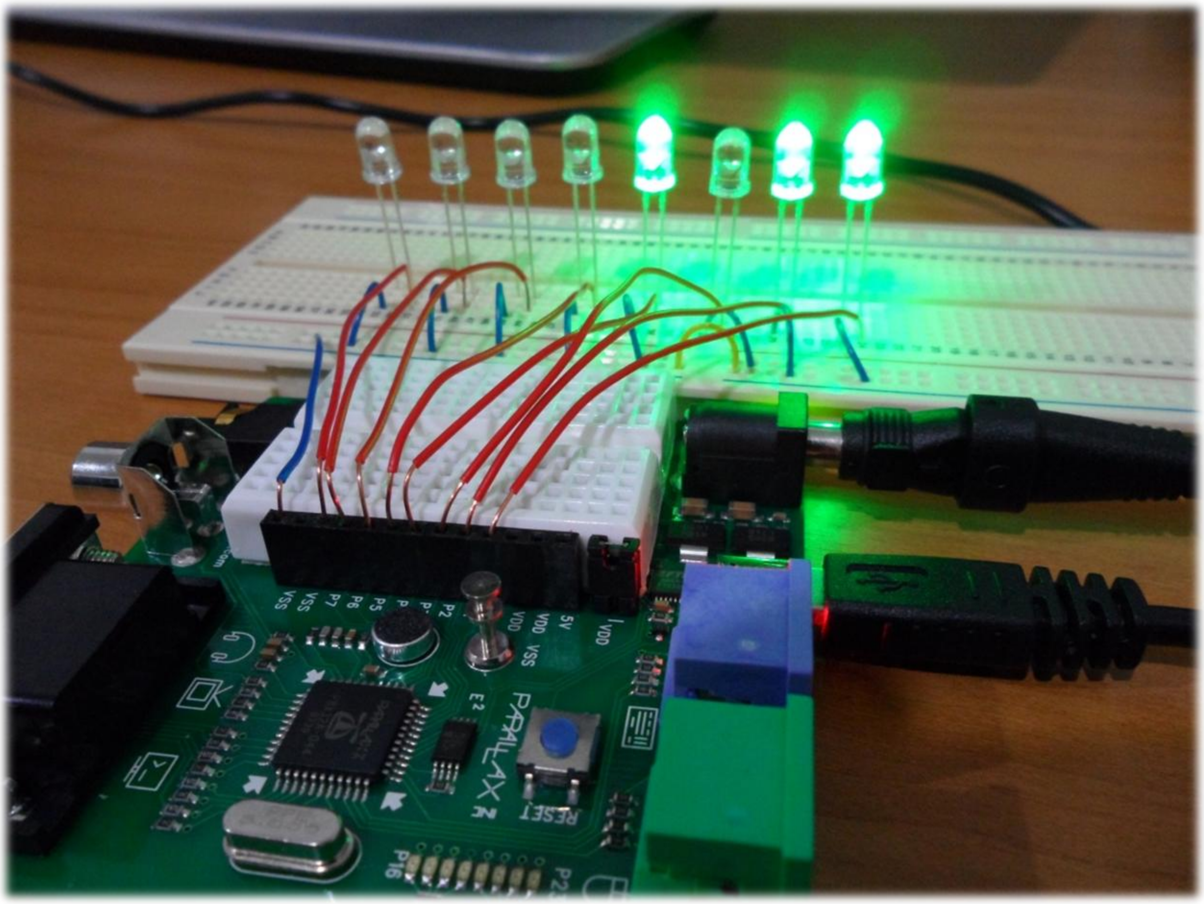
Activity Name:	Binary Number Representation in LED Lights
Student Level:	Year 12
Satisfied Achievement Standards:	2.44 (Binary Representation), 2.45 (Almost all), 2.46 (Almost all). Details inside.
Apparatus:	12Blocks programming environment, a Parallax Propeller Demo board, 8 LED lights, a breadboard, wires.



Contents

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.

It also has the added benefit of exposing students to some basic concepts in electronics. Some achievement standards of electronics may in fact be covered by this activity. While students with electronics background will be in an advantageous position, background in electronics is not necessary as this guide provides all the details required to assemble the apparatus. If desired, students could have the apparatus preassembled. This activity consists of 2 phases with each phase building up on top of the previous one.

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. Description of the Activity:

Students will be using the 12Blocks programming language to write a modular and advanced (as per 2.45 & 2.46 naming) program to control the states of LED lights. They will be constructing a function/method to convert decimal numbers into binary representation. The 8 bit binary number result will then be used to control the on/off status of a corresponding set of 8 LED lights representing the 8 bits. The end result would be having a representation of binary numbers in LED light patterns!

The decimal numbers can be fed to the function as user input, or simply used statically depending on how much of the standards teachers want to cover.

Students will need then to write another program to take user input and call the earlier function passing it user’s input value. In case static decimal values were used, a similar program also needs to be written that properly calls this function.

The final end result should be a program that asks the user for a decimal number of a specific range and then it shows the binary representation of that number in LED light pattern. In the case of static data being used, the program should simply show the same result for the pre-determined values.

The program can be extended to include validation of user input, as well as other amusing games like patterns of chasing lights (discussed later).

A demonstration video of the final product of this activity can be watched on YouTube here: http://www.youtube.com/watch?v=wAhIGoDwydk

II. 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:

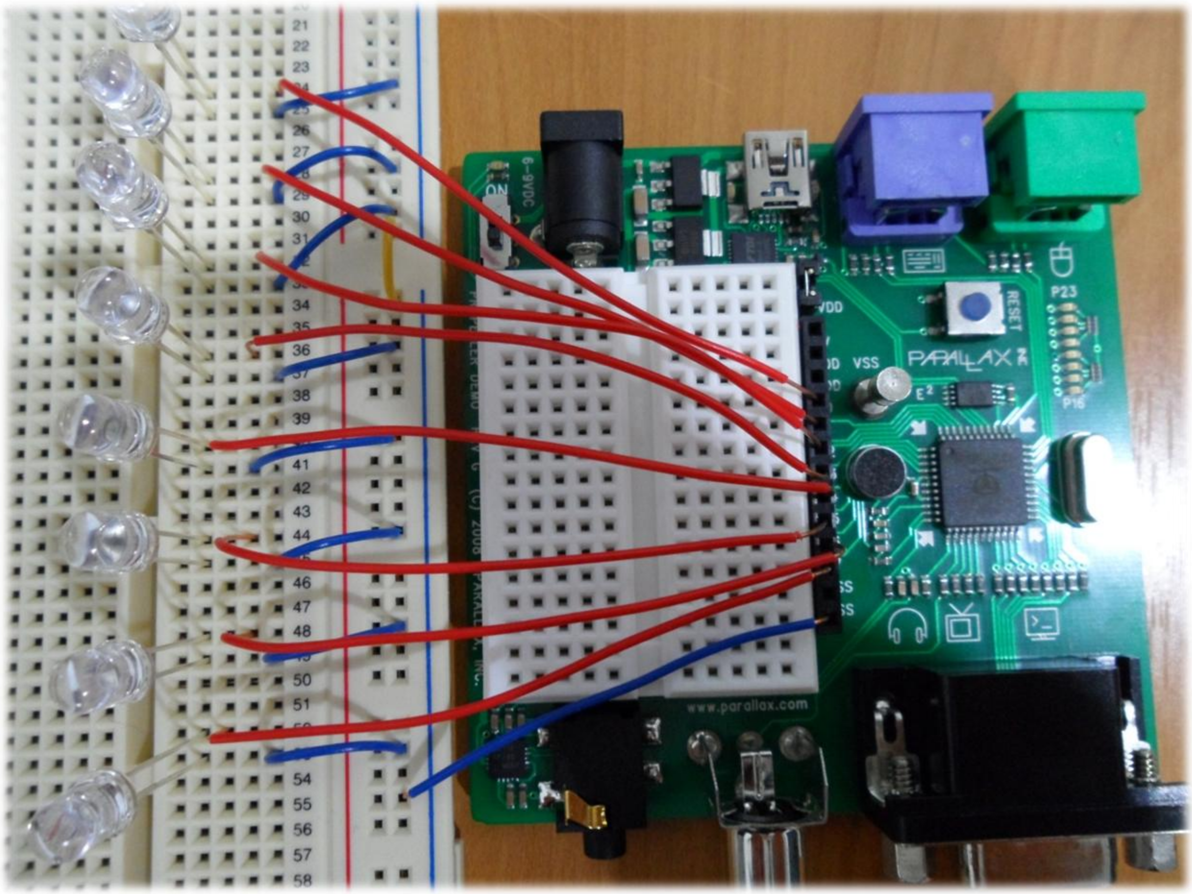
Area Covered	Standard	Notes
Understanding of binary representation of integers	2.44	A/M/E
Selecting and using appropriate data types	2.45	Achievement
Specifying variables for holding information	2.45	Achievement
Accessing and using data in indexed sequential data structures (arrays)	2.45 & 2.46	Achievement
Constructing a modular algorithmic structure	2.45	Achievement
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 user (Optional)	2.46	A/M/E
Testing with sample expected inputs	2.45 & 2.46	Achievement
Testing with expected & boundary data inputs (optional)	2.45 &2.46	Achievement with Merit (both)
Testing with expected, boundary & exceptional data inputs (optional)	2.45	Achievement with Excellence
Specifying data for a test case for the task	2.45	Achievement
Comprehensive testing (optional)	2.46	Achievement with Excellence
Well structured, maintainable, commented program with explanatory variable names and named modules (optional)	2.46	Achievement with Excellence

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

III. Assembling the Apparatus:

If students have no prior background in basic electronics then it will be a good initial step to watch the following video from Parallax Company, which covers almost all the details needed to assemble the apparatus of this activity. The video can be watched on YouTube here: http://www.youtube.com/watch?v=q_Q5s9AhCR0.

After watching the above video, the assembly should be quite easy to complete for students. The Parallax Propeller Demo Board has 8 input pins (located in the black strip beside the small breadboard) that are clearly labelled from P0 up to P7. The black strip has also two positive pins (labelled VDD) and two negative pins (labelled VSS). The positive leg (longer one) of each LED needs to be connected to one of the input pins separately. The negative legs of all the LEDs can then be connected together all to one of the VSS pins. Although in the pictures shown in this document no resistors were used when connecting the LEDs, it will be a good idea to use proper resistors to protect the LEDs from getting burned off. Since the embedded breadboard of the Propeller is quite small, students will need a bigger external breadboard to make the connections properly. The following picture clearly shows all the connections:



Additionally, a high resolution picture is also included with this guide.

★ **Acquiring the tools:**

1) The Parallax Propeller Demo board can be order from the company's official site at: http://www.parallax.com/StoreSearchResults/tabid/768/txtSearch/555-32100/List/0/SortField/4/ProductID/340/Default.aspx. It has a reference number of 555-32100 and costs around 60 US dollars.

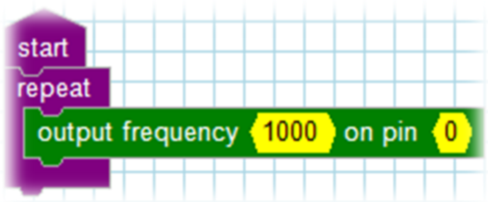
2) Wires, breadboards, and LEDs can be bought at any electronics shop. For those in Christchurch, you can try South Land Component Centre (http://www.sicom.co.nz) or JayCar Electronics (http://www.jaycar.co.nz/).

IV. About the 12Blocks Language

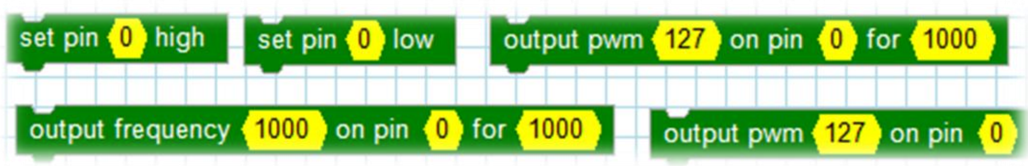
12Blocks is a visual and easy to use drop-and-drag programming language that is highly suitable for educational purposes, especially at introductory levels. It is mainly targeted at programming robots and microcontrollers and supports various types of devices. The best way to have a quick overview of 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.

V. Warming UP

Before students embark on writing the program, they should preferably spend some time getting familiar with the apparatus and the 12Blocks programming language. After getting familiar and comfortable with 12Blocks, a good start could be asking them to simply try to get one of the LED lights turned on. They need to correctly match each of the LED lights with the right pin on the Propeller board. An example code that will light up an LED connected to pin 0 could be something as simple as this:



The picture below shows some of the other code blocks that students can try:



Students should now be encouraged to go through some of the other basic code blocks and trying them out for themselves. This include trying some loops, conditional blocks, assigning data to variables, printing values and strings to the screen, and reading user inputs.

The 12blocks is fairly a simple and straight forward language and is 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.

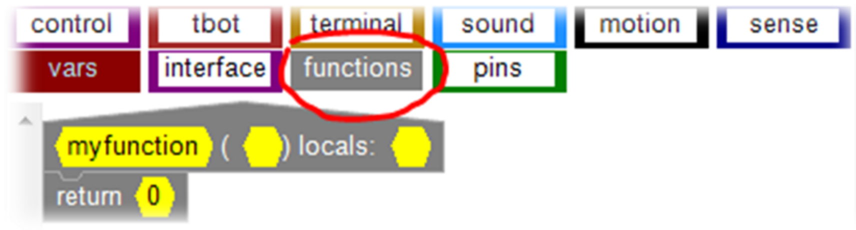
VI. Writing the Program (With Details of the Standards mappings):

As mentioned earlier, the activity should be completed in two phases. The first phase is to construct the function/method that takes an integer value as a parameter, converts it to binary, and feed the 8 binary bits to the corresponding pins in order to control the LED lights.

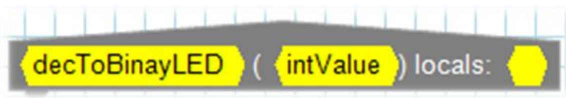
Phase 1: Writing the User Defined Function:

STEP 1: Writing the function header (interface)

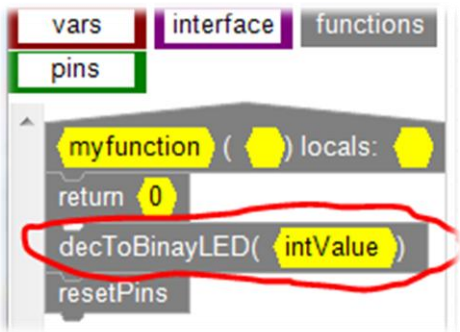
The *functions* tab in the library section of the 12Blocks programming environment has a block that can be used to define a new function. The picture below shows how it looks like. To define a new function simply drag the top angled block and rename the function with the desired name. Parameters can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



For our purpose, the function we define needs to take one parameter (the integer). Thus our function header can now look like this:



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



The new function (***decToBinaryLED***, in our case) can now simply be dragged and dropped and used just as the other code blocks to build other programs.

A.S. Relevance:	constructing named modules (**2.45 A/M/E**), programmer defined functions/methods (**2.46 A/M/E**)

STEP 2 (Optional): Comment

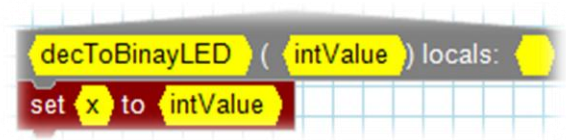
A '**comment**' block *(found in control section of library)* can be used to add description of the purpose of a particular piece of code. It is a good practice in programming to add explanatory comments for user defined functions. Below is an example in 12Blocks.

A.S. Relevance:	Documenting functions/methods (**2.46 Achievement With Merit**)

STEP 3 (Optional): Specifying a variable for holding the integer

Since we will be working on the integer value passed by the caller of our function, it is a good practice to hold a copy of the passed user's value into a local variable and use this copy in our code, leaving the original copy intact. It is not necessary to do this step as we could use the user's value directly, but this practice makes our code more maintainable. To complete this step we use the set method found in the vars section of the library.

The first cell in the '*set*' block takes the desired name of the variable while the second takes the value. Our function should now look similar to this:



A.S. Relevance:	Specifying variables for holding information (**2.45 Achievement**)

- ✓ **Tip:** It is necessary that the type of variable matches the type of the value that it will hold. In our case, the value is of type number (for simplicity, 12Blocks has one type for numbers called '*number*'). The '*set*' code block that we used above accepts both numbers and strings.

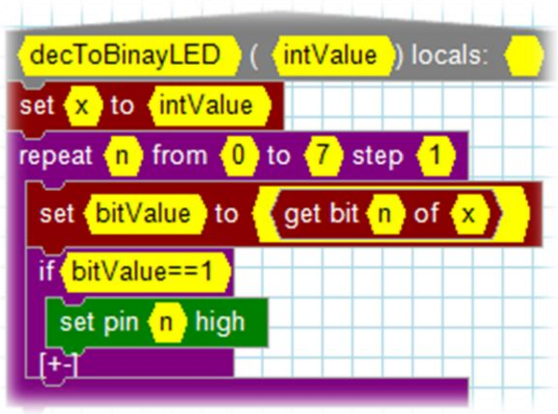
STEP 4: Converting to binary & lighting up the corresponding LED light

Now we have the value that we want to convert to binary. We need a way to find out the binary representation of this value. 12Blocks provides a block called '*get bit*' found in the *vars* section of the library.

Since we intend to represent integer numbers up to 255, this means we will need to consider 8 bits (bit 0 up to bit 7) for each value. This block retrieves a single bit at a time so we will need to use it 8 times to read the values of all 8 bits. To do this effeciently, we need to use a loop. 12Blocks has five types of loops which it call '*repeat*'. With a quick look at the *controls* section of the library, we can easily see that the most suitable one would be that shown below. This is because we can use the loop index to feed it to our get bit block to make it advance accordingly from bit 0 to bit 7.

Once we know the binary value of a certain bit, we can then use it to cotrol the status of the coressponding LED. For example, if bit 0 had a value of 1 then we need to light up the LED at position 0. We do that by controlling the signal outputted on pin 0 as we have demonstrated earlier. If bit 0 had a value of 0, then LED at position 0 needs to be off.

The picture below shows how the complete function would look like:



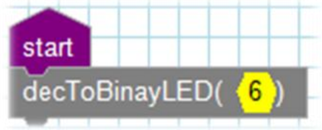
You can see that the index 'n' of the loop was fed to the '*get bit*' code block so that, for example, at loop 0, the get bit function would read bit 0... and so forth. The bit value was then saved to a variable called 'bitValue'. To control the corresponding pin, an if-statement block was used to send a High signal to the pin only if the bit value was 1. If it was 0 then we need to do nothing. The '*set pin high*' block can be found in the *pins* section of the library.

A.S. Relevance:	Understanding of binary representation of integers (**2.44 A/M/E**), Usage of expressions, iteration and selection control structures (**2.46 A/M/E**), Accessing & using data in indexed sequential data structures (**2.45 & 2.46 Achievement**), Constructing a modular algorithmic structure (**2.45 Achievement**), Well structured, maintainable, commented program with explanatory variable names and named modules (**2.46 Achievement with Excellence**)

✓ **Teacher's Tip:** The completed function shown above is obviously not the only way to achieve the required result. Teachers however need to evaluate students based on how well the function was actually implemented. For example, a student might achieve the same result without using a loop; they might use the '*get bit*' code block 8 times. This is where teachers can assess students competence. Students who simply used 8 copies of the '*get bit*' code block would probably be eligible for "Achievement" while those who used a loop would be eligible for "Achievement with Merit". This is because the standard requires writing an advanced program that is "***well structured, maintainable**, with explanatory variable names, module names, and commented functions*". Similarly, the same applies to those who used proper variable names vs those who did not, and those who used comment blocks vs those who did not.

Step 5: Testing our function

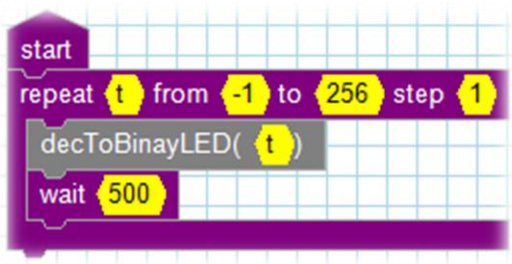
Our function is now ready and should be working correctly. We need to test it to make sure it does work. It can be tested simply like this:



Students can try different numbers to make sure all do work as expected.

A.S. Relevance:	Specifying data for a test case for the task (**2.45 Achievement**), **Merit** requires inclusion of expected & boundary cases while **Excellence** requires inclusion of exceptional cases (**2.45**). Testing and debugging program on sample expected inputs (**2.46 Achievement**)

A competent student might go further by conducting an efficient and comprehensive testing incorporating all expected, boundary and exceptional test data. This could be easily achieved by building the folowing simple program:



- ✓ **Tip:** The purpose of the wait block is to put a convenient gap in time between each number so our eyes can spot it. An important issue that must be considered carefully when programming with 12Blocks is that variables are all global. This is why we used a different index name here (t) in our loop. If we used the default index (n) which has already been used in our function definition then the program will not work as expected.

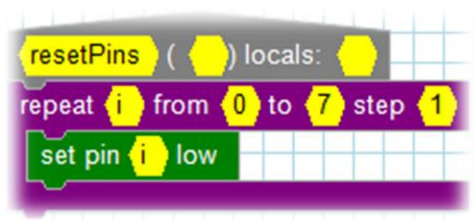
- ✓ **Teacher's Tip:** A student doing something similar to the above testing tool would probably be eligible for "Achievement with Excellence" as they have demonstrated the awareness for the need to test on all possible input data.

A.S. Relevance:	Testing the algorithm on all test data (**2.45 Achievement with Excellence**), Testing in an organized way on expected and boundary inputs (**2.46 Achievement with Merit**), Comprehensive testing in an organized and time effective way (**2.46 Achievement with Excellence**)

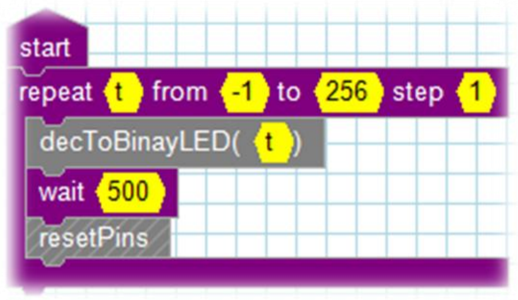
Step 6: Debugging and fixing bugs

A student who has implemented a testing tool like the one above will discover a bug in our program. Essentially, the function we have implemented will send the signal 'High' to the corresponding pins but the pins will remain outputting a High signal unless we send another command to change the signal to Low (off). Thus, numbers will not be represented correctly when the function is run consecutively.

To resolve this problem the student needs to use another block called '*set pin low*'. This can be achieved by writing another simple function that loops over all 8 pins and reset them all to Low. The below picture shows such a function:



The student can then use this function to reset all the pins after each number has been displayed. A correctly working test program can now be built as follows:



A.S. Relevance:	Testing the program on all test data (**2.45 with Excellence**), Testing and debugging on expected & boundary inputs (**2.46 with Merit**), Comprehensive testing and debugging (**2.46 with Excellence**)

★ **Bonus Value:** The above test program is itself an amusing and entertaining activity and will greatly help students comprehend and appreciate the binary system. It also makes it easy for students to spot the pattern for counting in binary by seeing how lights come on/off and move one bit at a time to the left.

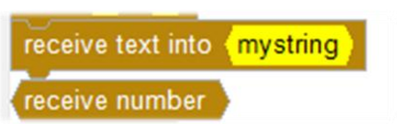
A demonstration video is available on YouTube for watching:
http://www.youtube.com/watch?v=ffuLAw2EeZM

We can now move to phase 2 of our activity where student will need to take input from the user and display it in LED light binary pattern.

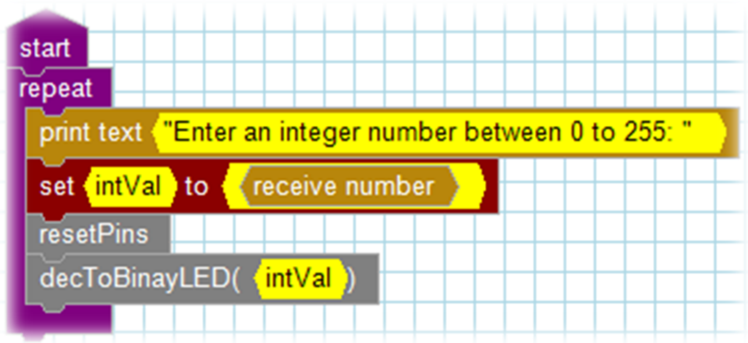
Phase 2: Writing an interactive program

Step 1: Reading user input

12Blocks make it very easy to read user input and use it. It has a convenient block to read numbers and another similar block to read strings (text). Both can be found in the *terminal* section of the library and are shown on the left. The '*receive number*' block can be easily used as follows:



This will allow the curser in the "Terminal" window to be in a state ready to accept user input. Once the user types something and hits enter, the value will be immediately received and stored in the x variable ready for usage. Preferably, we might want to print something on the screen instructing the user on what to do before trying to read user input. The below picture shows an example program that would read user input and feed it to our previously developed function:



- ✓ **Clarification:** As you will quickly notice, the program prints an instruction message on the screen and then waits for user input. Once user enters a number and hits enter, the entered value is stored in the variable '*intVal*'. The function **resetPins** is then called before calling the **decToBinaryLED** function. At first instant, it might look illogical to reset first before lighting up the LEDs. However, resetting after lighting up the LEDs would mean that as soon as the LEDs light up showing the binary number, they would immediately go off making it impossible to actually see any result. By calling **resetPins** first though, it means that the pins will not be sent a Low signal until the user enters another number. This is because the program stops and waits for user input every time just before calling the **resetPins.**

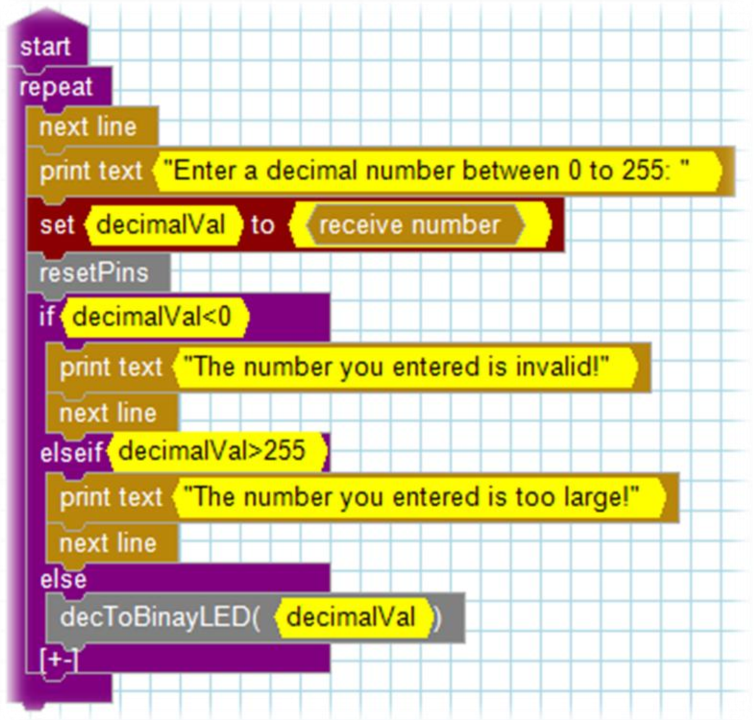
- ✓ **Clarification**: You will notice that an infinite repeat loop was used to keep the program running indefinitely. This is the easiest way to make the program keeps asking for user input. The Propeller board we used has only 8 input pins and does not have any controllable buttons. If a different board with a controllable button was used, then student can use that button (or add one if extra pins available) to start and stop the program. The "When" and "Stop" blocks in the control section can be used for that purpose.

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

Step 2: Adding User Input Validation

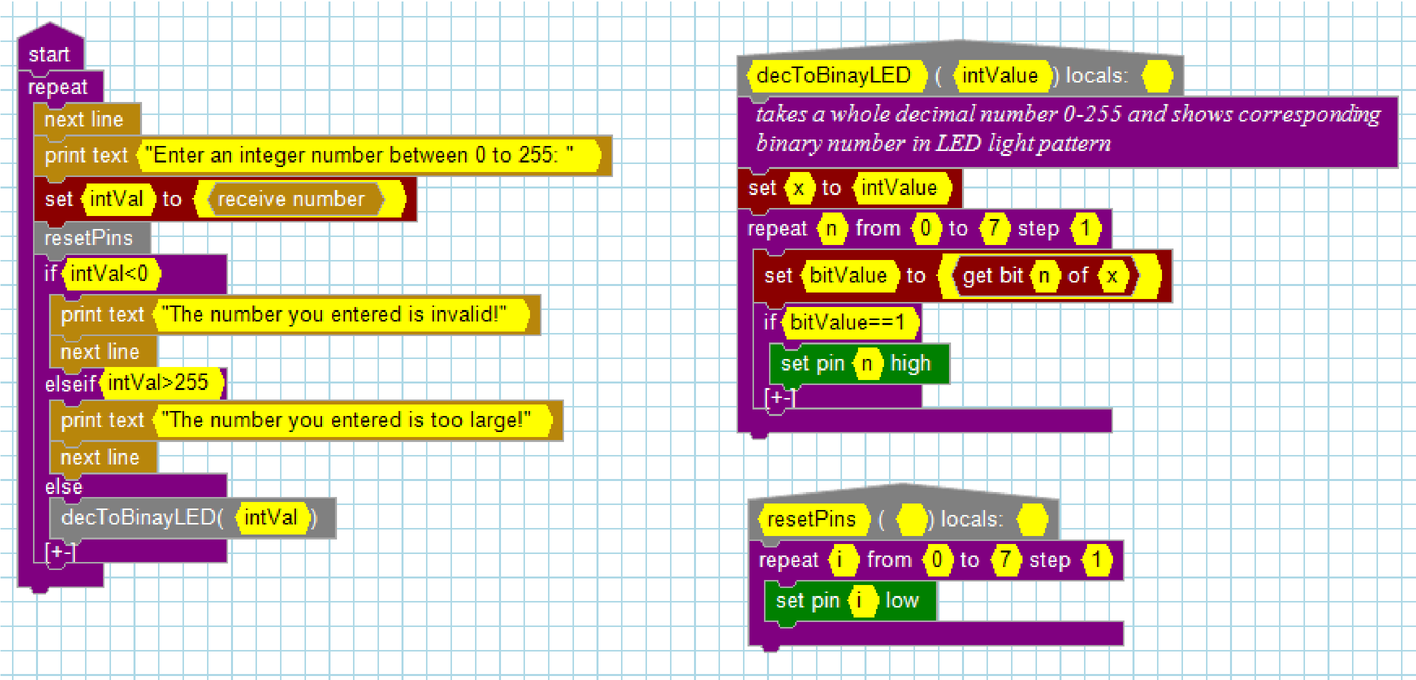
Students should quickly realize that the user might enter invalid input to the program and cause it to fail. Competent students should be able to enhance the above program by handling user input properly and validating it before passing it to the **decToBinaryLED** function.

An If-statement block can be used to check if the data entered by the user is within the correct range expected or not. The function can then be called only if the data was ensured to be within the accepted range. The picture below shows the enhanced program with user input validation:



VII. Complete Code in a Picture

The complete program is now shown below. A 12Blocks file of the complete program is included with this guide.



VIII. Extensions/ Variations:

Some other interesting variations of this LED binary number activity can be created. We have already shown earlier one variation of these which is the LED binary counter. Another variation could be an *LED Chase Lights* game where students use the function created earlier to build another simple program that produces patterns of LED chase lights. For example, students can try to display these specific numbers in sequence (1, 2, 4, 8, ...etc) which when displayed in the LED binary representation will have a nice effect of light chase.

This activity is nicely demonstrated in this video on YouTube:

http://www.youtube.com/watch?v=KMzUFANTrgM

- ★ **Educational Value:** The LED chase lights activity have a good educational value in that it will require students to manually convert specific binary numbers into decimal in order to feed them to the *decToBinaryLED* function so that the program produces the correct pattern of light chase. Students can try first a one bit chase, then two bits, etc.

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

IX. 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.