different approach of how to continue at this point. The programming needed should not be much different though than the one we will discuss here.

An easy way to implement this would be to record how much the robot has moved forward and then letting it move that much backward. We use a variable to hold this information and increment it continuously as the TBot moves forward. The picture below shows how to add this to our earlier code:

```
moveForward ( ) locals:
set x to 0
repeat n from 1 to 3 step 1
    set x to x+217
    move 217
    wait 1000
    play tone 3000 for 250 ms
```

A.S. Relevance:	Specifying variables for holding information (**2.45 Achievement**), Selecting & using appropriate data types (**2.45 Achievement**)

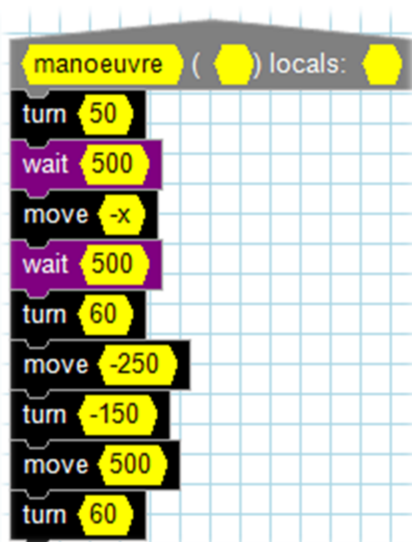
Tip: 12Blocks has one type for numbers which is simply called '*number*' and one type for strings called '*Text*'. A type of '*Value*' would accept either text or number. The '*set*' code block (shown below), for example, accepts both texts and numbers.

```
set x to 1
```

We then define another function for handling the backward movements and which will make use of the data held in the variable x. This can be achieved simply as follows:

✓ Step 4: Manoeuvring

Once the TBot is back in its original position, we simply now need to do slight manoeuvring to make it align itself with the next row of squares. This will require some experimenting with trial and error until the correct figures for how much to move and turn is determined. The below picture shows what students might reach at:



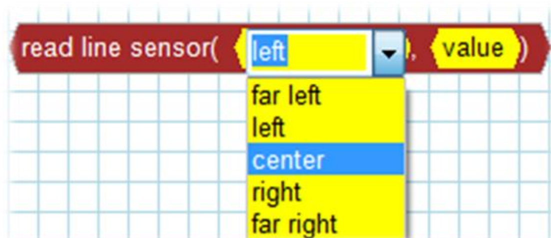
Note: the backward movement code has now been combined with the manoeuvring code and the function was renamed to simply 'manoeuvre'.

A.S. Relevance:	Construction of multiple programmer defined functions (**2.46 A/M/E**), A modular algorithmic structure where modules constitutes a well structured logical decomposition of a task (**2.45 Achievement with Excellence**)

Phase 2: Storing and using the data to do the magic work:

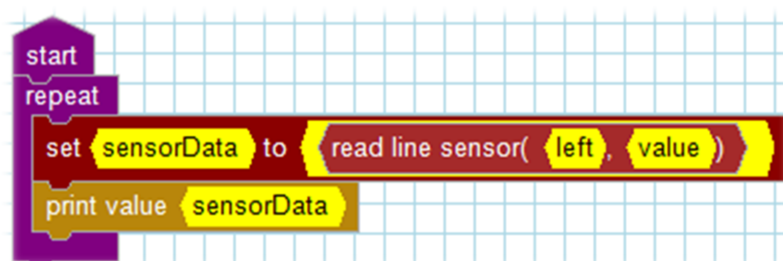
Now that the robot is able to go over each one of the squares, the next step is to build the code that will make the robot read off the colour every time it stops at each square block. The TBot has five infrared sensors in its underside facing downward. Since different colours would reflect back different amount of radiation back, we can use those sensors to find out the colour the robot is looking at.

12Blocks has a code block that could be used to easily read off sensor data. Students should be able to quickly find out the code block that should be used for this task. In the *tbot* section of the library, there is a code block named '*read line sensor*' (see below) which can be used for this purpose:



Tip: The *'read line sensor'* code block has two drop-down menus (the second shows 'value' already selected); one for selecting the specific sensor to be used and the other for selecting the type of reading desired (maximum, minimum, rate, & value).

A quick test for reading sensor data could be achieved like this:



Teacher's Tip: Students should be encouraged to work out for themselves how this code block could be used to identify white colour from black. It is a well known fact that more educational value is gained when students discover knowledge by themselves rather than when it is disclosed readily to them. With a quick test like the above, students will notice that white colour gives readings that are always above the figure 100,000. Black colour readings, on the other hand, will always be much lower. Hence, with a simple

comparison, we can easily consider any reading lower than 100,000 to be black colour and anything higher to be white.

Note: The sensor readings will differ depending on the type of surface and how shiny it is, as well as to the room's lighting. But the key idea will still be the same which is that there will be large difference between white and black colour readings that can be used to differentiate between them easily. Using the robot in sun light might not probably work correctly as there will be too much infrared noise.

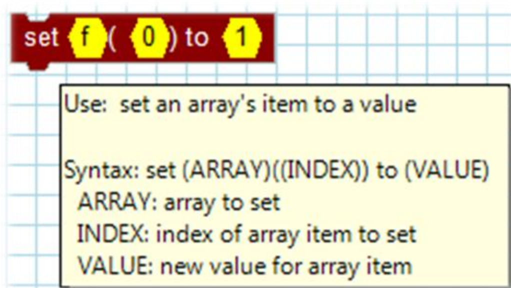
At this point, after students are comfortable with how to read sensor data, they should try to add this capability to the previous code for moving the robot such that the robot reads and stores this data as it stops at each of the squares. The following describes how this can be achieved.

✓ Step 1: Reading Off Colour Data

To make our program more organized and maintainable, we should create a user defined function that will specifically handle the sensor readings, rather than adding this directly to our previous code that handles the robot movements. The below picture shows a function named '*readColour*' that was built to read off sensor data and store them in a single array name 'dataArray':

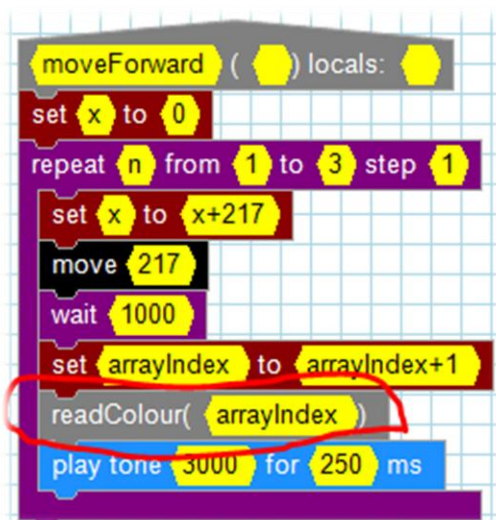
```
readColour ( index ) locals:
set s to read line sensor( center , value )
if s>100000
  set dataArray ( index ) to 1
else
  set dataArray ( index ) to 0
[+-]
```

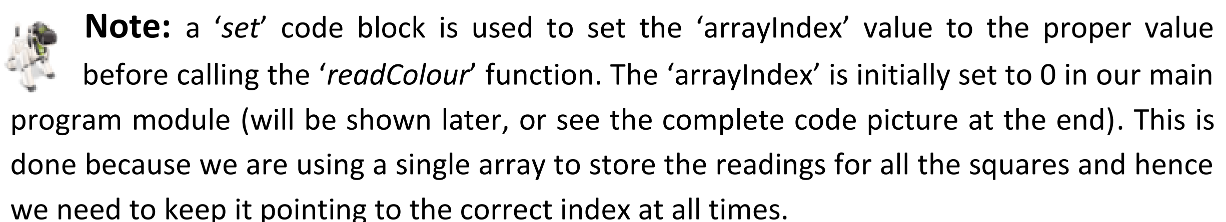
Explanation: the variable s is used to store the value of the reading. An if-statement is then used to compare that variable against the number 100,000 and store either a 0 or 1 in a specified array named 'dataArray'. A parameter named 'index' is used to allow the caller to provide the appropriate index when calling the function. 12Blocks provides a '*set*' code block (found in *vars* section of the library) that can be used to store data in an array. The picture below shows its details:



A.S. Relevance:	Obtaining & using input data from external source (**2.46 A/M/E**)

The '*readColour*' function can then be nicely called from the module program that handles the robot's movement. The picture below shows this:





Note: a '*set*' code block is used to set the 'arrayIndex' value to the proper value before calling the '*readColour*' function. The 'arrayIndex' is initially set to 0 in our main program module (will be shown later, or see the complete code picture at the end). This is done because we are using a single array to store the readings for all the squares and hence we need to keep it pointing to the correct index at all times.

A.S. Relevance:	A task sufficiently rich to allow the student to meet the standard (**2.46 A/M/E**), a modular algorithmic structure where modules constitutes a well structured logical decomposition of a task (**2.45 Achievement with Excellence**), a modular algorithmic structure with named modules where at least the top level module contains calls to other modules (**2.45 A/M/E**), accessing & using data in indexed sequential data structures (**2.45 & 2.46 Achievement**)

✓ Step 2: Storing and reading the colours data

We have mentioned earlier how this can be achieved in several approaches. However, due to current limitation of 12Blocks (not supporting multi-dimensional arrays), we decided to simply use a single array to hold the data for all the rows. You have already seen this when we built the '*readColour*' function above. This will add an overhead of having to deal with the data in a special way to find out the solution (the flipped square) later. However, this had actually made the task of storing the data easier for us (see the '*readColour*' function).

Teacher's Tip: This task is probably the hardest part of the entire program. It will require some careful thought and paper (or board) sketches and diagrams. Preferably, teachers should have a brainstorming session with their students to work out the solution together. First, students need to work out the key idea of how the flipped square can be found in the first place. After understanding this, they need to work out how we can extract the data from a single array in a manner that would allow us to apply the '*find flipped square*' rule to the data and ultimately finding the solution. The last step would be to translate that into an algorithm or pseudo code so it can be easily programmed in 12Blocks.

Teacher's Tip: This would be a very good task to ask the students to develop on paper a '*modular algorithmic structure constituting logical decomposition of the task'*, if desired (**2.45 with Excellence**).

As it is illustrated in the 'Card Flip Magic' game on csunplugged, the rule for finding the flipped square is simply to look for the row and column that have an odd number of coloured squares. The flipped square is where those row and column meet. So, for our purpose here, we will need to process the data in the array to find out that specific column and row where an odd number of coloured squares happen.

A.S. Relevance:	Understanding of error control coding (**2.44 Achievement**), discussing how a widely used technology is enabled by error control coding (**2.44 Achievement with Merit**), a modular algorithmic structure where modules constitutes a well structured logical decomposition of a task (**2.45 with Excellence**), a task sufficiently rich to allow the student to meet the standard (**2.46 A/M/E**)

We will illustrate diagrammatically how to solve the problem discussed above:

Our single array for the 6 by 6 matrix will simply look like this:

1	2	3	4	5	6	7	8	9	10	11	12	13	....	33	34	35	36

In the above diagram, the numbers inside each cell refer to the indexes. In the actual array after the TBot has passed over all squares, each cell will contain either a 0 (black) or a 1 (white). We can now re-arrange the array into a 6 by 6 matrix (as the diagram below) to make it easy for us to visualize the problem.

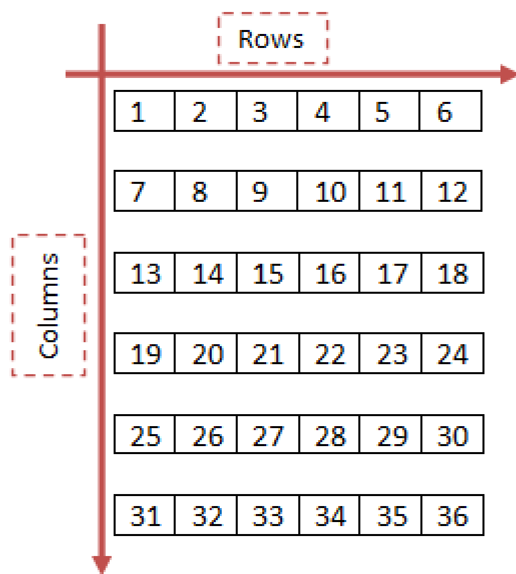


Diagram 1: Mapping the single array to the squares layout

Now, it is easy to see how our single array actually maps to the original layout of the squares which the robot has passed over. So to process the data, we will need first to go through each row to find the one containing the odd coloured squares, and then do the same for the columns.

Tip: To find the row or column with the odd number of coloured squares, we simply construct a loop that goes through each row (or column) at a time. Inside the loop, we put a counter that adds up the values (0s and 1s) of the row (or column) cells. The row (or column) we are looking for would then simply be the one that has a total with odd number.

✓ Processing the rows

Obviously, we will need a loop to process the rows, but the problem is how to deal with the messed up indexes. To construct such a loop, we will need a dynamic way to specify the start and end index for each row. It would have been easy if all rows started and ended with matching indexes, but in our case they are not. To solve this problem, note that simply by adding the number 6 to the start and end indexes of a row, we can determine the start and end indexes of the next row. This should enable us now to build a loop that processes each one of the rows dynamically. The picture below shows a user defined function for processing the rows:

```
findParityRow (  ) locals:
set counter to 0
set row to 0
set startIndex to 1
set endIndex to 6
repeat while counter%2==0
    set row to row+1
    repeat p from startIndex to endIndex step 1
        set tmp to ( get dataArray ( p ) )
        set counter to counter+tmp
    set startIndex to startIndex+6
    set endIndex to endIndex+6
print text "The Row that has a parity error is: "
print value row
```

Explanation: The inner *repeat* loop is used to calculate the total for each row by reading and adding the values in its cells. A '*get*' code block is used to read the array data and store it in a temporary variable that is in turn used to perform the addition. The *startIndex* and *endIndex* are initially set to 1 and 6 respectively. When the first run of this loop completes, the 'counter' variable will hold the total for the first row. Then, both indexes are incremented by 6 to make the loop ready for processing the next row. The outer repeat loop is used to enable us to go over all of the 6 rows. The condition '*counter%2==0*' is used to check if the total is odd or even by calculating the remainder of dividing by 2. Since we will have one and only one row with an odd number of coloured squares, we only need to move to process a next row if the total of the current row was not odd. Once we find a row with odd total then we stop processing the rows. A '*row*' variable is used to keep track at which row the outer loop has stopped, since that would be the one we are looking for.

Although the two code blocks at the end are used to print the result on the screen, a better idea is to let the TBot speaks out the result by replacing those two code blocks with something like the following:

```
speak "the row with parity error is"
spell row
```

Unfortunately, 12Blocks currently does not support variable referencing in either of the *speak* or *spell* blocks, so the [spell (row)] code block above will not work. Hopefully, this will be added soon though.

✓ Processing the columns

By looking at diagram 1 above, we can see that it is easy to process the columns by reading the cells vertically. For example, the first cell of column 1 will have index 1, the second one will have index 7, the third will have index 14, and so forth. The pattern is now clear, column 1 will have a start index of 1 and an end index of 31 but instead of stepping 1 cell in each iteration, we need to step 6 cells! The start and end indexes of the next column can also be determined by incrementing those of the previous one by 1. So now we can easily build a dynamic loop that goes over each of the 6 columns to process the data.

In a similar manner to processing the rows above, the following function will process the columns:

```
findParityColumn ( ) locals:
set counter to 0
set col to 0
set startIndex to 1
set endIndex to 31
repeat while counter%2==0
    set col to col+1
    repeat q from startIndex to endIndex step 6
        set tmp2 to get dataArray ( q )
        set counter to counter+tmp2
    set startIndex to startIndex+1
    set endIndex to endIndex+1
print text "The Column that has a parity error is: "
print value col
```

The only difference here is that we need to step 6 cells in every iteration of the inner loop and before processing a next column, we need to prepare the start and end indexes by incrementing them by 1.

A.S. Relevance:	A task sufficiently rich to allow the student to meet the standard (**2.46 A/M/E**), a modular algorithmic structure where modules constitutes a well structured logical decomposition of a task (**2.45 with Excellence**), selecting & using appropriate data types (**2.45 Achievement**), specifying variables for holding information (**2.45 Achievement**), accessing & using data in indexed sequential data structures (**2.45 &4.46 Achievement**), usage of expressions and iteration control structures (**2.46 A/M/E**)

Congratulations!

That was fairly a long step, but fortunately this completes our entire program. Now all the modules of our Magic Robot game are completed and all what is left is to call those modules (functions) properly from our main program.

The Main Program Module:

The main program is actually the simplest part of this activity. We have already built all the required algorithm structures for performing the various tasks and processes. We can simply now call them appropriately in a program similar to this:

```
when left button released
set cruise speed 300
set arrayIndex to 0
moveForward
manoeuvre
moveForward
manoeuvre
moveForward
manoeuvre
moveForward
manoeuvre
moveForward
manoeuvre
moveForward
findParityRow
findParityColumn
stop
```

Explanation: the program above is almost self-explanatory. A '*when*' control block along with a '*button*' block are used to make the program run when the left button is pressed and released. As we have indicated earlier, the 'arrayIndex' variable needs to be kept pointing at the correct index at all times since it is used to store the data in our single array. The best way to do that is to initialize it in our main module here and then increment it appropriately as it is used (see the '*moveForward*' function). A '*moveForward*' and '*manoeuvre*' functions are then called six times to make the robot go over all 6 rows. Finally, the '*findParityRow*' and '*findParityColumn*' are called to do the data processing and ultimately revealing the answer.

A.S. Relevance:	constructing and using named modules (**2.45 A/M/E**), construction & calling of multiple programmer defined functions (**2.46 A/M/E**), writing a well structured, maintainable, commented program with explanatory variable names and named modules/functions (**2.46 Achievement with Excellence**), comprehensive testing of a program (**2.46 Achievement with Excellence**), a modular algorithmic structure where modules constitutes a well structured logical decomposition of a task (**2.45 Achievement with Excellence**)

Educational Value: By completing this activity, students will have developed a well structured, organized, and maintainable program using several functions (modules) to perform the different tasks. Moreover, the final end product of this activity will allow students to experience a real world application of the error detection and correction concept and help them to better understand it. This activity could probably be a model one that nicely covers the New Zealand Digital Technologies Achievement Standards of 2.45, 2.46, and part of 2.44. Moreover, it achieves this in a new attractive and entertaining methodology that is expected to inspire students and keeps them motivated and engaged during their learning process.

IX. Notes

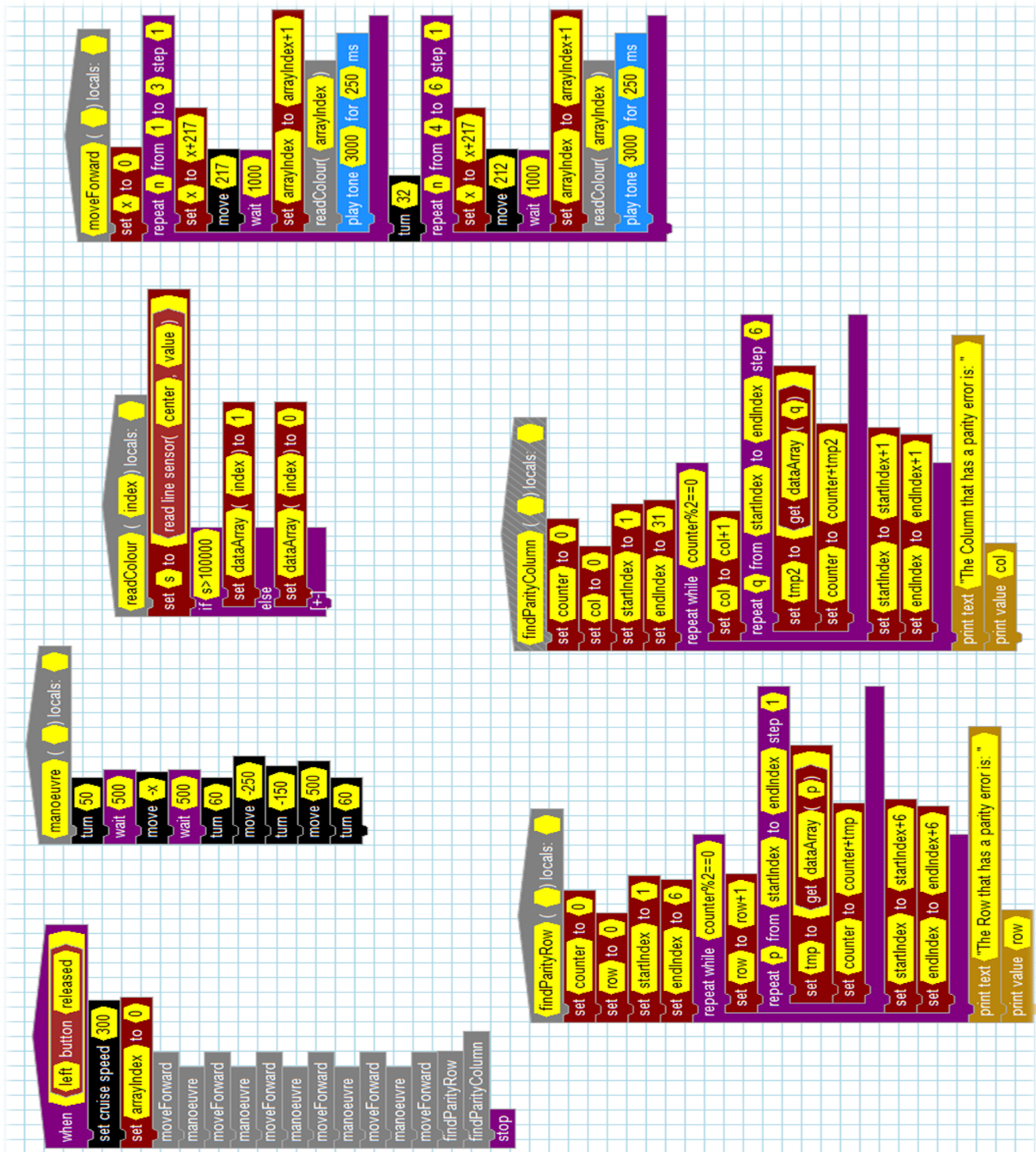
[1]: A video demonstration for the final fully working Magic Robot activity could not be created due to limited time for this project. The video given here was made at an early stage of this project and does not show the part of the robot giving the answer. Nonetheless, it should give an idea of how the overall activity might look like.

[2]: 12Blocks currently have a code block called ‘find line’ that is currently not implemented. Once implemented (in a future version), it would make it very easy to make the TBot follow a line using the sensors.

X. Acknowledgment

We would like to give our sincere thanks to Prof. Tim Bell, who has supervised this project and provided extensive help and advice while we worked on it. We also would like to give our sincere appreciation and thanks to Hanno Sander (the developer and owner of 12Blocks and the TBot) for his generous act of allowing us to use his tools free of charge and for his kind help and support, without which we could not complete this project.

XI. Complete Code in a Picture



A 12Blocks file of the complete program is included with this guide.