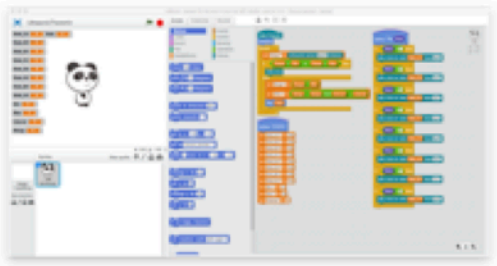
A Gentle Introduction to Robotics

Volume 1: mBlock and the mBot



Charles McKnight

A Gentle Introduction to Robotics

Volume 1: mBlock and the mBot

Charles McKnight

2016
Senestone, Inc.

* * * * * * * *

ISBN-10: 0-9975317-0-3
ISBN-13: 978-0-9975317-0-1

Dedication

When I was a child my grandfather told me to make sure that I spent time educating children when I was an adult or I would deserve the future I would get. This book is dedicated to those who taught me as a child and to the children who continue to teach me as an adult.

I also want to express my gratitude, love, and appreciation for my lovely wife, Jie, and my son, Ian, whose patience and support enabled me to complete this book.

Contents

Acknowledgments

No book is truly an individual effort and as such I would like to express my thanks and appreciation to Darach and Tom Ennis, Luke Faith, Danyu Zhu, and Ian McKnight for taking the time to review and critique this manuscript. It just wouldn't have happened without your insights and comments to make the text better than it was.

I would also like to extend my thanks and admiration to the following:

Makeblock for creating a wonderful set of robotics kits and parts.

The Lifelong Kindergarten Group at the MIT Media Lab for creating Scratch.

All of the friendly, helpful people on the Makeblock and Arduino forums.

Preface

Robots All Around Us

Robots are appearing all over the globe at an ever-increasing pace. Is it an invasion from Mars or worse, a realization of SkyNet from the Terminator[1]? No, gentle reader, nothing so dramatic. The increasing interest in robotics is driven by the appearance of relatively inexpensive hardware that enables ordinary people to create extraordinary machines to fulfill a variety of needs. South Korea has recently edged out Japan for the country with the highest robot density with 347 robots per 10,000 manufacturing employees[2] (Japan has a density of 339 robots per 10,000 manufacturing employees). When you get beyond the manufacturing numbers and begin to add in the large number of autonomous vehicles (UAVs, self-driving cars, etc.), the trend towards an increasing use of robots is undeniably clear as are the opportunities for robotic-related skill sets.

With the global focus on Science, Technology, Engineering and Mathematics (STEM), software development and other technology-based education is being introduced at an increasingly early stage in schools, often as early as mid-elementary levels. Many middle schools and high schools in the United States offer technology classes for web robotics as a normal part of their curricula so that students have the opportunity to gain exposure to the technologies that will potentially be part of their daily lives. The rise of the maker movement with its emphasis on building things in the physical world rather than in a virtual world is also spurring the interest and development in multiple areas of technology, including robotics.

What is a Robot?

Maja J. Matarić provides a clear, concise definition of a robot[3] :

> "A robot is an autonomous system which exists in the physical world, can sense its environment, and can act on it to achieve some goals."

This definition excludes tele-operated machines, such as remote-guided UAVs or those machines requiring a human operator to provide the intelligence to perform their functions. While the definition flies in the face of the common usage of the term, thanks in no small part to television shows and movies, these machines are decidedly not autonomous and therefore really should not be referred to as robots. It should be noted that some robots can be trained to perform their functions though the use of an external harness controlled by a human, but in this case it is for training only and that the robot is capable of performing its functions autonomously after being trained.

[1] Daly, J. L., Gibson, D., & Hurd, G. A. (Producers), & Cameron, J. (Director). (1984). The Terminator [Motion Picture]. United States: Orion Pictures.

[2] Source: "10 Countries with Most Robot Density" by Ejaz Khan, http://goo.gl/L9q9oD

[3] Matarić, Maja J. "The Robotic Primer". 2007. Massachusetts Institute of Technology. ISBN 978-0-262-63354-3

What is Robotics?

Robotics is the study of the branch of technology that deals with the design, construction, operation, and application of robots. The field of robotics is an intersection of many other fields, including, but not limited to, electrical engineering, mechanical engineering, physics, and software engineering. With recent developments, many roboticists are also turning back to Nature for inspiration and are mimicking the mechanisms and patterns found there.

Along with the increased interest in robotics we are seeing an increase the number of hobbyist kits available for the amateur roboticist. Kits such as the LEGO™ Mindstorms™ EV3 Robotics Kit, VEX IQ and VEX EDR from Innovation First International, Inc., etc., as well as smaller, less expensive kits, such as the Parallax BoEBot or the Makeblock mBot that is used in this book are used in many schools as part of their STEM programs.

Some kits, such as the LEGO™, Parallax or VEX kits are based on a proprietary controller although the firmware can usually be modified to support other languages. Other kits are based around Open Source hardware platforms, such as the Arduino or Raspberry Pi and provide a relatively less expensive entry point. Regardless of the platform, programming language, or programming environment, they all utilize the same principles with regard to robotic programming, i.e., the platform utilizes sensors, motors, and other devices to sense and interact with its environment.

Why is Robotics Important?

New discoveries and uses for robots are in the news nearly every day. In Japan, robotic usage ranges from tasks that are too dangerous for humans to the more personal task of assisting the elderly. As the field continues to advance, it is vital that people understand the benefits that accompany the use of this type of automation and how positive uses can improve the quality of life for everyone.

There are many people who see the development of robotic technology as a threat that will result in the loss of many jobs as robots and other forms of automation come into play. Interestingly, there are just as many people who believe that just as the mechanical loom displaced the textile workers of 19th century England only led to more affordable goods in greater quantities, so will robotics and automation lead to lower prices, greater availability of goods and services, and the creation of new jobs.

Regardless of which side of the argument you are on, developing the technology skills necessary to thrive in the coming economy is vitally important.

Why This Book?

This book is an introduction to robotics and robotic programming using the mBlock[4] development environment and the mBot[5], a STEM-oriented robot kit. Although there are many books available for programming the LEGO™ Mindstorms™system and Arduino-based systems in general, most of the robot kits are in the $350 range and up and are therefore beyond the reach of many schools and students.

At the time of this writing, an mBot could be purchased for less than $100 USD and offers an excellent starting point into understanding robotic programming, the use of sensors, and controlled movement via electric motors. The kit is based on the Arduino ecosystem and can be further expanded through additional sensors and other parts that are available from Makeblock[6].

The mBlock development environment is based on MIT Scratch 2.0 offline editor and is fully open source so that additional blocks can be added to support new sensors and devices. With a

[4] mBlock is derived from the MIT Scratch project by Makeblock (http://mblock.cc).
[5] mBot is a robot kit meant for use in STEM environments that has been developed by Makeblock.
[6] http://makeblock.cc

rapidly growing community, the combination of the mBot and the mBlock development environment provide a compelling approach from both a cost and capabilities standpoint.

In any case, enough preaching. Let's get on with learning!

Questions about the book may be sent to epubs@senestone.com.

Source code is available at Github https://goo.gl/CWROM5.

1 | Getting Set Up

Before You Start

Programming a robot requires that you have a robot (the mBot in our case) and an installation of the mBlock Integrated Development Environment (or IDE). When you have successfully built the robot and have attached batteries, it is time to download the IDE from Makeblock (http://www.mblock.cc/index.php). You will need to download and install the version for your operating system.

> **NOTE:** Only Mac OS X and Windows are supported at the time of this writing. If you use Linux, you will either need to install Wine and use the Windows version or some other virtualization platform that supports Windows or OS X.

Installing the Arduino Driver

When you have downloaded and installed the mBlock programming environment, you will need to click on the Connect menu item, and select the Install Arduino Driver from that menu. Follow the instructions that appear on your screen to install the Arduino driver.

Figure 1: Install the Arduino driver

The Arduino device driver enables the mBlock IDE to recognize the mCore circuit board when it is attached via a USB cable. Check the Makeblock forums[7] to see where the most current drivers can be downloaded. If you are using OS X and have installed the driver but you still cannot connect to the mBot, you may have a driver conflict.

There is an excellent set of troubleshooting instructions at this link[8]. The site owner also sells signed USB drivers for OS X. If you go this route, you may need to create a symbolic link to the

[7] forum.makeblock.cc
[8] https://goo.gl/FF9MYc

driver because mBlock has trouble with the length of the name of the driver[9]

To communicate with the mBot (which is based on the Arduino Uno architecture), you must be able to connect via the USB (Universal Serial Bus) serial cable provided with the mBot kit. This serial connection enables you to upgrade the firmware on the mBot, reset the default program, or to upload your own programs by sending the information to the onboard programmer. In turn, the onboard programmer overwrites the memory of the mBot with the new information.

What is an mBot?

The mBot is an affordable, STEM-oriented robotic kit produced in China by Shenzhen Maker Works Technology Co., Ltd. The kit is based on the Arduino Uno architecture and takes approximately 15-30 minutes to assemble. Due to its target audience, there is no soldering required. Makeblock also offers a number of add-ons to expand the mBot platform.

Although the mBot is based on the Arduino Uno platform, it is not a 'genuine' Arduino. However, you can program the mBot via the mBlock programming environment (which is based on MIT's Scratch 2.0) or the Arduino programming environment included with mBlock.

A third option is to download the libraries from the Makeblock-Official repository on Github and install them into a standalone Arduino programming environment. This last approach is beyond the scope of this book, but will be covered in the next volume in the series.

When writing programs for the mBot, it is important to realize that the mCore board is a clone of an Arduino Uno, an open source micro controller board. The architecture of the Arduino Uno (and its clones) provides 32 Kilobytes of Program Memory (PROGMEM) and 2 Kilobytes of memory for variable storage (SRAM).

This differs from the memory in a general purpose PC where all of the memory is shared by the program and its variables and requires care when defining variables. In fact, if you use up all of the available 2 Kilobytes of SRAM, your program will crash so be careful!

Building the mBot

When you open your mBot kit, you will find instructions that will walk you through building the mBot. It is very important that you follow the instructions carefully to ensure that the mBot is built correctly[10].

Connecting Power

If you have a Lithium Polymer (LiPo) rechargeable battery (available separately from the kit), you can use it in place of the battery pack that comes with the kit and charge it through the USB cable because the mBot has a charging circuit built into its circuit board. The LiPo battery is likely less expensive in the long term because it can be recharged where you would have to replace the AA batteries when they run low.

You can also use rechargeable AA batteries and recharge them if you would prefer. If you use AA batteries, it is strongly advised that you replace (or recharge if you are using rechargeable batteries) all of the batteries at the same time. Otherwise, the weaker batteries will drag down the charge of the other batteries at a faster rate.

[9] Peter van Nes posted a fix in the Makeblock forums. (http://goo.gl/px8od8)

[10] Be sure to pay extra attention when connecting the motors. Hooking the motors up backwards is one of the most common errors.

Connecting to the mBot

The mBot kit comes with a USB serial cable that enables the mBot to be connected to your computer. This cable has a USB standard-A male connector on one end and a USB standard-B male connector on the other end.

Figure 2: USB Standard-A (left) and USB Standard-B (right) male connectors

The USB standard-A male connector is plugged into a USB port on your computer and the USB standard-B plug goes into the mBot.

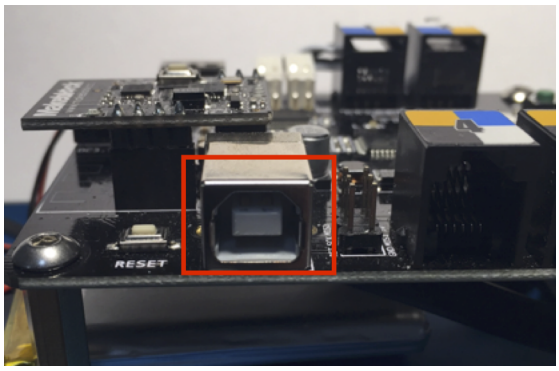


Figure 3: mBot USB Standard-B Female Connector

Connecting to the mBlock Environment

As general rule, turn on the mBot before connecting it to the computer. In addition to avoiding a small power spike that occurs when the mBot starts (that may cause your computer to reboot). This practice will ensure that the computer detects the mBot when it is connected to the computer. After the mBot is connected to the computer, click on the Connect menu item, followed by clicking on the Serial Port submenu item.

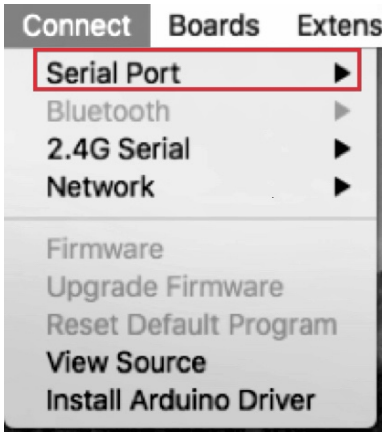


Figure 4: USB Serial Port Connection menu

What you will see under the Serial Port submenu item will depend on the operating system. On a Windows system, you will see COM ports. If more than one COM port appears, it is likely that the mBot is on the last COM port on the submenu. If your computer has OS X, you should see an entry for /dev/tty.wchusbserialxxx or something similar.

If you have successfully connected the mBot to the mBlock IDE, you should see the message "Serial Port Connected" in the title bar indicating that the mBot is connected.

Congratulations! You are now ready to begin writing and uploading your programs!

2 | The Software Development Process

Developing software may seem like a difficult task, but it follows a process that will enable you to be successful in creating your programs for the mBot. The process generally consists of five major steps:

1. Requirements
2. Analysis
3. Design
4. Implementation
5. Testing and Debugging

There are many ways to approach software development, but unless you are just hacking away at a program you will find that spending some time with each of these steps will decrease the amount of time required to be successful in your development efforts.

Requirements

Requirements specify what the program must do, such as turning on an LED, making sounds, or solving a maze. They also provide a means for the developer to demonstrate that the program is complete. There are many ways to define requirements, but a good general rule of thumb is limit each requirement to a single thought or capability. For example, solving a maze might be the overall requirement for a robotic program. However, it might be good to know if there is a time constraint for solving the maze because this will help determine how you approach the design of the program. Likewise, if a requirement cannot be fulfilled the developer needs to call this out early on so that the requirement can be refined or eliminated. In any case, all programs start with some set of requirements so that you will know what you are building.

Analysis

Analysis is the process of reading and understanding the requirements so that you can identify what you will need to do to perform the work associated with those requirements.

Analysis also allows you to identify what other things need to be done to support the design for the implementation of the requirements, such as whether or not you need to connect with other systems or if you need to do the work within a particular period of time.

Design

Design is the process of determining how that you will implement the requirements based on the results of your analysis. This point in the development process is where you design the program logic in detail.

Note: Analysis answers *what needs to be done* and Design answers *how does it need to be done*. The two are often intermingled which can lead to errors. Make sure that you take the time to understand *what* before diving into *how*.

Implementation

At this point of the process, you are actually writing the code and test code for your application as well as updating your design and/or analysis documents if you discover that you have missed something.

Testing & Debugging

This final phase of the development process ensures that your application works as intended and fulfills the requirements. If it is not working correctly, this phase is where you would spend time debugging the errors in your code.

As with implementation, if you discover something that was not originally seen in your analysis or design, you need to go back and update those items so that the final code will match them.

Why All the Fuss About Going Back to Update Analysis and Design Documents?

For our applications, it is a matter of developing and practicing good habits. However, other reasons for updating the analysis and design documents are:

* To enable others to understand what you did, how you did it, and why.
* To remind yourself of what, how and why when you come back to that code in a few months.
* To identify reusable items for another program.

But That's Not Agile! That's a Waterfall!

If you study Agile programming, you will see that it essentially breaks down a large problem into smaller pieces instead of trying to build the entire system all at once. The rationale is that shorter iterations allow the product to be tested more frequently as it grows and evolves as opposed to waiting until the end to test everything. Each iteration ends with a shippable, functional product that has been well tested.

This approach supports the notion that requirements may change over the lifetime of the development and provides the development team with the opportunity to reprioritize requirements based on customer feedback. However, each piece is governed by one or more ?user stories? which act as the requirements. The developers responsible for each piece (or sprint) have to spend time understanding the requirements (analysis) and determining how they will implement (design) the code. The Agile process also emphasizes automated testing at a number of different levels.

With all of the same steps being applied to each iteration, I'd submit that Agile frequently is little more than a series of small waterfalls rather than one big one.

Hopefully you can understand why following the process is important. It is like solving a complicated problem in mathematics where you must follow all of the steps to get a correct answer.

3 | Programming Guidelines

Naming Conventions

What is in a name? Well, everything really. Choosing good names makes it easier to understand the intent of the code. This concept applies to variables and custom blocks. Generally speaking, one should use a highly descriptive name so that it is obvious what is being looked at.

For example, naming a variable 'a' is certainly valid but a name like 'Color' would let the reader know that the variable is a color, not just a random value. Naming a variable 'CurrentColor' is more readable than a one or two character name. Remember that the more descriptive the name, the more self-documenting the code will be and this practice will reduce the number of comments needed to describe the code.

Valid Characters

Valid characters for program names depend on the underlying operating system, but usually are comprised of letters, numbers, or underscores. However, names for blocks and variables should begin with a letter or an underscore. The reason for this constraint is that the Scratch code will be translated into C/C++, the native language of the Arduino, and that language requires that the first character be a letter or an underscore. The rest of the name can be any combination of letters, numbers, and underscores.

You might wonder if spaces can be used, but they cannot. When mBlock generates the underlying Arduino code, it uses the name of the block or variable. Because spaces are not allowed in Arduino code, everything after the space and the space itself will be ignored. If you have two (or more) blocks or variables that start with the same characters before the space, the Arduino compiler will treat the duplication as an error and it will not finish compiling or uploading your code.

Snake-Case vs. Camel-Case

If a variable or block name has more than one word in it, the two most common conventions are Snake-Case and Camel-Case.

Snake-case uses underscores to separate each word. For example 'my function' becomes my_function and 'my variable' becomes my_variable. The proponents of this convention assert that the underscores make the names more easily read than camel-case names.

Camel-case names mix uppercase and lowercase letters to signify the beginning of each word. The general practice is to make the first word of a block (function/method) all lowercase and to have the rest of the words start with a capital letter. Variables differ slightly in that the practice is to start all of the words with a capital letter.

For example, a block named 'my function' becomes myFunction and a variable named 'my variable' becomes MyVariable. Camel-case has been the choice for many languages, such as Java[11], and as such has a large number of proponents as well.

[11] Java is owned by the Oracle Corporation

Both naming practices are used in industry, although snake-case has fallen out of favor and camel-case is more often seen. You should use whichever practice works for you, but be consistent.

Design First

One of the first things that new developers tend to do is to jump in to writing code for their programs. While this may work out for a simple program with a few steps, it frequently results in disaster and a lot of rework in larger programs. The lesson that should be learned is to think about what you need to do and take the time to design your programs. The actual coding of the solution becomes much easier and there are fewer surprises that require rework.

Current popular programming philosophy revolves around Agile programming seems to eschew 'big upfront design' in favor of directly coding from user stories and constantly 'refactoring' the code. While this approach has merit, the author is of the opinion that some thought about the problem needs to occur before jumping into coding a solution regardless of the size of the program. If you do not understand the problem, you will spend a lot of time re-coding your program!

Frequently a simple flowchart will suffice to document and test your thoughts before spending time actually coding the solution. In the end, it is your choice on how you wish to work, but choose wisely and be consistent.

Breaking Down the Boulders

When working through a large problem, it is a good idea to break the problem into smaller pieces that are more easily solved. This 'divide-and-conquer' process also helps you to break up the program into blocks of code that may be potentially reused later in other programs.

Do One Thing Well

A common error in design is the tendency to have a 'do-everything' block in the flowchart. This type of block represents a large quantity of functionality and often looks like the center of a spider web in a flowchart. This practice is heavily discouraged because in addition to not communicating exactly what is going on inside of the block, it makes it challenging to implement and debug code based on the design. To avoid this situation, make sure that your blocks define one operation or concept.

In high-level flowcharts, you may see blocks that represent a related set of functionality. Do not confuse these with a do-everything block.

A common approach defines the high-level program logic as a flowchart and then provides additional diagrams that show the specific functionality within a high-level block. This top-down approach enables the designer to provide the right level of detail for each diagram.

Flowcharting and Pseudocode Design Tools

Two common methods for designing programs are flowcharting and pseudocode.

Flowcharting

Flowcharting provides a graphical approach to designing the flow of the program, enabling designers to see how the program will work and how data will flow through it (See Appendix A: Flowcharts for a brief overview of the different common symbols).

The default flow of execution is sequential unless otherwise altered through the use of loops and decision-based branches. As you will see in Figure 5, a sequence structure is simply a series of steps that are executed in the order they are arranged, i.e., a sequence. Most mBot programs are

written around this concept due to the single-core, sequential execution that occurs within most micro-controllers.

Much of our programming in the mBlock environment will be sequential. The micro-controller on the main board of the mBot does not directly support multi-tasking nor does the Arduino have an operating system that supports multitasking[12]. In fact, the Arduino doesn't have an operating system at all so your program is in direct control of the robot hardware. Therefore we must formulate our programs with this in mind. The important thing to keep in mind is that most sequential programs have a definite beginning and a definite ending.

The first thing most programmers generally do after the start block is to define any variables and set the starting values for those variables. This step is called 'initialization' and will be seen in most programs.

If you have no variables to define or to set initial values for, you do not have to have an initialization step. This will be the case for many simple programs, but be aware that it is a good practice for complex programs to do as much of your initialization in one place as possible.

This practice cuts down on the need to search through all of the code to determine what values your variables are starting with. We will cover methods to define and initialize variables when we begin writing programs that need them.

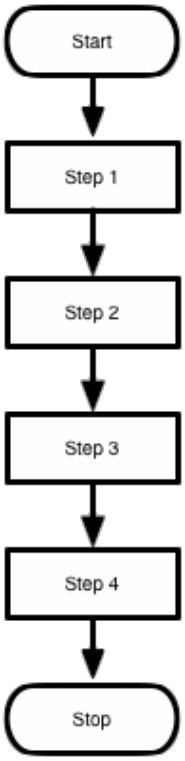


Figure 5: Simple Sequence Flowchart

As the name implies, a sequence is executed one step at a time in the order listed. Each step in the flowchart is a process, or series of program instructions. In more complex programs, the flowchart will contain decision symbols that define places where the flow of the program execution may change based on some value or condition.

This approach works well to provide a quick insight into the flow of execution, but may be light on the details in each step. If a step contains a number of sub-steps, it may be diagrammed out

[12]There are advanced techniques that mimic multitasking, but they are beyond the scope of this book.

on a separate page. With modern flowcharting tools, one can often dive into the step to see what sub-steps are occurring where doing so with paper diagrams would be cumbersome and tedious.

Pseudo-Coding

Pseudo-coding defines the program by expressing execution steps as short phrases. An example of the program above in pseudocode would be:

```
When the program starts
    Execute Step 1
    Execute Step 2
    Execute Step 3
    Execute Step 4
End the program
```

Figure 6: Pseudocode Example

This format works well when each step has a significant amount of details that need to be called out. It also can use additional indented steps to represent the sub-steps in a process. However, reading through a large pseudocode document can be quite tedious.

Designers will often use flowcharts for the overall program flow and detail the specifics in pseudocode. There really is no hard and fast rule other than use what works for you. However, make a point of designing first so that you know what you need to do!

Test, Test, Test!

Testing allows you to verify that your program is correct and that it is performing as you expect. When you are designing a program it is important to be thinking about how that you will test each part of the program.

This is an area where going through the process of creating the flowchart diagrams will pay dividends because it will allow you to think about where things might go astray and what that you should do about them before you spend time coding a bad solution.

A good test will select a feature and determine whether or not the feature is performing as expected. This is called golden path testing.

However, it is equally important to have tests that intentionally use bad data to see how the program will react. Quite often this type of testing will reveal instances where you might need to rethink how you are doing things to avoid the errors.

Another area is performance testing where the program is tested to see how quickly it can execute, to identify potential places that the program can be optimized, and also to determine how resources such as memory or processor time are being used.

Each of these areas (and many more) enables the developer to create a robust program with the best possible performance. There is an entire industry that is focused on software and hardware testing so you should not overlook learning these valuable skills.

4 | mBlock Programming Environment

mBlock and Scratch

mBlock programs are written in a language named Scratch that is developed by the Massachusetts Institute of Technology. It is a visual, block-oriented language that is used for teaching programming. However, it is also suitable for a wide number of other applications.

For the purposes of this book, we will cover the parts of the mBlock IDE that we will use to write programs for the mBot and the Scratch palettes that apply to mBot programming. We will also do a walk through of the blocks on the Robots palette that are specific to the mBot.

Although we will discuss some of the mBot blocks in this chapter, you may consult Appendix B: mBot Blocks as a reference for the mBot-specific blocks and their functions. For blocks in the other palettes, you may consult any of the many online references and tutorials for Scratch 2.0 listed in Appendix F: Resources.

A Quick Tour of mBlock

mBlock is an Integrated Development Environment (IDE) that provides all of the tools necessary to create programs for the mBot and other Makeblock robots. mBlock also integrates a version of the Arduino IDE, the native environment for writing Arduino code (remember, the mCore board is an Arduino clone) and uses these tools when compiling and uploading code to the mBot.

When you first launch the mBlock IDE, you will see the following screen as mBlock launches into Scratch Mode:

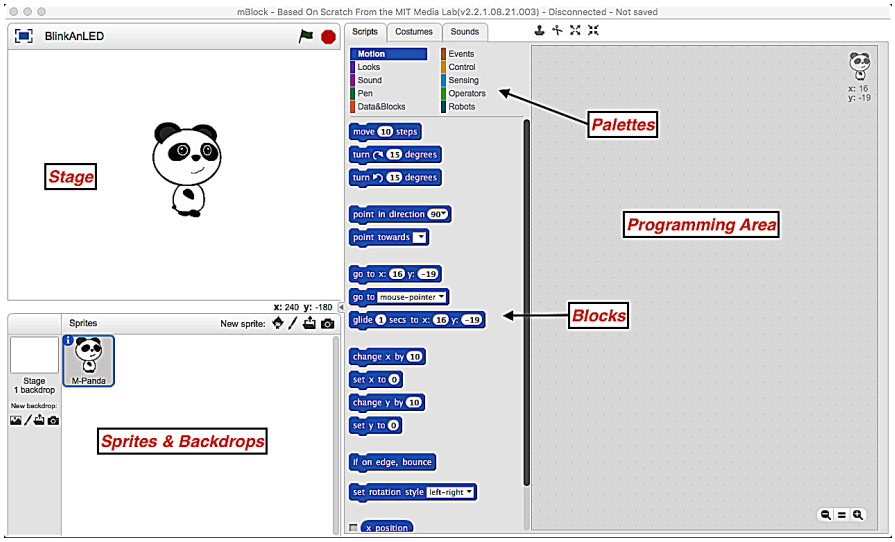


Figure 7: mBlock IDE

mBlock Panels

Stage	In the upper left-hand corner you can see the stage where any sprites will appear. In this case there is a panda sprite.
Sprites & Backdrops	The lower left-hand corner contains a list of sprites and backdrop images for the stage.
Palettes	The palettes for Scripts, Costumes, and Sounds occupy the center of the mBlock IDE.
Programming Area	The area to the right of the palettes is the programming area. You can drag blocks from the palettes to create programs for the mBot. Because this is a modified version of the Scratch 2.0 IDE, you can also create Scratch programs as well. When running your programs from within the mBlock IDE, you are allowed to use any of the Scratch blocks as well as those on the Robots palette. You can also use the sprite to display debugging messages.
Zoom	There is a pair of small magnifying glass icons in the bottom right-hand corner of the programming area that will allow you to zoom in and out of the code area. You can also click and hold the left mouse button to pan left or right as needed to move around the programming area.

Table 1: mBlock Panels

All Scratch Blocks Are Not Supported in mBot Programs

It is important to note that if you create an mBot program that will be uploaded to the mBot via the USB cable, there will be many Scratch blocks that will not work. The IDE will notify you when you are using unsupported blocks by displaying a pop-up dialog that contains the unsupported blocks. In Figure 8 you can see an mBot Program 'hat' block followed by a Scratch *say [message]* block.

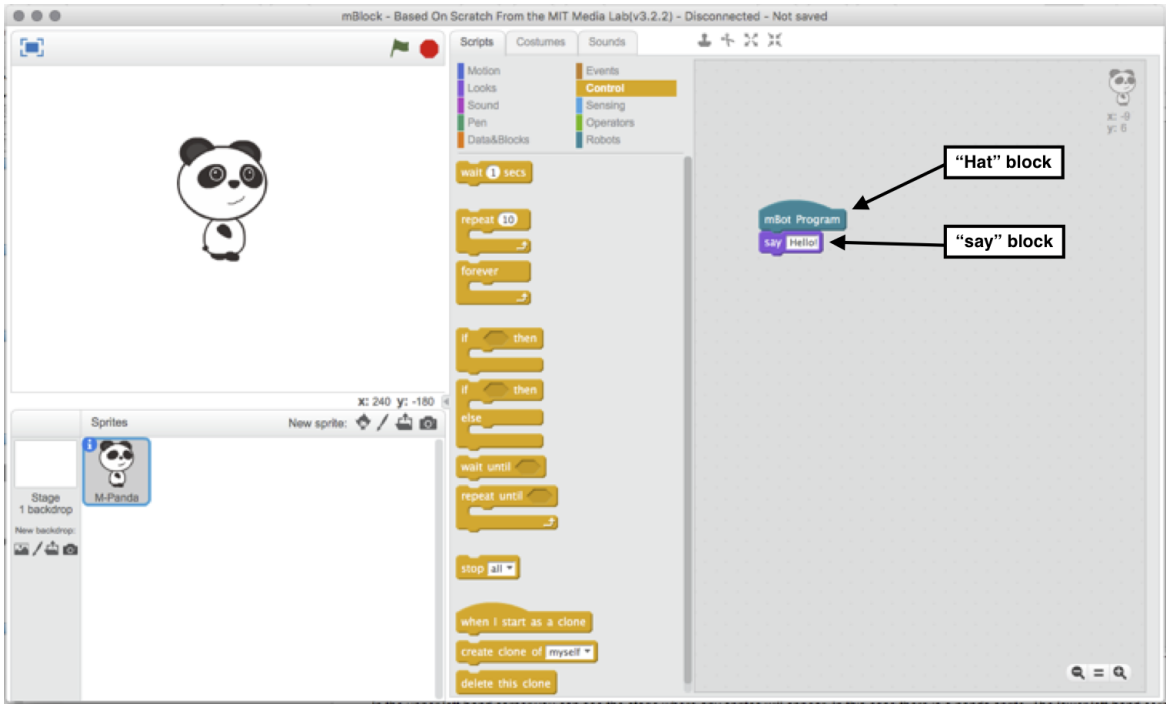


Figure 8: mBot Program with an Unsupported Block

If you attempt to compile and upload this program, you will see the dialog below:

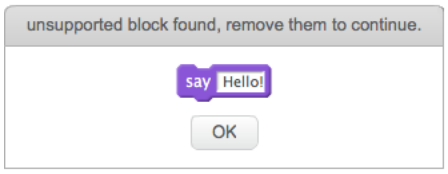


Figure 9: Unsupported Block Dialog

Why does this occur? The unsupported blocks generally require the program to be run on a computer and its system resources, and in this case the say block requires that its message be displayed on screen by the sprite in the Stage area. However, the mBot does not have a built-in screen so this function will not work if the program is loaded into the mBot.

For an mBot program, the code generator only generates code for the functions associated with devices on the mBot. Any block that is not supported will cause the Unsupported Block dialog to pop up[13] and mBlock will not allow you to continue until you have removed all unsupported blocks from your program.

[13]Occasionally you will also see the Unsupported Block Dialog pop up when you load a new program to replace one that is already loaded in mBlock. This is a minor bug and can be safely ignored.

Arduino Mode

In the Arduino Mode (Edit->Arduino Mode), the IDE will hide everything but the palettes and the programming area. The Arduino Mode will also disable any palettes or hide any blocks that are not supported by the Scratch to Arduino code generator so that you cannot select anything that cannot be used. Therefore, it is recommended that you write your mBot programs in Arduino Mode to avoid inadvertently using unsupported blocks.

Arduino Mode also opens a window on the right-hand side that shows the Arduino code being generated as you add, modify, and remove blocks.

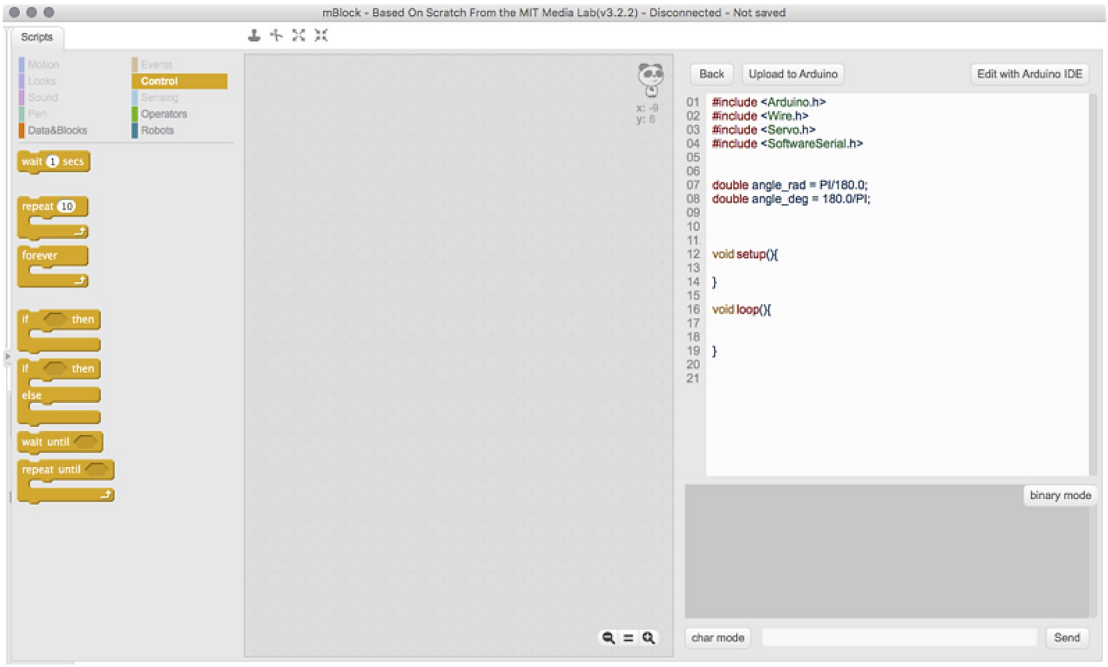


Figure 10: Arduino Mode

You should see several buttons over the Arduino code window, **Back**, **Upload to Arduino**, and **Edit with Arduino IDE** (we will not need the binary mode, char mode, or send buttons in this book so they will be ignored). Let's briefly cover their functions.

Back

The Back button will end the Arduino Mode and go back to Scratch mode.

Upload to Arduino

The Upload to Arduino button will launch the compiler to convert the Arduino code you see on the screen to a binary format that can be uploaded to the mBot. If the compilation is successful, the program will be uploaded to the mBot. If it is not successful, the error(s) will appear in the grey area under the white code area. If the mBot is not connected to the mBlock IDE, you will see the dialog in Figure 11.

Figure 11: mBot Not Connected Dialog

If this occurs simply insert the USB cable into the mBot, then connect the mBot to the mBlock IDE via Connect->Serial Port. You should then be able to compile and upload the program to the mBot.

Editing with Arduino IDE

The mBlock IDE is integrated with the Arduino IDE for the purposes of generating code that can be easily uploaded to the mBot, compiling the code, and uploading the code to the mBot. To accomplish this feat, the Makeblock team has also provided a copy of the Arduino IDE that is already configured to work with the mBot (and other Makeblock products).

Let us take a quick look at a sample program while mBlock is in Arduino mode. Note that the code in the white area roughly corresponds to the blocks.

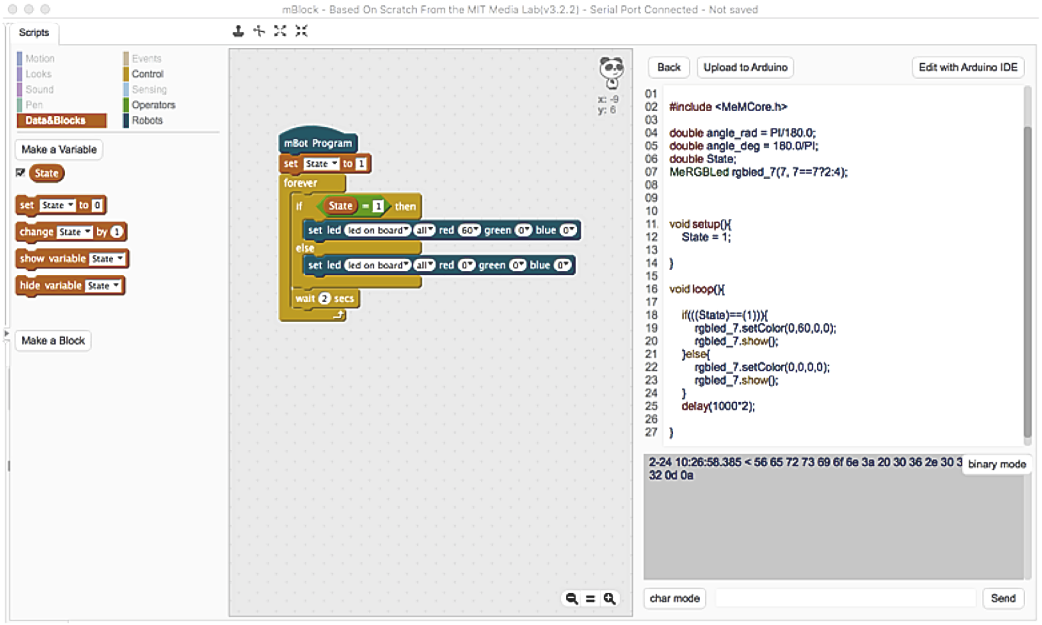


Figure 12: Arduino Code Generated from the mBlock Program

Now click the Edit with Arduino IDE button and mBlock will launch the Arduino IDE and load it with the sample code from the mBlock window.

Figure 13: Arduino IDE with Code Generated by mBlock

Everything in the white code window in mBlock has been copied to an Arduino project. The Arduino environment enables you to write programs that are too complex or too advanced for mBlock. However, the Arduino IDE is beyond the scope of this book and is only mentioned for completeness.

Block Palettes

mBlock has several palettes of blocks that are divided into the following categories:

Palette	**Description**
Motion	Sprite movement blocks
Looks	Appearance-related blocks for sprites / State
Sounds	Sound-related blocks (Stage-only)
Pen	Drawing blocks (Stage-only)
Data & Blocks	Custom variables, lists, blocks
Events	Event management
Control	Script control structures (loops, waits, if/if-then)
Sensing	Sprite/video controls (Stage-only)
Operators	Mathematical and logical functions
Robots	mBot-specific blocks[14]

Table 2: Block Palette Table

[14] mBlock also supports other robot kits, but they are beyond the scope of this book.

For the purposes of this book, we will be looking at the palettes for *Data & Blocks*, *Control*, *Operators*, and *Robots*. Let's look at the kind of blocks that are in the palettes by type and then we will dive deeper to explore each of the palettes.

Block Types

Each palette contains many different types of blocks that can be used to create a program. Each block falls into a particular category, so let's first look at the block types.

Hat Blocks

A *hat* block is used to start a script.

Figure 14: Hat Block

Control Blocks

Control blocks such as loops or if/if-else statements are referred to as *C* blocks because they wrap around a set of blocks.

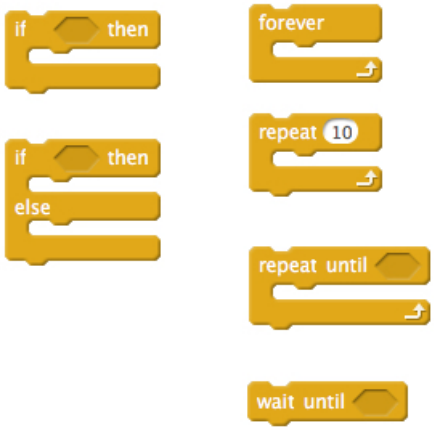


Figure 15: C Blocks

Operator Blocks

Operator blocks are hexagonal (comparison or logical operations) in shape. These blocks either provide comparison functionality or return a value based on the operation.

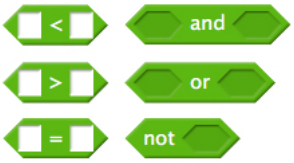


Figure 16: Comparison and Logical Operators

Reporter Blocks

Reporter blocks are oval-shaped and provide a value usually set by the hardware. The Reporter blocks supported by mBlock for robotic programming are currently limited to numerical operations for programs that will be run on the mBot.

Figure 17: Arithmetic Reporter Blocks

Stack Blocks

Stack blocks allow the programmer to set a value or to have the program execute an operation such as turning on an LED or motor. Stack blocks are rectangular and have a tab on the bottom left of the block with a matching indentation on the top of the block.

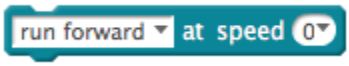


Figure 18: Stacking Block

Control Palette

The Control Palette of the mBlock IDE contains the controls for looping, decisions, and timers.

Delays, Delays, Delays

Hurry up and wait. Although most programs need to execute as quickly as possible, there are occasions where you may want to pause until a condition has been met, such as a button being pressed or a certain amount of time has passed. Thanks to its Scratch heritage, mBlock provides two important blocks for pausing your program.

Wait Block

The *Wait* block causes the program to pause the program for the designated time in seconds. The oval shape of the white input area indicates that the wait block requires a numerical input. This should make sense to you because time is measured in numerical units. You can also put a numerical variable into the input area instead of a number.

It is important to realize that ***NOTHING*** will occur in the rest of the program while the wait block is waiting. While there are certainly times that a wait is valuable you should make sure that it is exactly what you want to do.

Figure 19: Wait Block

In the example above, the *wait [value] secs* block will wait for the number of seconds specified by the *value* parameter.

Wait Until Block

The *wait until* block causes the program to pause until the specified condition occurs (Figure 20). You can see that the wait until block has a hexagonal white input area. This shape corresponds to the comparison and logical operators and indicates that the condition must evaluate to true for the loop to exit. If the comparison evaluates as false, the loop will continue to execute. A variable is normally used for the value being tested.

Figure 20: Wait Until Block

In the example above, the *wait until [condition]* block uses a comparison operator to test whether the value of a variable *Flag* is equal to 1.

Loops and Decisions

When creating a program you will often notice that you have a series of code blocks that need to be executed multiple times. In traditional programming languages, these instructions might be isolated into *functions* if there are many steps to execute.

The mBlock/Scratch environment supports this capability (see Custom Blocks in the Data & Blocks palette section) and it is largely considered a "best practice." However, if you are executing a small number of lines but you need to do so repeatedly, then you might consider using a loop block. The mBlock/Scratch IDE supports three different loop types: *Repeat*, *Forever*, and *Repeat Until*.

Repeat Loop

The *repeat* block (shown below) is also called a for-loop in some languages. The block works by sequentially executing the instructions it contains for the number of times specified in the loop block. If you use a variable instead of a value in the input area, you can control the value of the variable from within the loop and potentially exit early if desired.

Figure 21: Repeat Loop Block

The example above repeats the blocks within the loop ten times before exiting the loop and continuing on with the program. This type of *counting loop* is available in most computer languages and typically takes the form of a *for loop*.

Forever Loop

The *forever* block never stops repeating its instructions and cannot be broken out of. When running your programs on the mBot, this loop runs continuously until you turn off the power to the mBot or load another program.

Figure 22: Forever Loop Block

Repeat Until Loop

The *repeat until* block (shown below) will repeat its instructions until a pre-selected condition is met. The boolean operator specifying the condition will be in the small hexagonal slot on the top line of the Repeat Until block.

Figure 23: Repeat Until Loop Block

If-Then Block

The *if-then* block enables the program to make a decision based on some pre-selected condition. This block will execute the instructions it contains if the pre-selected condition is met. The boolean operator specifying the condition will be in the small hexagonal slot on the top line of the If-Then block.

Figure 24: If-Then Block

If-Then-Else Block

The *if-then-else* block enables the program to select between two paths of execution based on the specified boolean condition. If the specified boolean condition is met (i.e., evaluates to true), the block will execute the first set of instructions. If the condition is not met, the block will execute the second set of instructions.

Figure 25: If-Then-Else Block

Data & Blocks Palette

Variables

Variables are used to hold values (numbers, text, etc.). Each variable has a unique name, allowing the variable name to be used in place of the value. When you select the Data & Blocks palette and click Make a Variable, you will see the dialog below.

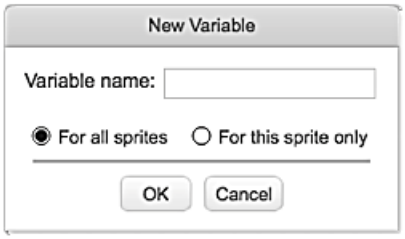


Figure 26: Make a New Variable Dialog

There is a place to enter the variable name, and an option to limit the visibility of the variable to a single sprite (local scope) or to all sprites (global scope). You will see the word scope used a lot in other languages so let's take a moment to understand what scope really means to the programmer.

Local scope (visible to only one sprite) means that no other sprite can see or change the value of the variable. This feature helps to eliminate bugs caused by other sprites changing the value of a variable in an unexpected way.

Global scope (visible to all sprites) is useful when the programmer needs to share information among all of the sprites. Global scope is like a blackboard where anyone can see it and change what is on the blackboard.

With mBot programs, there are no individual sprites, so be sure that any variables you create have global scope that allows all sprites to see them.

Once you have created a variable, you will see additional blocks displayed in the Data & Blocks palette.

Important Note

mBlock's Arduino code generator currently only generates numerical variables. Support for strings has not been added at this time. Attempting to use a variable with a string will generate error messages. This limitation does not apply when writing native Arduino code as we will see in volume 2 of this series.

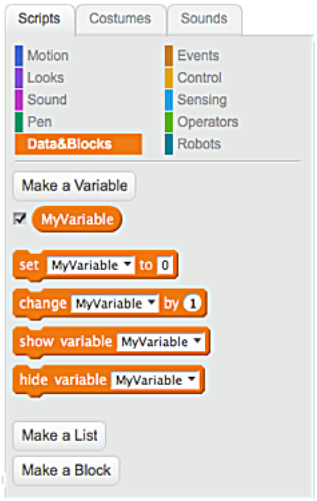


Figure 27: *MyVariable* Created and Added to Palette

Blocks

The Make Block button creates a new custom hat block that you can name. You can add more blocks under the newly created hat block and then use it in your programs as a custom block when you need specialized behavior but want to keep the main program flow short and clean.

So how does one create custom blocks? Select the Data & Blocks palette, and click on the Make a Block button. You will see the dialog below.

Figure 28: New Block Dialog

Click on the purple square to give your block a name.

Figure 29: Naming the Block

If your block needs to receive values to execute an operation, click the arrowhead by Options to see the available input options.

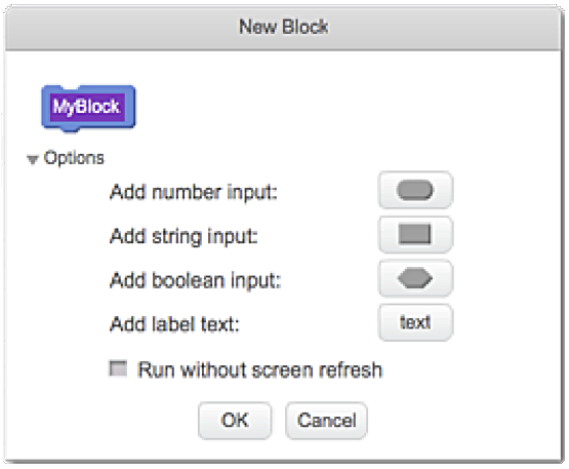


Figure 30: New Block Input Parameter Options Dialog

You should note the shapes of the inputs correspond to the input shapes on other blocks. Specifically, the numeric input shape is an oval, the string input shape is a rectangle, and the boolean (or comparison) input shape is a hexagon.

You can use the *Add label text* to make the name of the block read more like a sentence by inserting words between the other inputs. You can add as many inputs as you like, but as a general rule you should try to keep the number to as few as the block actually needs to do its work.

In the figure below we have added a numeric input and a boolean (logical) input. Note the 'x' in a circle above each of the inputs. If you decide that you do not need the input variable, click the 'x' to remove it.

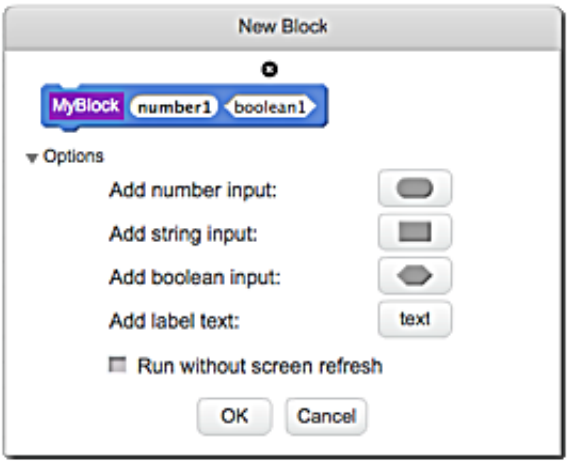


Figure 31: Custom Block with Two Inputs Example

You can think of the input variables (called function or method signatures in other languages) as a template you use to create a copy of the block. From this point you will see a new hat block in the programming area and a new block on the Data & Block palette.

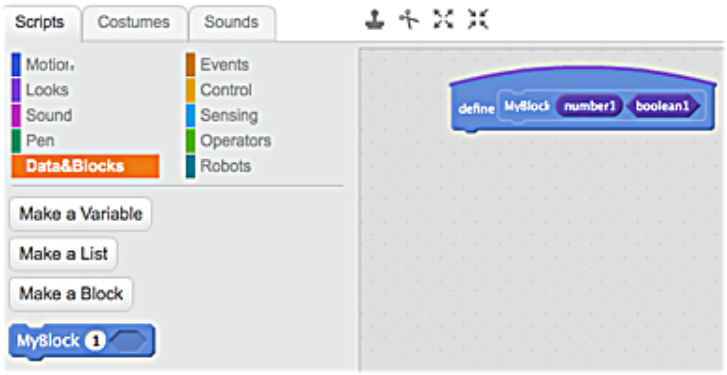


Figure 32: After Creating a Custom Block

You can now use the new block just like any other block. However, it is important to realize that blocks created in your program exist only within your program.

Important Note

mBlock does not support copy and paste operations between projects. If you have developed a set of blocks that you want to reuse, it is a good idea to put them all into single project and make a copy of the project whenever you start a new one so that you will have all of the blocks there without having to recreate them.

Operators

mBlock provides all of the operators available in Scratch 2.0. These operators include mathematical operations, comparison operations, logical operations, and string operations.

Arithmetic Operator Blocks

The arithmetic operator blocks support addition, subtraction, multiplication, and division. The values (or variables) are placed in the white circles on the blocks. Each block acts like a set of parentheses to ensure that the proper order of operations occur when executing a calculation. Additional arithmetic operator blocks can also be placed on a white circle to represent nested operations within an additional set of parentheses.

Figure 33: Arithmetic Operator Blocks

Random Number Within A Range

The *Pick Random* block provides a random number generator that will return a value within the specified range. The first input slot represents the beginning of the range and the second input slot represents the end of the range.

Figure 34: Pick a Random Number in a Range Block

Comparison Operator Blocks

The Comparison Operator blocks provide the operations of greater than (>), less than (<), equal to (=) for numeric values. Like the arithmetic operator blocks, each comparison block is surrounded by parentheses to ensure proper evaluation. Compound comparison statements can be constructed by nesting additional comparison operators.

Figure 35: Comparison Operator Blocks

Logical Operator Blocks

The Logical Operator blocks provide boolean evaluation expressions for the logical operators *and*, *or*, and *not*. You may note that the hexagonal slots provide the ability to use other comparative or logical operator blocks to create compound expressions.

The ability to use these other blocks in this fashion is referred to as *nesting*, where other blocks are *nested* within each hexagonal slot. When you are thinking through nested expressions, just remember that each comparative or logical operator block should be assumed to be surrounded by parentheses.

Figure 36: Logical Operator Blocks

Logical *and* will evaluate as true if both expressions being evaluated are true and will evaluate to false otherwise.

The logical *or* will evaluate as true if either of its conditions are met and will only evaluate to false if both conditions are false.

Logical *not* will invert the result of a boolean comparison, i.e., if the expression has evaluated to true a logical *not* will change the value to false and vice versa.

String Operator Blocks

The string operator blocks appear to be more useful in Scratch mode because variables in the generated Arduino code are declared as a double precision floating point number. This likely makes sense in Arduino mode because there isn't a screen to display text on the mBot.

These operations are included for completeness although they will not be used in this book.

Figure 37: String Operator Blocks

The *join* operator joins two strings together. The input slots accept strings that are type in or variables.

The *letter of* operator takes a numerical value in the first slot that represents the position in the string in the second slot. Both of these slots can take variables in addition to typed in, or literal values.

The *length of* operator returns the length of the string in its input slot. The string in the input slot can be a literal or a variable.

The *cast to string* operator takes a number as its input value and returns the number as a string. The number can be a literal or a variable.

Mod and Round Blocks

The Mod and Round Operator blocks provide the functionality for retrieving the modulo (remainder after dividing the first variable by the second) and rounding off a number.

Figure 38: Mod and Round Operator Blocks

Additional Mathematical Operations Block

The Additional Math Operations block provides the following mathematical functions:

Figure 39: Miscellaneous Mathematical Operations Block

abs The **abs** function strips away the sign and returns the value as a positive number.

Example: abs(-6) = 6

floor The **floor** function rounds down to the lowest integer for a value.

Example: floor(1.6) = 1.0

ceiling The **ceiling** function rounds up to the nearest integer for a value.

Example: ceiling(1.6) = 2.0

sqrt	The **sqrt** function returns the square root of a value as a double-precision floating point number. **Example:** sqrt(4) = 2.0
sin	The **sin** function returns the sine of a value as a double-precision, floating-point number. **Example:** sin(3) = 0.052335956242944
cos	The **cos** function returns the cosine of a value. **Example:** cos(3) = 0.998629534754574
tan	The **tan** function returns the tangent of a value. **Example:** tan(3) = 0.052407779283041
asin	The **asin** function returns the arc sine of a value. **Example:** asin(.5) = 0.523599
acos	The **acos** function returns the arc cosine of a value. **Example:** acos(.5) = 1.047198
atan	The **tan** function returns the arc tangent of a value. **Example:** atan(.5) = 0.463648
ln	The **ln** function returns the natural logarithm of a value. **Example:** ln(1.5) = 0.405465
log	The **log** function returns the logarithm of a value to the base of 10. **Example:** log(1.5) = 0.176091
e^	The **e^** function returns the value of Euler's Number (approximately 2.718) raised to the provided value. **Example:** e^3 = 20.086
10^	The **10^** function returns the value of 10 raised to the exponential value provided. **Example:** 10^3 = 1,000

Table 3: Additional Mathematical Functions

Robots Palette

The Robots palette houses all of the blocks that are associated with the various devices on the mBot (including others that may be added later). Although all of the blocks are available in Scratch Mode and Arduino Mode, you should note that in the current implementation of mBlock (3.2.2), the *when button* block can only be used if you are running the program from within mBlock (Scratch Mode). If you use this block and attempt to upload it to the mBot as part of your program you will find that the code generator does not generate any code for it.

Why does it work in Scratch Mode? In Scratch Mode, the program is running on the computer, not the mBot. The program is feeding commands to the mBot over the radio connect (WiFi or Bluetooth), and the computer operating system is capable of multi-tasking and multi-threading[15]. The micro-controller on the mBot cannot do multi-tasking or multi-threading so the code generator does not bother generating any code for it.

Let's do a quick tour of the major blocks for an ordinary mBot with no additional sensors or motors. Figure 40 is an overhead picture of the mBot. Note that the line following sensor is mounted on the bottom of the mBot is not shown.

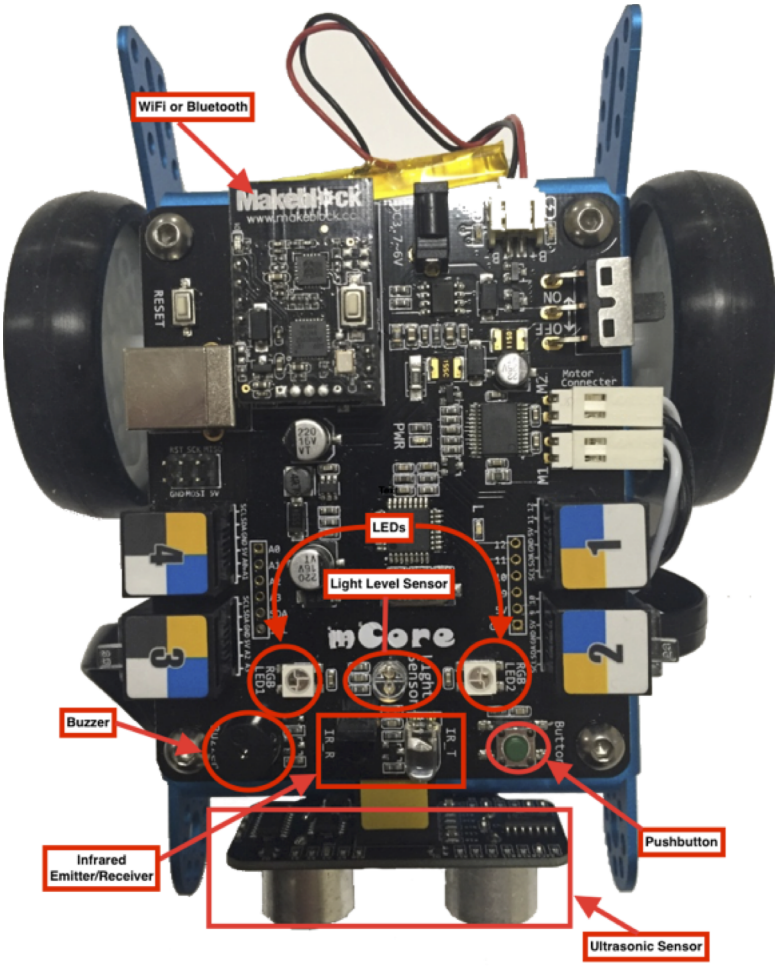


Figure 40: mBot Sensors and Devices

[15]There are advanced techniques that are possible in the native Arduino environment, but they are not directly available in the mBlock environment at this time.

Button Event Block

The *button* block compares the current state of the button on the front of the mBot with the desired state (pressed, released) and returns a true or false value. The shape of the block indicates that it can be used in control blocks, such as an *if-then* block.

Figure 41: Button Pressed Block

mBot Program Hat Block

The *mBot Program* hat block mark the beginning of a program and are required to compile and upload a program. Currently you may use multiple mBot Program hat blocks in a single program, but all of the code inside each of the hat blocks will be consolidated into a single setup() function and a single loop() function. This flexibility may cause a casual reader of the program to assume that the mBot is multi-tasking when it is not and is best avoided.

Figure 42: mBot Program Hat Block

Infrared Remote Pressed Block

The mBot comes with an infrared (IR) remote control that can be used to communicate with the mBot. The remote sends its signals to an infrared receiver on the mBot. If the mBot is running a program that uses these signals, it will respond accordingly.

Figure 43: mBot Infrared Remote Control

The *ir remote pressed* block allows the use of the infrared remote control (Figure 43) to control the mBot. The buttons R0-R9 correspond to the numeric buttons on the remote, the letters A-F correspond to the lettered buttons on the remote, the arrow keys correspond to the arrow keys on the remote and the settings button corresponds to the *gear* button in the center of the arrow keys on the remote. The block will report true or false if the designated button has been pressed. The shape of the block indicates that it can be used with control blocks.

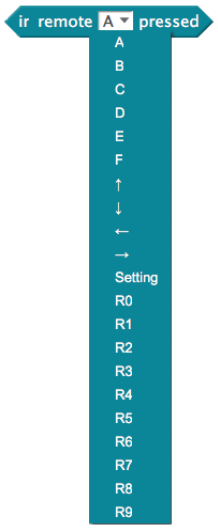


Figure 44: Infrared Remote Button Pressed Block

Set LED Block

The *set led* block enables the program to turn one or both LEDs on to set the color of the LED(s). The color codes are based on a combination of red, green, and blue (RGB) colors that provide a range of colors from black (red=0, green=0, blue=0) to white (red=255, green=255, blue=255). Note that to turn off the LED(s) you will need to set the color to black.

Figure 45: Set LED Block

The first parameter (defaults to led on board) will use the LEDs on the mCore board of the mBot. The other settings (Port 1-4) in this dropdown are used if you are controlling additional LEDs that are connected to one of the numbered ports on the mCore board. The second parameter (defaults to all) identifies which LED is being addressed. If the parameter reads all, both LEDs will receive the same color settings. On the mCore board there are only two addressable LEDs, so you should normally specify either 1 or 2.

The last three parameters specify the intensity of each color for the LEDs.

What is an LED?

LED stands for **L**ight **E**mitting **D**iode. A diode is an electronics device that allows electrical current to flow in one direction, but blocks it if the current is going the opposite direction. An LED is a type of diode that emits light when electrical current is passing through it in the right direction. The mBot has several LEDs built into its circuit board. LED1 and LED2 can be controlled from your programs if you desire.

Light Level Sensor Block

The *light sensor* block reports the light level detected by the light level sensor on top of the mCore board. The parameter may be set to the onboard light sensor or to Port 3 or 4. The default setting is for the onboard light level sensor. The light sensor returns values from 0-65535 where 0 is no light and 65535 is the maximum amount of light possible.

Figure 46: Light Sensor Block

Line Follower Sensor Block

The *line follower* block reads the values from the line follower sensor on on the bottom of the mBot chassis at the front. The two emitter/receiver pairs determine whether or not the line sensor is completely over the line, off on the left, off on the right, or completely off of the line. The sensor reports the following values:

Sensor Position	Value
Both Sensors Over Line	0
Right Sensor Off Line	1
Left Sensor Off Line	2
Both Sensors Off Line	3

Table 4: Line Follower Values

Figure 47: Line Follower Block

Motor Blocks

mBlock provides two methods for controlling the motors. You can use the Run Motors block (to control both of the motors or use the Set Motor block and specify which motor to control.

The *run* block's first parameter dropdown list offers run forward, run backward, turn left, and turn right. The motor speed can range from -255 to 255. This block will provide a tank-like turn by using a negative speed the inner motors during a turn, effectively causing the motor to run backwards.

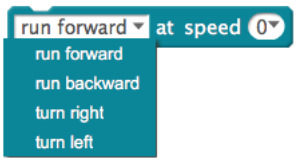


Figure 48: Run Block

The *set motor* block allows you to specify a particular motor (M1 or M2 for the mBot) and a speed for the motor that can range from -255 to 255. This block is especially useful when doing car-like steering turns where the outer wheel in the turn needs to turn faster than the inner wheel.

Figure 49: Set Motor Block

Play Tone on Note Block

The *play tone on note* block provides a method for playing a specified tone and stopping the play of that tone. Be aware that this is a *blocking* call and cannot be interrupted until its duration is over.

> In earlier versions of mBlock, the *play tone* block did not have a duration and a wait block had to be used in conjunction with a *stop tone* block to control the duration of the tone.

The available tones range from C2 to D8. These tones correspond to the same notes on a piano keyboard. In version 3.2.2 the tone can be set to play for a pre-defined length or one can specify the desired length in seconds by typing that value in beat parameter.

The pre-defined note lengths are:

Label	Length in Milliseconds
Zero	0 ms
Eighth	125 ms
Quater (should be Quarter)	250 ms
Half	500 ms
Whole	1000 ms
Double	2000 ms

Table 5: Note Lengths

Figure 50: Play Tone Block

Stop Tone Block

The *stop tone* block turns off the buzzer. The current *play tone on note* block has a duration setting which makes this block unnecessary and it will likely be removed in a future version.

Figure 51: Stop Tone Block

Ultrasonic Sensor Block

The Ultrasonic Sensor operates similar to a sonar set by sending out an inaudible ping and calculating the distance by determining the time required for the ping to return to the sensor. The corresponding block reports the distance in centimeters.

Figure 52: Ultrasonic Sensor Block

Timer Blocks

The mBot has a built-in hardware timer that measures the time in milliseconds from the time the mBot is turned on.

The *reset timer* block causes your program to store the current value of the hardware timer in an internal variable named lastTime. That value is divided by 1000 to convert milliseconds into seconds.

When you use the *timer* block to get the current time, you will get the current value of the hardware timer less the value of the lastTime variable. These values are also converted into seconds by dividing the value by 1000.

Figure 53: Timer Blocks

Commenting Your Code

Adding comments to your code enables you and others to understand what the code does and the intent behind why it was written the way it was. Many times developers fall into the trap of thinking that their code is self-documenting and therefore they should not have to comment it. While it takes practice to write good comments and to know what to comment, the value comes when someone else has to maintain your code (or when you come back to it after several months).

Without comments, the developer must spend extra time to understand why the code was written in the fashion it was. Often developers come to the conclusion that it will be faster to rewrite the code from scratch because they cannot quickly figure out what the code is doing and why it was written in that fashion. This rework, often mistakenly called "refactoring," comes at a cost in time and potential new bugs. A comment in the mBlock environment is shown below.

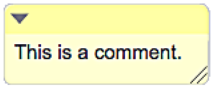


Figure 54: Sample Comment

So what constitutes a good comment? A good comment describes the section of code and the intent of the code. For example, a comment that says, "Add 1 to MyVar", does not mean much and is visually distracting because it does not communicate anything that is not obvious by looking at the code. However, a comment that says, "Add 1 to MyVar whenever this condition occurs so that a decision can be made when MyVar reaches this value" is much more meaningful.

In mBlock code, unless you are doing something that is very complex, it is suggested that you have a comment that is an overview that describes what the code does as a whole and why. Add more comments if you need to clarify a particular piece of the code.

5 | Blinking LEDs

Program Description

On almost every piece of electronic equipment you will see light emitting diodes (LEDs) that will produce one or more colors. Some act as warnings, others indicating that a switch is turned on or off, and still others may be used together to display a message. The mBot has two LEDs, marked LED1 and LED2 (see below) that we can control with a program.

Figure 55: LED1 and LED2 on the mBot

For our first program we will make the LEDs on the mBot blink. We shall have the mBot alternate between turning each LED on for 1 second and off for 1 second. This program provides an opportunity to become familiar with programming in the mBlock/Scratch environment and the process for loading and executing the program on the mBot.

Requirements

The requirements for this program are:

1. The program will alternate blinking each LED.
2. The program shall run in a continuous loop.
3. While one LED is on the other LED must be off.
4. Each LED will be turned on for one second.
5. Both LEDs shall produce a bright red color.

Analysis

Looking at the requirements, we can see that the program will have to loop and alternate turning each LED on and off in one second intervals. We can also see that the colors for the LEDs have been specified. There is not much more to look at for this set of requirements, so let's move on to design.

LED Color Settings

An LED is a kind of light and when you think of a light, you would usually think of it being on and off. From a color perspective, these two states might be black and white.

However, the LEDs on the mBot can produce many colors. Every color visible to the human eye that can be produced on the LEDs can be represented by a combination of values for red, green, and blue. If you would like to experiment with colors, there is an excellent demo program at this link[a] that will allow you to manipulate the RGB color values and see what color each combination of values produces.

[a] http://www.zebra0.com/color/index.php

Design

Rather than just hacking out the program, let's create a flowchart to describe our design.

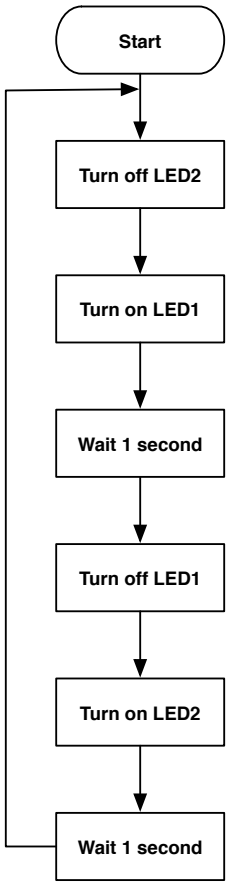


Figure 56: Blinking LEDs Flowchart

Looking at the flowchart, you might be thinking, "Why did the program start by turning off

LED2?" This is a great question and is related to our earlier discussion about performing initialization at the beginning of the program.

Turning off LED2 puts the LED into a known state (or value) and ensures that LED2 will be off when we turn on LED1. Everything else appears as one might expect, except that there is no stop block for the program.

As you can see, the line goes from the bottommost block to a point just before the first executable step. This line indicates that the program will loop back to that point after executing the last step in the program.

Program Code

To write our program, we need to launch the mBlock environment, which will differ depending on your operating system (refer to the Getting Set Up section). When you have successfully launched the mBlock environment, you should see the following on the screen:

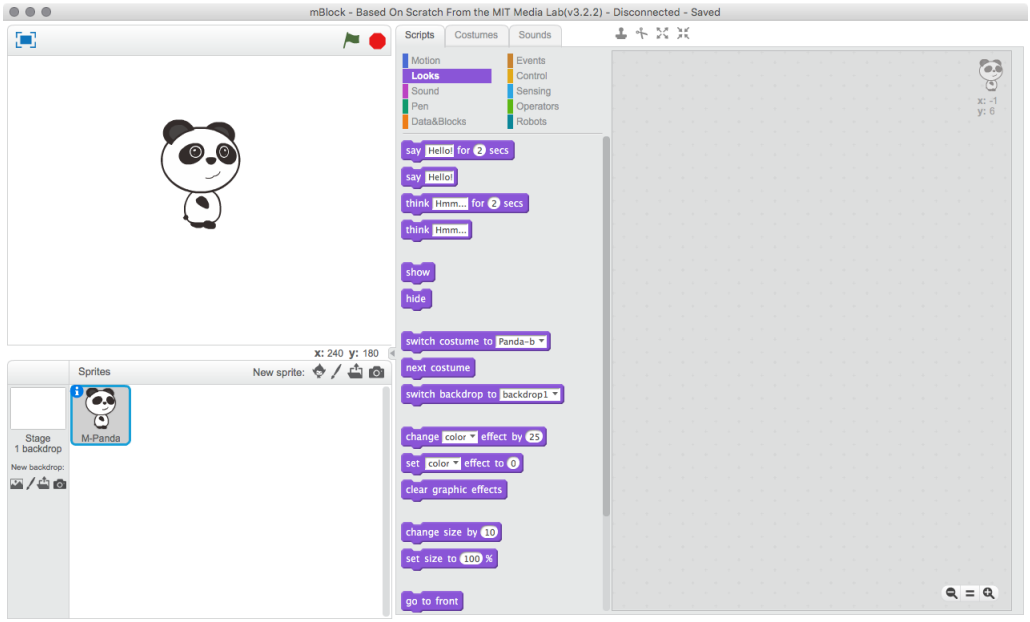


Figure 57: mBlock IDE

mBlock programs are Scratch-based, so you will need to drag blocks from the various palettes to create a program that looks like the one below. Refer to the Robots Palette in the mBlock Programming Environment chapter if you need help locating the proper blocks.

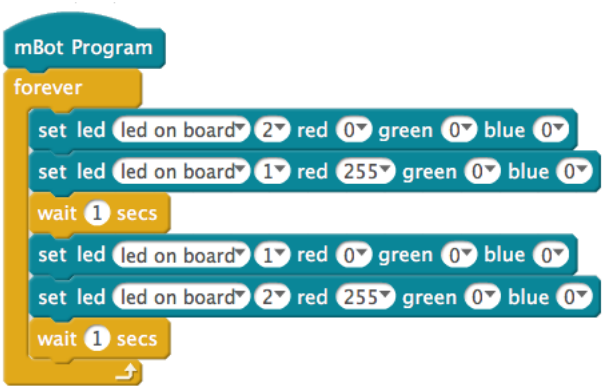


Figure 58: Blinking LEDs Program

Now save the program by clicking on the File menu and clicking the Save Project menu item. Save the program as BlinkingLEDs.sb2.

BE SURE TO FREQUENTLY SAVE YOUR CODE!!!

Deep Dive

Let's take a moment to dissect this program to understand what is going on within it:

1. The first block (mBot Program) is what the Scratch language calls a *hat block*. All Scratch programs (and therefore all mBlock programs) begin with a hat block.

2. The second block displayed says, "forever," and holds a number of other blocks inside of it. The Scratch language calls this a "C" block and it represents a looping structure. In the case of the *forever* loop, it will run the statements contained within it until you reset the default program or load another program into the mBot. Either process overwrites the memory in the mBot, effectively erasing the old program.

3. The first block in the *forever* loop is a robot-specific block that sets LED2 to the *off* setting. You may notice that we have settings for red, green, and blue on the *set led* block. Each color can contain a value from 0-255, which will enable you to control the color displayed on the selected LED (You can also select all of the LEDs at once by choosing *all* instead of a specific LED). If all of the color values are set to zero (0), the LED will turn itself off.

4. The second block inside the forever loop sets the led block to a bright red color value for LED1.

5. The third block inside the forever loop is from the control palette and causes the program to wait for one (1) second before continuing. As you may also have noticed, the color of the wait block and the forever loop block are the same, implying that both come from the same palette.

6. The last three blocks essentially duplicate the first three blocks, turning off LED1, turning on LED2, and then waiting for one (1) second. After the one (1) second wait, the program returns to the top of the loop and begins executing the instructions in the loop again.

Compiling and Loading the Program

If you click the mBot Program block, mBlock will open the *compile and load* window. Clicking the Upload to Arduino button will compile and load the program to the mCore board on the robot (see below).

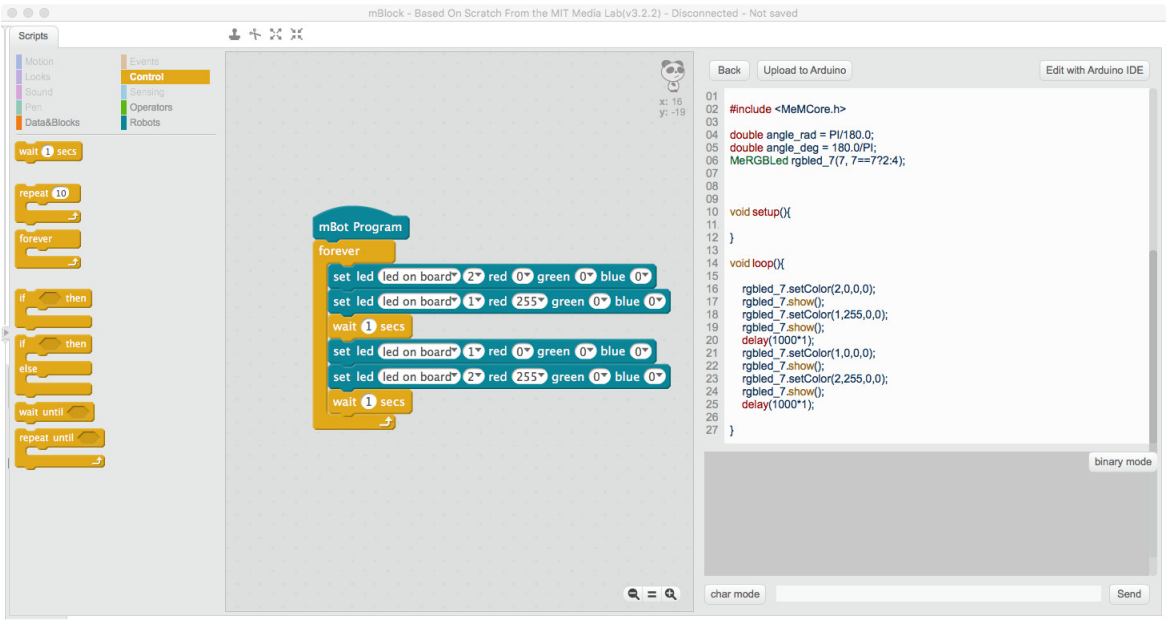


Figure 59: Ready to Compile and Load Program

After the program has been compiled and loaded into the mBot, the mBot will automatically run the program directly on the mBot. While this is not a problem for programs that do not involve any movement, it might be a good idea to think about ways to prevent the mBot from immediately executing movement after a program is uploaded to avoid your mBot running off of a table top!

It is also possible to run the program from within the mBlock IDE should you so desire. However, be warned that anything relying on timing accuracy will be affected by the time it takes to send and receive messages over the air.

Timing and Over-the-Air Delays

Everything takes time. When you run a program on the mBot, the microprocessor executes the program as fast as it can, ensuring that the mBot can respond to each statement in the program as quickly as possible. When you are running the program from the mBlock IDE, you must establish a radio connection to the mBot via 2.4GHz WiFi or Bluetooth. This connection allows mBlock to send each command to the mBot. This approach also allows the use of all of the Scratch blocks.

So why would one bother to write mBot programs that get uploaded to the mBot instead of running everything from the mBlock IDE on the computer? While the computer will certainly be many times more powerful than the mBot's microprocessor, it takes time to send commands and receive responses when using a radio connection. During this time, often called *lag time*, the microprocessor on the mBot will continue to execute the program instruction. In some cases, this may pose a problem because sensor readings that control movement and other operations may change rapidly enough that the data is out of date before it reaches the computer.

Imagine trying to drive a Mars Rover and keep it from driving over a cliff. The lag time for

a radio signal going from the Earth to Mars ranges from 4 minutes to 24 minutes[a]. As you can imagine, a lot can happen to the Mars Rover in that time! Fortunately, our lag times are much lower because the computer is much nearer the mBot. However, you will see some lag time issues when trying to achieve accurate movements or sensor readings from within the mBlock IDE.

[a] http://blogs.esa.int/mex/2012/08/05/time-delay-between-mars-and-earth/

Testing and Validation

Now we come to what is arguably one of the most important part of programming (besides designing and writing the code), testing the program to ensure that it works as we expect.

Test Procedure

1. Observe the LEDs on the mBot.
2. Is LED1 on while LED2 is off?
3. Is LED2 on while LED1 is off?
4. Is there a pause between the LEDs turning on and off?

If your mBot meets all of the criteria in your test procedure, you have correctly fulfilled all of the requirements for the program. Congratulations!

Your program has been uploaded to the mBot, so you can run it even if the mBot is not connected to the computer by the USB cable. To try this out, turn off your mBot, unplug the USB cable, and turn on the mBot. Your program should still continue to make the LEDs blink.

If you want to return the mBot to its factory shipped state, click on the Connect menu item and click the Upgrade Firmware menu item followed by clicking the Reset Default Program menu item. At this point, your mBot is completely restored to its factory settings.

Why would you want to do this? Sometimes it's necessary to ensure that all of the mBot's memory gets flushed out so that there are no traces of older programs laying around to cause problems. If your program is acting up, you can often get it to behave by performing the flush cycle mentioned above. After doing so, reload your program to the mBot and see if that straightens things out.

Troubleshooting and Debugging

But what if the lights don't blink? In this case, you will need to troubleshoot and debug so let's look at what might have potentially gone wrong (and don't worry, troubleshooting and debugging are not only normal occurrences but in fact are valuable skills to build). We'll start by narrowing down the potential points of failure into a checklist and work through each of them.

1. The mBlock IDE was not connected to the mBot and the upload failed
    a) Is the cable is connected to the mBot and the computer?
    b) Did you click the Connect menu item and choose the correct serial port?
2. The mBlock IDE cannot connect to the mBot
    a) Is the mBot turned on?
    b) Have you upgraded the firmware on the mBot from the Connect menu item?

c) Did you install the Arduino driver for the mBot?

3. There is an error in the program.

   a) Did you check that your program matches the one in Figure 58?

While it is not possible to anticipate all of the possible errors that might occur, the ones above are the most commonly encountered issues. Hopefully you will not encounter any of the connectivity issues, but if one of them occurs you will be building a skill that will help you debug your setup in the future.

Challenges

Each of the following challenges requires you to change the basic program to achieve the stated behavior. Feel free to experiment with other combinations.

1. Try to change the color of both LEDs to a different color.
2. Try to change the color of each of the LEDs to a different color.
3. Try to change the sequence of the LEDs to blink LED1 two times before blinking LED2.
4. In the Morse Code, a sequence of three short flashes, followed by a sequence of three long flashes, followed by a sequence of three short flashes sends an SOS message. Can you write a program to send a visual SOS message?

6 | Play an Octave

Program Description

The use of sound in devices is universal. Think about your personal devices, such as cell phones or gaming handsets. They all use a variety of sounds for notification or as part of a game to improve the experience. The mBot has a built-in buzzer that can play different tones, so with that in mind, let's have the mBot make some noise!

This program will play sounds using the built-in speaker on the mBot. The built-in speaker has the capability to play notes from B0 to D8 (see Figure 60[16]), a total of slightly over 6 octaves. This range allows the robot to play different tunes in response to different events if you like. It can even play simple songs!

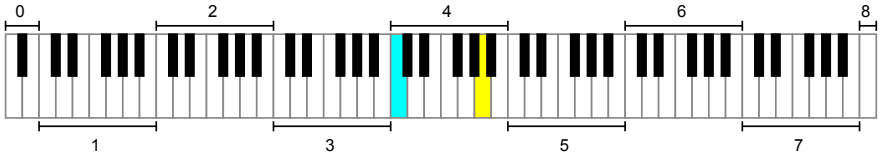


Figure 60: Piano Keyboard Octaves

Interesting Math Fact

Did you know that you could calculate the frequency of any note on a piano keyboard? The formula for the calculation is:

$$f(n) = 2^{\left(\frac{n-49}{12}\right)} \times 440\,\text{Hz}$$

Requirements

The requirements for this program are:

1. Play an eight-note scale from C4 to C5.
2. Each note shall be a quarter note.

[16] https://en.wikipedia.org/wiki/File:Piano_Frequencies.svg

Analysis

Although there is a formula for calculating frequencies for notes, we can use the pre-calculated frequencies in the table in Appendix D: Musical Notes. Why would we choose to use a pre-calculated table instead of calculating the frequency every time, you might ask. The rationale is that if the result of a particular calculation will not change, then the program should perform that calculation once and save the result to reuse because it will be faster than making the same calculation for the same result every time. As with everything there are always trade-offs. If memory is extremely limited, it might be better to do the calculation every time.

Fortunately, the team at Makeblock has pre-defined those values for you so you will not have to go through that extra work. You can verify the frequencies by going to Wikipedia[17] if you like.

Design

The most obvious approach to writing the program would be to put in a series of play tone blocks with a 0.25-second duration for each of the play blocks as shown below. This design will certainly fulfill the requirements, but it requires the implementor to pay attention to the duration for each block that plays a note.

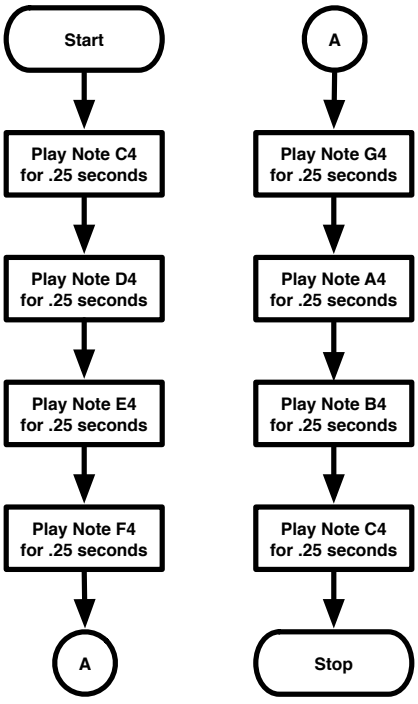


Figure 61: Play an Octave - Initial Design

Note the circles with letters in them. You may have already guessed that these circles designate a continuation of the program flow on the same page. In flowcharting terms, these are referred to as on-page connectors and they are used to allow a flowchart to be extended horizontally (or to other points in the diagram on the same page). There are also off-page connectors that are shaped like an upside-down house (pentagon) that work in the same fashion as on-page connectors except that they point to portions of the diagram that might be on a different physical page.

[17] https://en.wikipedia.org/wiki/Piano_key_frequencies

As you can likely also imagine, this approach to programming is not very flexible and would be prone to errors if the implementor is not paying close attention to the duration of the note. So how do we get around this problem? The first step would be to see where we are duplicating code and isolate that into its own custom block. So we will do that first.

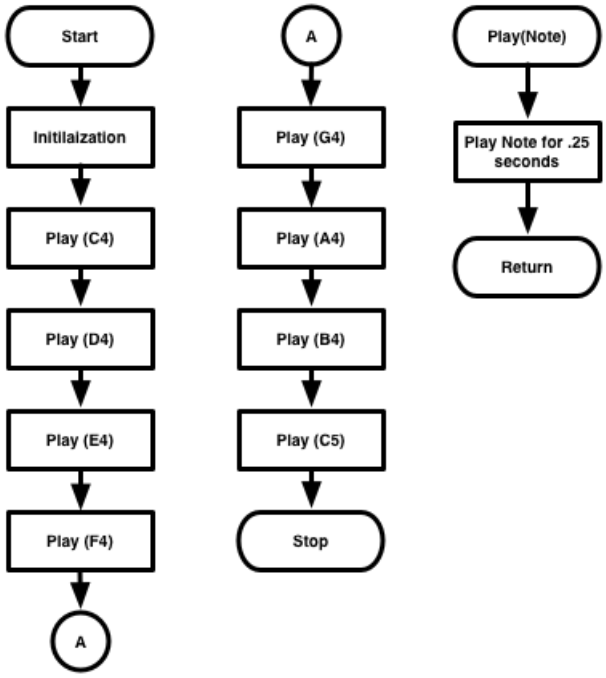


Figure 62: Play Octave - First Design Revision

As you can see, we've isolated the parts that repeat (Playing the note for .25 seconds) into a custom block (also called a function or method) named Play that takes a Note as a parameter. This step makes it possible to modify the *Play* block in the future should we want to also send in a specific duration for the note. This step will also helps improve the readability of the code by hiding the unimportant details, in this case the duration, in the custom *Play* block.

Why Not Use A List?

You might be thinking, "Hey, mBlock lets me make lists so why aren't we using a list for the notes?" This is a very reasonable thought, but unfortunately the mBlock code generator does not support lists although you can certainly use lists if you are running your program through mBlock rather than uploading it to the mBot.

The actual implementation of a list in Scratch appears to be that of a linked list although it acts like a dynamic array within mBlock. Creating and managing this data structure in the code generator would be a relatively trivial task because linked lists are well-known data structures with well-tested implementations.

So why would the code generator not support generating a list? I can't speak for the Makeblock team, but I suspect it has to do with hardware memory constraints on the mBot[a]. The mBot has 32 kilobytes of program memory (called PROGMEM) but only 2 kilobytes of variable memory (called SRAM) and the global variables used by the Makeblock libraries consume a bit under half of the available SRAM. This being the case, every byte used counts and a frequent cause of program crashes in this type of environment have to do with running out of memory.

Instead of having the risk of a continuously expanding list causing program failure, I believe the choice was made to forgo lists unless one is working in the Arduino programming environment. At that point, the developer is assumed to be responsible for memory management. With the STEM focus of the mBot, I believe that the developers chose to err on the side of caution so that younger children would not become frustrated or discouraged.

In volume 2 we will be working exclusively in the Arduino programming environment. This approach will provide you with much more flexibility in what you do and how you do it.

[a]These constraints occur because the mBot is based on the Arduino Uno which has the same hardware limitations

Program Code

One of the first things we need to do is to create our custom block, so let's do that first. To do that in mBlock, we need to click on the Data&Blocks palette followed by clicking on the Make a Block button. You should see a dialog like the one below:

Figure 63: Make a Custom Block Dialog

In the purple area, type: *Play*

Figure 64: Set Custom Block Name

Now, click the triangle by Options to show the options for the block. You'll see a dialog like the one below:

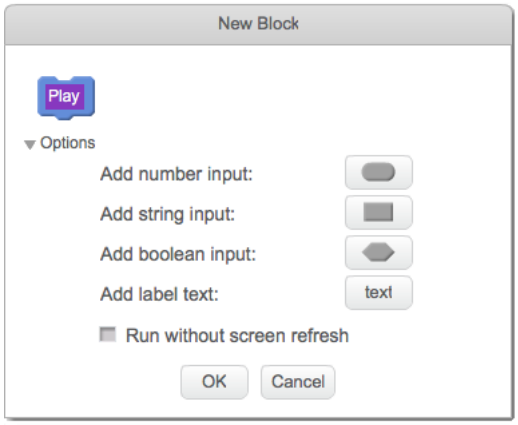


Figure 65: Make a Custom Block Options Dialog

Note that each type of input has a shape. If you inspect the shapes on the various palettes, you will see that they use the same shapes to indicate the type of input that is expected.

We need to add an input, but what type of input should we use, a number or a string? You might think that if the note is named C4 then the input should be a string. However, that is not the case. The C4 is merely the name associated with the note, not the actual note (which is a number). Add a number input and name it Note.

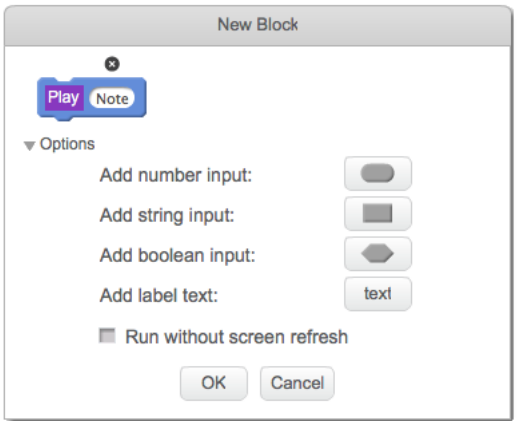


Figure 66: Play Tone Block with Note Parameter

You should see a hat block that appears on the screen that looks like the one below.

Figure 67: Custom Play Tone Hat Block

Now let's add the rest of the blocks to our custom block. First, click on the Robots palette and

drag over a block that matches Figure 68 under our custom *Play [note]* hat block until it "attaches" underneath the hat block.

Figure 68: Play Tone On Note Block

Click and drag the purple Note from the hat block and drop it into the oval on the play tone on note block that says C4. We'll also change the beat from Half to "Quater[18]".

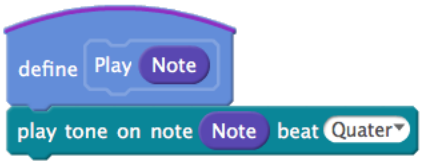


Figure 69: Play Tone Block with Quarter Note Added

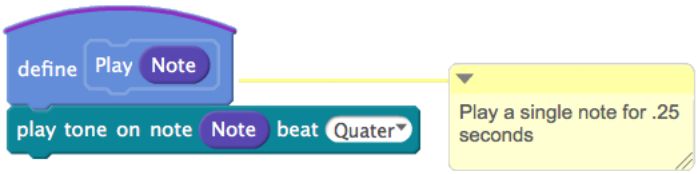


Figure 70: Final Play Tone Block with Comment

At this point, you should also see a block under the Make A Block button that you will use in your programs. It will look like Figure 71:

Figure 71: Custom Play Tone Block in Data & Blocks Palette

Drag the custom block from the palette to just under the mBot Program hat block and put the string C4 in as the parameter. You should have noticed that you cannot put a string into the custom block. Remember that we defined the input as number, not a string (note the rounded shape of the input variable on the custom block).

We can look up the numbers associated with the tones (see Appendix D: Musical Note Values) and manually add them, but then how would you know what tone was by looking at the numbers? You could add comments by right-clicking each block but that would rapidly get very messy. A better way is to assign the values to our own variables.

[18]There is currently a misspelling in the library for which a correction with the correct spelling has been requested.

Variables To The Rescue

Variables are named holding places for values. The values can be strings, numbers, or boolean (true/false, yes/no)[19]. The use of well-named variables makes your programs much easier to read and maintain.

They also make your program easier to modify because you can declare them once and use them all over your program. If you decide you need to change the value of a variable, then you only need to change it in one place, rather than trying to find the values all over your code.

Making Variables

Our application only uses eight notes, so we will create eight variables (C4, D4, E4, F4, G4, A4, B4, C5) to hold the values for the notes. To create a variable, click on the Data&Blocks palette, then click on the Make a Variable button. You will see a dialog pop up that looks like the one below:

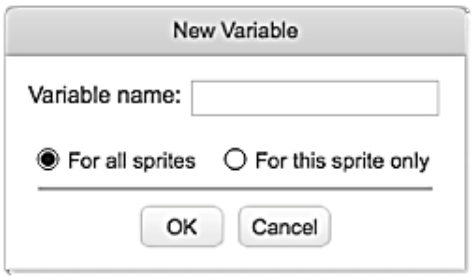


Figure 72: Make a Variable Dialog

Leave the *For all sprites* radio button set, and enter the note name C4 and click the OK button. Do this for all eight notes. When you have finished, the palette under the Make a Variable button should look like figure below:

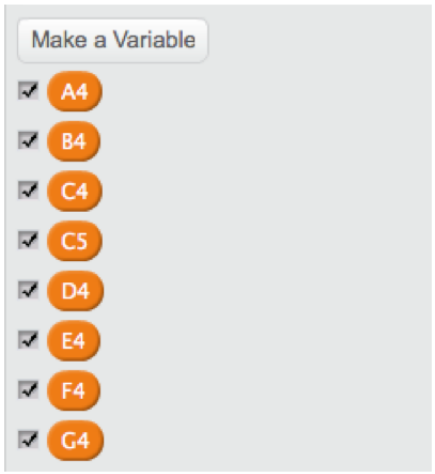


Figure 73: Data&Blocks Palette After Adding All Note Variables

[19]Currently mBlock code generation only supports variables of the double-precision floating point type.

You will also see the following blocks appear below the variables.

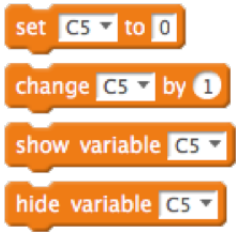


Figure 74: Variable Method Blocks

Drag the C4 variable over to the Play input slot, and now try to compile and upload the program again.

NOTE: You must drag the variable into the Play block input. If you try to type in C4, you will find that you cannot because the slot is expecting a numeric value and C4 is a string. When you declare variables you should use them rather than trying to type in the variable name because mBlock (Scratch) will automatically assign a reference to the variable. If you type in the name of the variable, you are only typing in a literal string which mBlock will use. This is a frequent source of errors for new mBlock and Scratch programmers. In this example, we have taken advantage of the type of input slot (numeric) to prevent the programmer from accidentally entering any other type of value.

In other languages, this is referred to as *static typing*. *Static typing* will allow the compiler to check that you have used the correct type of variable based on how it has been declared (numeric, string, etc.) and will display an error if you have used the wrong type of variable data.

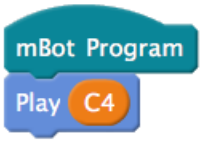


Figure 75: Play a Single Note with the Play Block

What? It did not play a tone? Why not? You have declared the variable C4, but you did not assign a value to it so it will have whatever happens to be in that piece of memory, which may or may not be a playable note.

Let's set the value for C4 to 262 and see if that makes a difference. Click on the Data & Blocks palette, and drag the set block to be directly under the mBot Program block and before the Play block:

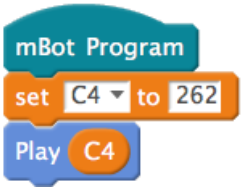


Figure 76: Setting the Value of the Note

Now compile and upload the program to the mBot. You should hear two beeps of the same tone. Why two? On loading, the mBot (due to its Arduino background) has a delay before it starts running a new program and may run part of a previous program before completing the setup loop. Subsequent runs should not experience two beeps. To try this, press the reset button towards the back of the mCore board on the mBot. This has the effect of resetting the board and running the last program. You should only hear one beep at this point.

Add the rest of the variables using the values in the table below:

Note	Value	Note	Value
C4	262	G4	392
D4	294	A4	440
E4	330	B4	494
F4	349	C5	523

Table 6: Note Values for the Play Octave Program

After you have added the variables, we are ready to write our program logic!

Program Logic

We are finally read to write our main program logic. The program shown below uses our custom play block.

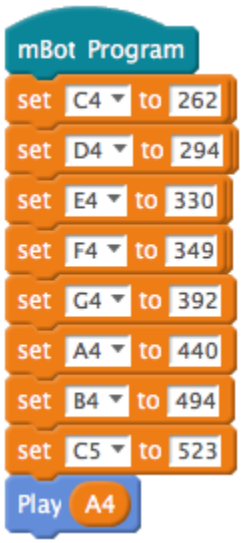


Figure 77: Play Octave Main Program Logic

Now, compile and upload and let's see if it works.

You should note that the compile did not successfully complete due to an error. Look at the compile window and you should see the message below:

```
1-27 12:52: Process exited with 1
1-27 12:52:14.790 > project\_P\_a\_Oc\_a\_\_20\_3.ino.cpp:17:8:
       error: conflicting declaration 'double A4'\\
```

It was mentioned earlier that the Makeblock libraries declare some global variables. This is an example of one such conflict with a global, read-only variable. The easy fix is to rename our note variables to use a prefix of Note_ (note the underscore after the word Note). Right click on each of the variables and add the prefix. Your program should now compile, load, and execute correctly:

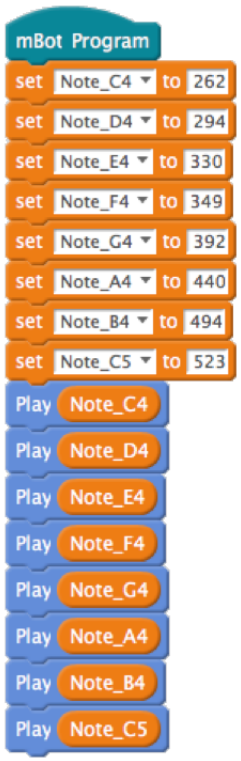


Figure 78: Corrected Play Octave Main Logic

Now that the program compiles correctly, we should think about a couple of other small items, namely readability. In the corrected program above we call a lot of initialization methods and a lot of play methods.

Let's neaten up the main logic a bit by creating a couple of other custom blocks. Create a block named Initialization (don't use setup, that's already used by the code generator) and drag all of the set methods under it.

Figure 79: Play Octave Initialization Block

Drag the Initialization code block under the mBot Program block.

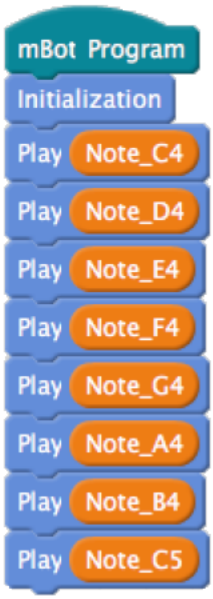


Figure 80: Play Octave Main Logic with Initialization Block

As a final cleanup for readability, create another block named PlayOctave and drag all of the play blocks under it.

Figure 81: PlayOctave Block

Drag the PlayOctave block into the mBot Program under the Initialization block.

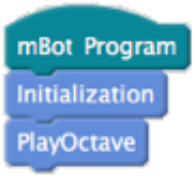


Figure 82: Final Main Logic

Keeping things DRY

The notion of isolating duplicated code into its own block (called a method or function in other languages) is referred to as the *DRY* (Don't Repeat Yourself) principal and is considered a best coding practice in all languages. Duplicated code makes it harder to modify a program because the developer must look through all of the code and hope that nothing is missed. This type of isolation makes it easier to modify programs and reduces the number of places where the developer has to look when debugging an error.

You might also want to add comments to the blocks so that it will be obvious what each one does. On a small program like this, it's optional but on larger programs you will want the added documentation to help yourself and others understand the intent of the blocks.

Although there continues to be a debate around the value of comments in source code, the author considers it a best practice in all programming languages that support comments (see Figure 83 on the next page) to ensure that the intent of the code is well-communicated.

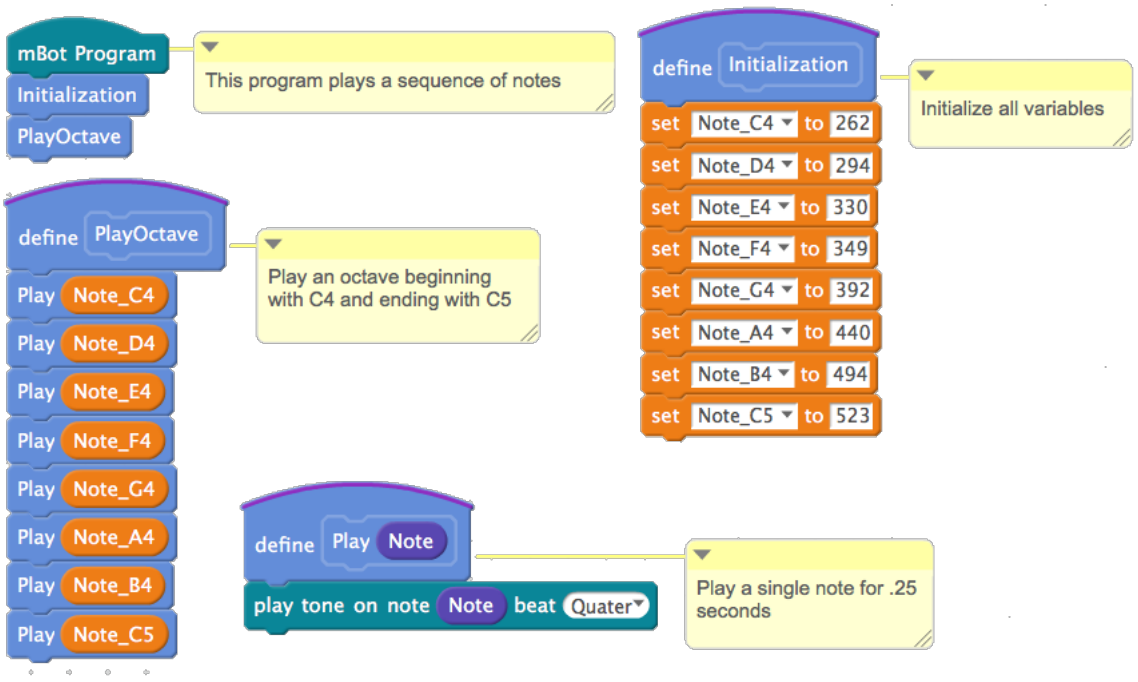


Figure 83: Final Version of Play Octave Program

Although creating the Play block is a bit contrived (because Play Tone on [Note] Beat [Quarter] could simply be inserted in the PlayOctave block), the purpose of creating the Play block was demonstrate how to create a block that takes an input parameter and to act as a starting point for one of the challenges at the end of the chapter.

Readability is also improved a bit because it is immediately obvious what PlayOctave is doing without needing to worry about the note durations. By hiding the implementation of Play Tone, we can now change the duration of the note in a single place rather than across several statements just as we only need to change the frequency of a note in a single place.

It is generally considered a best practice to minimize the number of locations where a value needs to be changed. Typically this will be accomplished through the use of variables in some kind of initialize block. This practice helps eliminate software bugs that would occur if you needed to change a value in several places in the code and missed one or more of those places.

Testing & Validation

Testing Procedure

1. Run the program
2. Do you hear the familiar 'do-re-mi-fa-sol-la-ti-do' scale?

If you hear the full eight notes, congratulations! If you do not, then you will need to troubleshoot and debug the program.

Troubleshooting and Debugging

But what if the program does not play any notes? Let's narrow down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
    a) Is the cable is connected to the mBot and the computer?
    b) Did you click the Connect menu item and choose the correct serial port?
2. The mBlock IDE cannot connect to the mBot
    a) Is the mBot turned on?
    b) Have you upgraded the firmware on the mBot from the Connect menu item?
    c) Did you install the Arduino driver for the mBot?
3. There is an error in the program.
    a) Did you create the variables correctly?
    b) Did you initialize the variables?
    c) Did you check that your program matches the one in Figure 83?

Challenges

Each of the following challenges requires you to change the basic program to achieve the stated behavior. Feel free to experiment with other combinations.

1. Modify the Play block to take a duration so that you can control the note played and the duration.
2. Work out the notes for a simple melody and modify the program to play the melody.
3. Modify the program to flash the LEDs when a note is played. You can be as creative with this as you like.

7 | Press the Button

Program Description

So far we have looked at LEDs and making sounds, but none of this involves any user interaction with the mBot. If you look at the front part of the mBot circuit board, you will see a push button. This type of push button is a normally-open momentary contact button and is often used to signal the mBot to do something.

Figure 84: mBot Push Button

Electronic Switches

A switch is used to control the flow of electricity through a circuit. When a switch is closed, or in the *on* position, electricity can flow through the circuit. When the switch is in the open, or *off* position, the flow of electricity is interrupted. There are many types of switches, such as the ones on your wall that control the lights or the tiny pushbuttons on the mBot.

The mBot's green pushbutton is referred to as being *normally-open* due to the default position for the switch being open, or off. It is also a *momentary contact* switch because the switch is only closed, or on, while your finger holds it down. Electricity can only flow through the circuit when the button is pressed.

There are also normally-closed momentary contact buttons that do the opposite, i.e., the circuit is closed with electricity flowing through it until the button is pressed, interrupting the flow of electricity through the circuit. Both types of buttons have their uses, but it is important to know which type you are using!

The goal of this program is to wait until a the button is pressed and then to play a tune. We can reuse all of the Play Octave program and make a few minor changes. You can save a lot of time by recognizing what pieces of earlier programs are reusable. This practice will enable you to reuse your analysis and designs as well as your code. Ultimately developers strive to make as much of their work as reusable as possible for exactly this reason.

Requirements

The requirements for this program are:

1. The program should wait for the push button to be pressed.
2. When the button is pressed, play the following sequence of notes:

   E4 D4 C4 D4 E4 E4 E4 R D4 D4 D4 R E4 G4 G4

   The **R** represents a rest, or pause, in the melody.

Analysis

Having already written a program that plays a list of notes, the only differences we see are the addition of the "R" to indicate a rest, or pause, and the need to wait for the push button to be pressed to start playing the melody. So let's move on to design.

Design

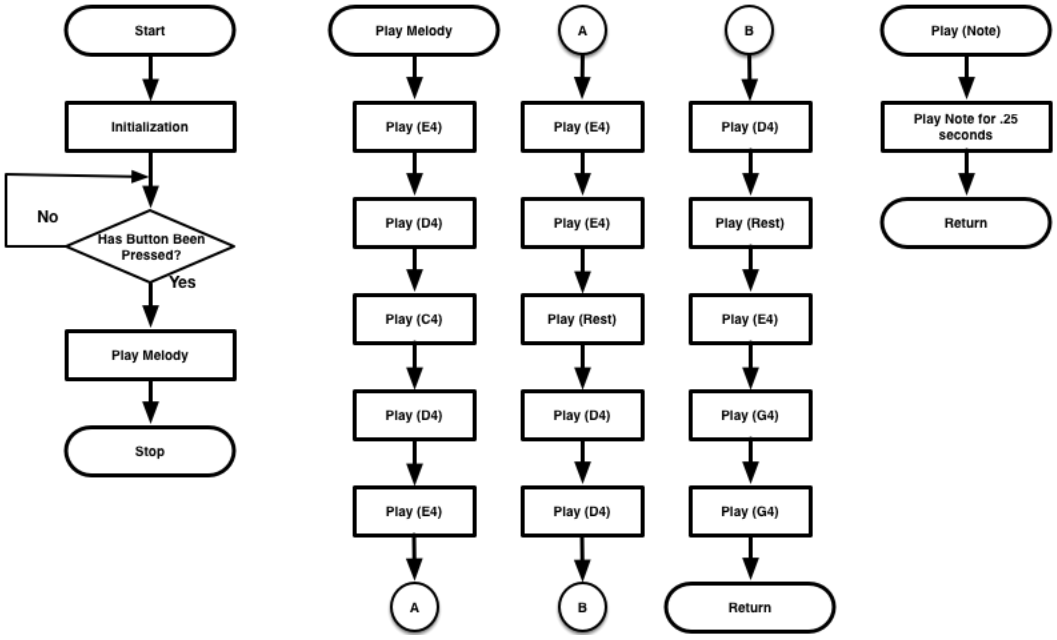


Figure 85: Press the Button Flowchart

Although a different sequence of notes are used, this program design is essentially the same as the Play Octave program from the last chapter. A loop has been inserted that will wait until the button on the front of the mBot is pressed. Like the Play Octave program, the work of playing a note and the melody are isolated into their own custom blocks.

As you may have seen, the decision block that follows the Initialization block has two arrows, one indicating the path if the decision is no and another one that indicates the path if the decision is yes. The circular objects, are on-page connectors that enable us to keep the diagram concise.

Program Code

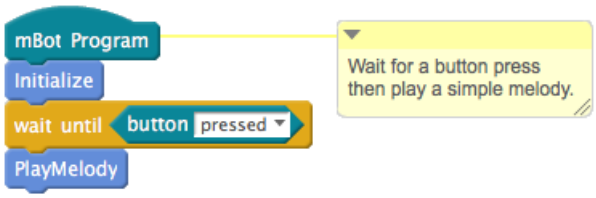


Figure 86: Press the Button Main Logic

The main logic differs from the Play Octave program with the addition of the wait until block that uses the button reporter block to determine whether or not the button has been pressed. It is a quite common practice to evolve an existing program's capabilities through the addition of small bits of code.

> Agile process[a] practitioners generally structure their work to introduce small changes and use automated testing to ensure that the new functionality works as expected and to verify that the additions have not caused existing functionality to stop working.
>
> [a] https://en.wikipedia.org/wiki/Agile_software_development

The Play block accepts a value and determines whether or not the value is note (greater than zero) or a rest (equal to zero). The *if-then* statement controls whether a note is played or if a wait equal to the duration of a quarter note occurs.

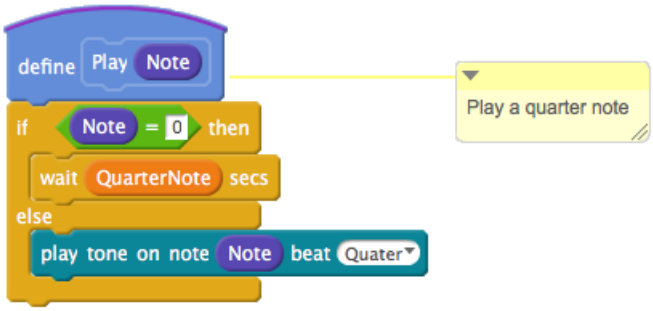


Figure 87: Press the Button Play Block

The other section of the code that might be of interest is the Initialization block. As you can see in the figure below, we don't need to define many notes for the melody. However, we are assigning a value of zero to the Rest variable so that no note will be played.

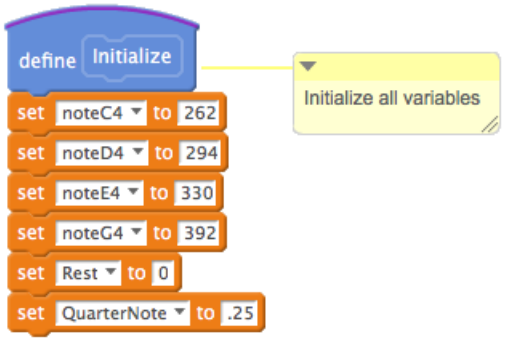


Figure 88: Press the Button Initialization Block

Final Program

Below is a solution for the Press the Button program. Although the code is pretty well self-documenting, adding the comments is more about building the habit of commenting your code.

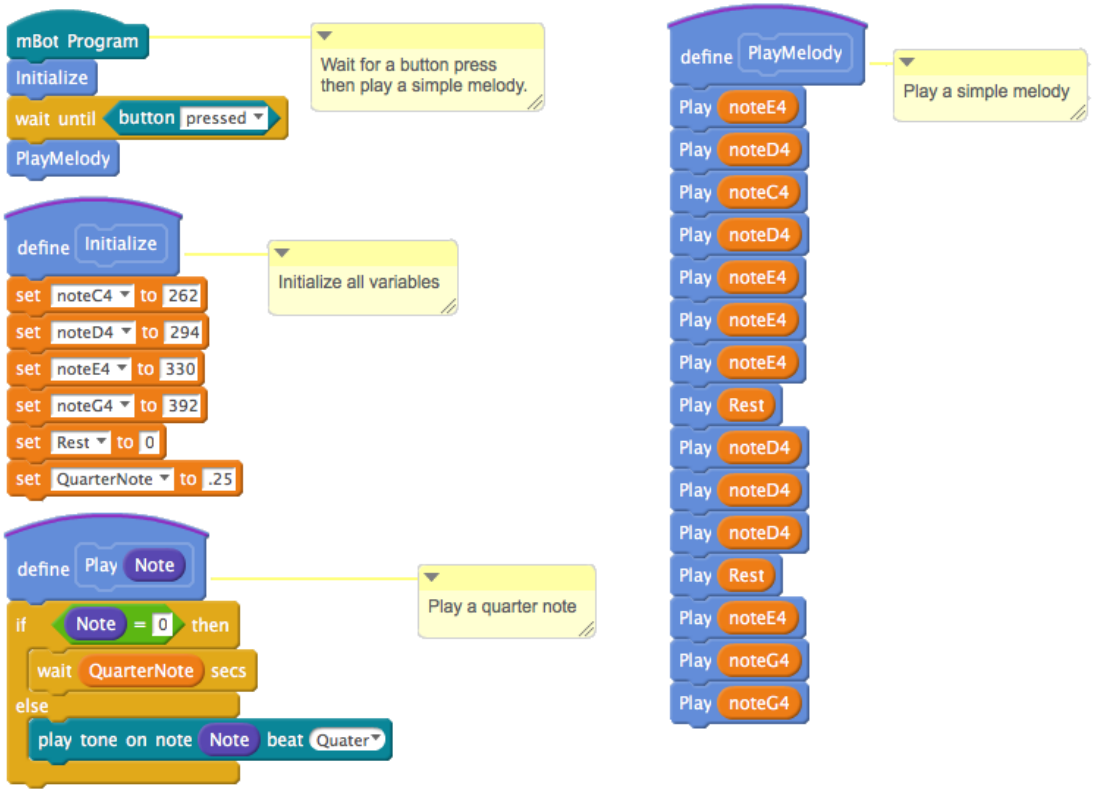


Figure 89: Press the Button Final Program

Testing & Validation

Testing Procedure

1. Press the green button on the robot and you should hear a familiar melody.
2. Does the mBot wait to play the melody until the button is pressed?
3. Does the melody play after the button is pressed?

If your program fulfills all of the test conditions, it has passed. Congratulations!

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
3. There is an error in the program.
   (a) Did you create the list correctly?
   (b) Did you create the variables correctly?
   (c) Did you check that your program matches the one in Figure 89?

Challenges

Try to modify the program to make it do the following:

1. Make the program continue (loop) to play the melody after the button is pressed.
2. Make the program continuously play the melody after the first button press and stop at the next button press.

8 | Light It Up!

Program Description

Have you ever had the power go out at your home? Some people have blackout alarms that make a sound when power is lost, so let's write a program that has the mBot do the same thing.

The mBot has a light level sensor on the top of the circuit board that can be used to measure the level of available light around the robot. This sensor can also be used to trigger the robot to perform an action when the light rises above or falls below a user-specified level.

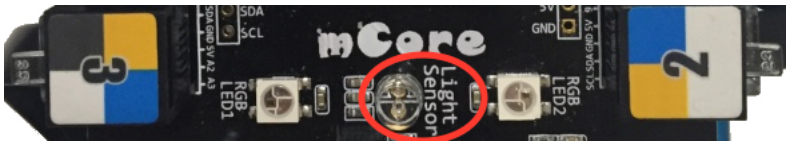


Figure 90: mBot Light Level Sensor

Requirements

This program will act as a black-out alarm. The program shall:

1. Measure the available light.
2. Make a sound when the current light level drops 20% below the normal light level.
3. Turn off the sound when the light level rises above the low-light threshold.

Analysis

The light sensor measures light and returns a value in the range of 0 - 1024. Light levels vary, so the program will need to take an initial measurement to determine the light level that will be used.

Due to the normal variances in light level measurements, it is a good practice to take several samples and then use the arithmetic mean. For example, if the mean of the readings is 480 the deviation for what triggers the speaker to turn on might be -10% of the mean, i.e., a measurement of 432.

At the rate that the light sensor updates its values, it is likely a good idea to measure the light intensity at a designated interval, such as 1 second.

Design

The design of the program accounts for the need to collect light level samples and determine the arithmetic mean for a "standard" light level by using a loop at the beginning of the program. All values are added to the Accumulator and then divided by 10 to calculate the arithmetic mean. From this point the light threshold is calculated and the program enters an endless loop where it reads the current light intensity and compares it against the threshold value to determine whether or not to make a sound.

You will note that there is no Stop block in the program because the results of the final If-Then blocks are to route the program back up to read the light sensor again in a 'forever' loop that you must break out of manually by turning off the robot or resetting the default program.

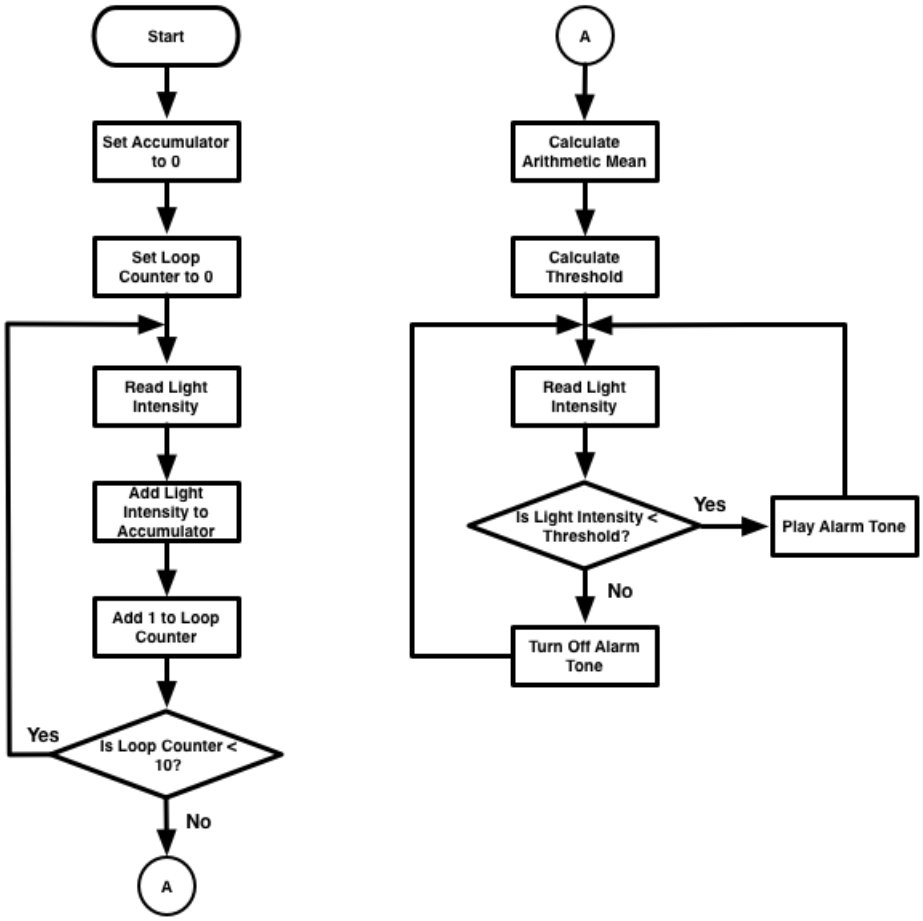


Figure 91: Light It Up! Flowchart

What Goes Where?

When the mBlock code generator generates the Arduino code, anything within the forever block goes into the *loop()* statement and will run forever. Anything that does not appear within a "forever" loop will appear in the *setup()* method.

Program Code

As we look at the flowchart, we can see that we have a bit to do in the Initialization block to get all of our measurements gathered and set up the program variables. We set the threshold to 80% (100% - 20%) of the arithmetic mean that represents the "normal" light level. Should the requirement change from 20% to some other number, we can change it in a single spot where the calculation occurs. You could also set up a dedicated variable for the percentage should you desire to make the program more self-documenting.

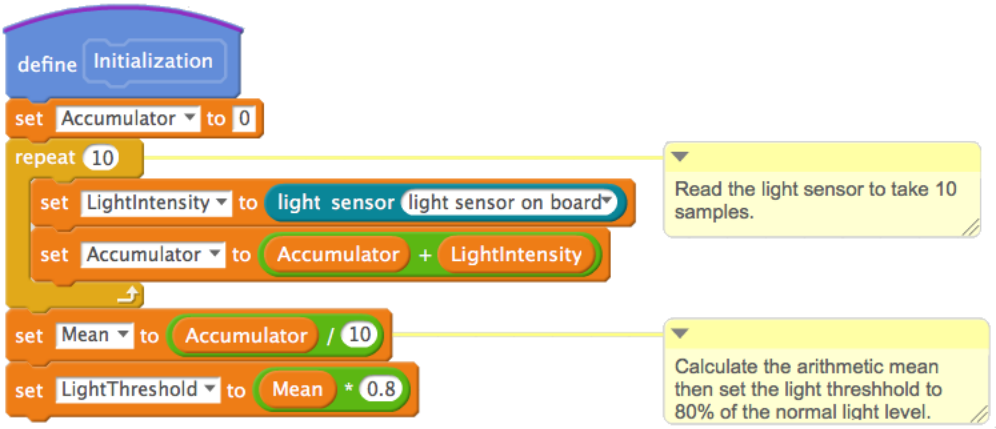


Figure 92: Light It Up! Initialization block

Let's move on to the heart of the program, the CheckForBlackout custom block. First we set the LightIntensity variable to the current reading from the light sensor on the mBot. The comparison of that value to the light threshold we calculated in the Initialization block determines whether or not we have the mBot make a sound.

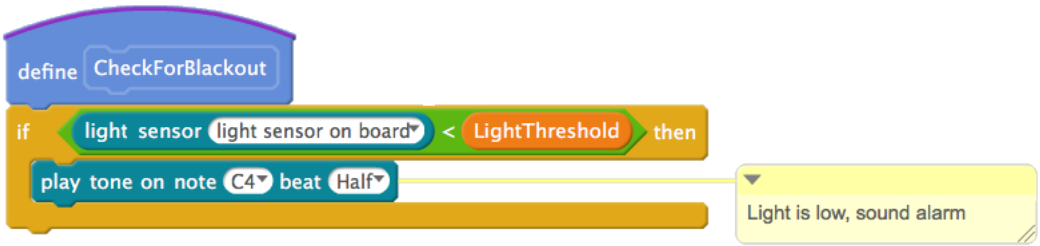


Figure 93: Light It Up! CheckForBlackout block

Could all of this code be in the main program rather than in custom blocks? Of course, but by isolating the code into a couple of custom blocks we can make the main program code very short, but extremely readable. This quality of readability is one that you should strive for in all of your programs. It will ensure that you or anyone else looking at your program will grasp what is being done, especially when you come back to your program several weeks or months later.

As you can see, our main program is extremely short, but it is immediately obvious what the program does.

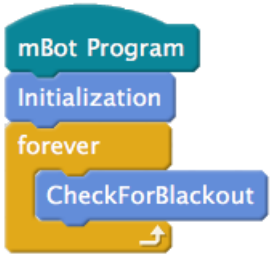


Figure 94: Light It Up! Main Program

Elegant Simplicity

When I was a junior programmer one of my mentors constantly urged us to write code that was elegantly simple. His definition of elegant simplicity centered on breaking the problem into smaller chunks of code that did one thing well and then stringing those chunks together with a small "main" program. This approach enabled us to solve each problem and test it before moving on to the next one, knowing that the foundation we were building on was solid.

In retrospect, this was a predecessor to practices currently associated with the Agile development process. You will develop your own development style over time, but I would urge you to keep the idea of "elegant simplicity" in mind as you do so.

Testing & Validation

You will need to test your program with several different light levels to determine whether or not the program is working as intended.

Testing Procedure

1. In a brightly lit room, turn on the mBot.
2. Cover the light sensor with your hand or something else that will drop the light level.
3. In a less brightly lit room, repeat the procedure above.

If the mBot makes a sound when the light levels drop, you have succeeded in making a blackout alarm. Congratulations!

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBlock IDE cannot connect to the mBot

(a) Is the mBot turned on?

(b) Have you upgraded the firmware on the mBot from the Connect menu item?

(c) Did you install the Arduino driver for the mBot?

3. There is an error in the program.

   (a) Did you create the variables correctly?

   (b) Did you initialize the Counter variable to zero?

   (c) Did you use the proper comparison operator?

   (d) Did you check that your program matches the ones in Figures 92-94?

Challenge

Modify the program to make it react to a 10% increase in the light level from the *normal* light level with a different tone than the one used for a decrease in light level.

Things to Ponder

1. If you modified the program to turn on the LEDs when the light level was low what would you expect to happen? How would you work around what happens?

2. Do you think you would get a different effect if the color of the light was different than white?

9 | Ultrasonic Theremin

Program Description

The mBot has an ultrasonic sensor mounted on the front that looks like *eyes*. This sensor operates by sending out an ultrasonic signal from one of the *eyes* and receiving the reflection of the signal in the other *eye*. This signal is too high for humans to hear, but some animals, such as dogs, have hearing that is sensitive enough to hear some ultrasonic signals. When in operation, the ultrasonic sensor operates in the same fashion as the sonar (*SO*und *NA*vigation and *R*anging) on a submarine or surface vessel.

Figure 95: mBot Ultrasonic Sensor

Generally speaking, ultrasonic sensors are used for a number of different purposes, such as range finding, navigation, etc., but we shall create our own musical instrument, called a *theremin*, to experiment with the ultrasonic sensor. A theremin is an electronic musical instrument that plays different notes based on the proximity of a hand or other object. We will use the ultrasonic sensor to detect the range in front and play notes based on the distance and change them based on a how close the object is to the ultrasonic sensor.

Requirements

Our ultrasonic theremin has the following requirements:

1. The robot shall begin playing a tone when it detects something within a range of 15-50cm.
2. The robot shall stop playing a tone if the object is less than 15 centimeters away or more than 50 centimeters away.
3. The robot shall play a maximum of one octave (8 notes, C D E F G A B C) based on an equal spacing over the 15-50cm range.

4. The highest note should be played at 15 cm.
5. The lowest note should be played at 50 cm.

Analysis

Having already discussed how the ultrasonic sensor works, we can focus on solving the problems associated with creating the theremin. The first item is to determine the physical spacing between note changes so that we can set up tests to change notes according to the requirements.

From that point, we need to know when to turn off the buzzer on the robot when the object is outside of the specified range of distances. The ultrasonic sensor measures distances in centimeters so if you want to use inches, you'll have to do a conversion by multiplying the number of centimeters by 0.3937 to get the number of inches. Likewise, to convert from inches to centimeters requires that you multiply inches by 2.54 to get the number of centimeters.

The first task will be to determine the interval values that we will use to change our note values. Due to the way the ultrasonic sensor reads data, we can use 15cm as our lower bound to trigger the highest note but we will need to add a cushion of 6-10cm to our upper bound to ensure that the lowest note gets played. Otherwise, the speaker will turn off immediately after it hits anything above the threshold number. Finding a good cushion is usually accomplished by adding one and a half additional steps (58), but often some experimentation is required to get a consistently good number.

Our next challenge will be to calculate the index into a list of notes. There is no truncation block in mBlock (or Scratch), so we will need to do this in a different fashion using the modulo operator. The modulus of a division contains the remainder and the modulo operator returns the remainder of a division. We will use the following steps to calculate the index:

1. Subtract the minimum value from the range to scale the range down to the values we are interested in.
2. Use the mod operator to get the remainder from dividing the range by the number of intervals.
3. Subtract the remainder from the range.
4. Perform the division to get the index.

The actual formulas would look like:

$$\text{Normalized Range} = \text{Current Range} - \text{Minimum Range}$$

$$\text{Note Index} = (\text{Normalized Range} - (\text{Normalized Range mod Interval})) / \text{Interval}$$

So let's see how all of this math works out in a couple of examples. First off, we know that our range is from 15-50cm and that we are working in 5cm intervals. Let's also assume that the ultrasonic sensor returns a value of 15.72cm.

Our first step will be to normalize the range by subtracting the low end of the range from the high end.

$$50 \text{ cm} - 15 \text{ cm} = 35 \text{ cm}$$

Now that we have a normalized range, we need to account for any fractional values because the ultrasonic sensor returns a measurement with decimals. We can use the mod operator to determine what the decimal portion is and subtract it out.

$$15.72 - (15.72 \bmod 5) = 15.72 - 0.72 = 15$$

At this point, we only need to divide the decimal-less range by the interval to get a number representing the note based on the range.

$$15 / 5 = 3$$

Isn't mathematics awesome? Because we only perform our calculation on numbers that we know are in the range we are looking for, we can be fairly certain that we will get a valid index number from our calculation.

Design

The flowchart below illustrates the principal operations that need to occur in the program.

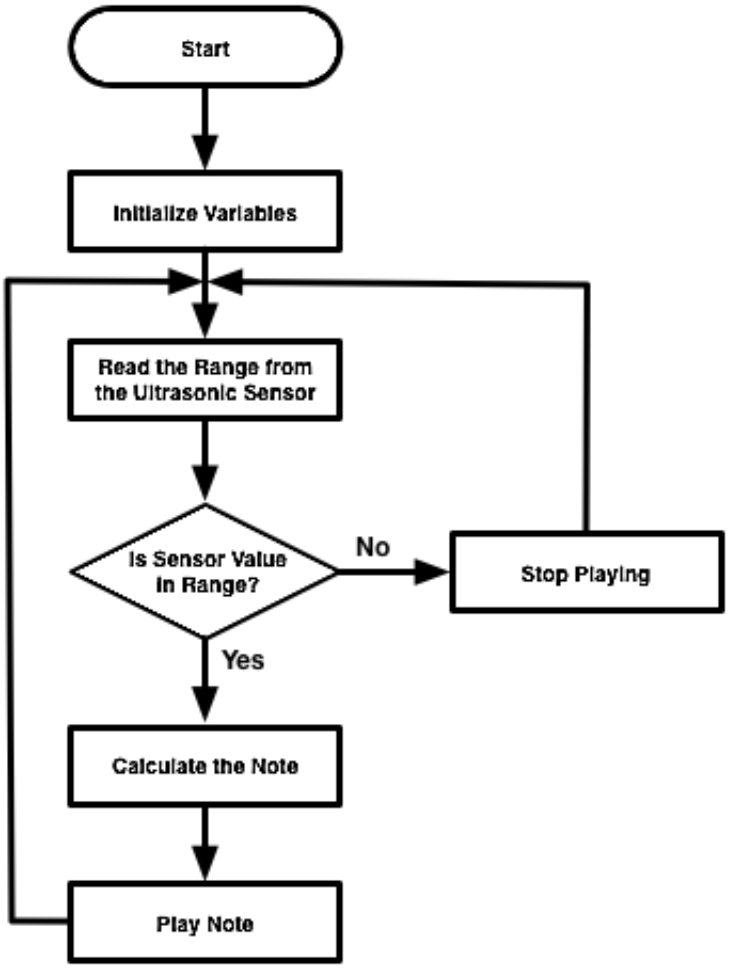


Figure 96: Ultrasonic Theremin Flowchart

You should note that there is not a Stop terminal symbol in the flowchart because we want the program to continue looping. This is the same technique that we used in the Chapter 5: Blinking LEDs and the Chapter 8: Light It Up! programs.

Program Code

We will look at the code section by section to ensure that the code and reasoning behind the code is well understood. The first section is the initialization block which contains the familiar task of setting up variables with values that represent the notes of the octave. These blocks are followed by a set of blocks that sets the minimum range (MinRange) and the maximum range, *MaxRange* as well as the length of the measurement intervals, *Interval*.

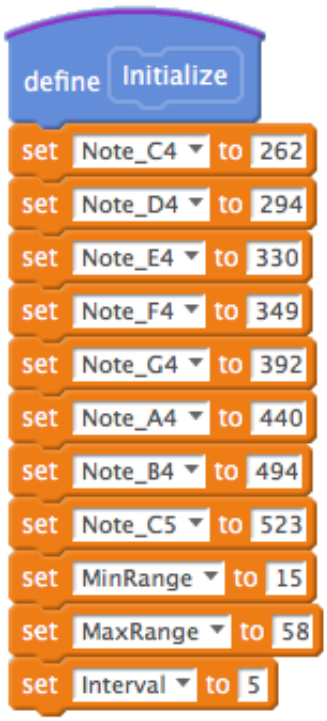


Figure 97: Ultrasonic Theremin Initialization Block

The Play block is relatively straightforward as well, although you should note that we initialize the Tone to zero at the beginning of the block to ensure that it has a known value that we can test against at the bottom of the block. The *if-then* statement merely tests to see whether or not a valid tone has been selected and plays it if one has.

Figure 98: Ultrasonic Theremin Play Block

An alternate implementation would be to simply play the selected note in each of the *if-then* tests rather than just assign a note to be played at the end. If you choose to implement in this fashion you can eliminate the Tone variable, its set block at the top of the block, and the *if-then* test at the end of the block.

Either approach works and as a general rule is more of a matter of personal taste (unless you are tight for variable space at which point you should use the alternate approach to avoid having to create an extra variable). Finally we arrive at the main program logic.

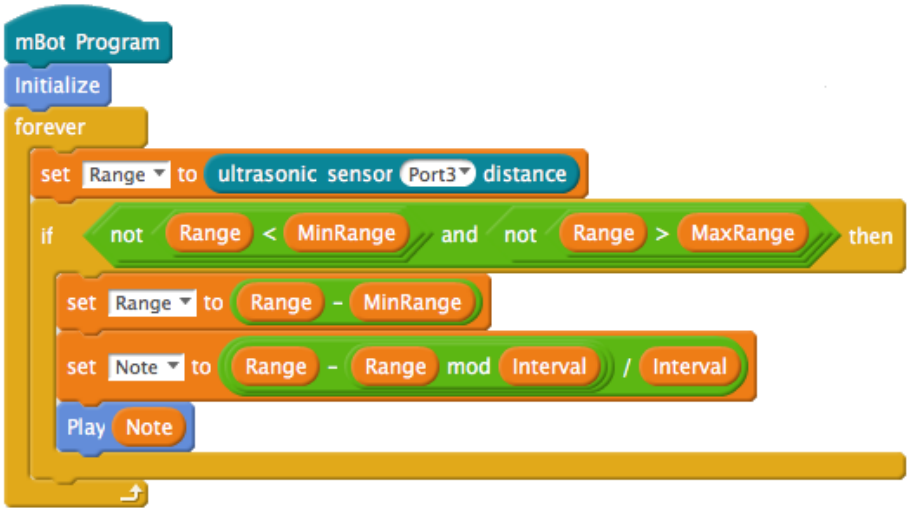


Figure 99: Ultrasonic Theremin Main Program Logic

The value of the ultrasonic sensor is read into the variable *Range* which is then tested to see if it falls within the legal values we have defined for the minimum and maximum range.

It is necessary store the value in the range variable because we are making two comparisons against the same value. If we were only making a single comparison we would not necessarily need the Range variable.

The *if-then* test has a number of embedded logical functions that may not be easily read from Figure 99. Keeping in mind that each logical operator may be assumed to be surrounded by parentheses, the test condition would read:

$$\text{if } ((\text{not}(\text{Range} < \text{MinRange})) \text{ and } (\text{not}(\text{Range} > \text{MaxRange}))) \text{ then}$$

Nested Operators

When constructing nested operators, it may be easier to work from the inside out, i.e., do the innermost operations, then drag them into containing operations. To illustrate:

1. Create the expression (Range < MinRange)
2. Drag this expression into its own *not* block
3. Drag the *not* block into the left-hand side of an *and* block.
4. Create the expression (Range > MaxRange)
5. Drag this expression into its own *not* block
6. Drag the *not* block into the right-hand side of an *and* block.
7. Drag the *and* block into the *if-then* statement.

Having determined that the Range variable is inside of our specified continuum of values, it is time to perform the scaling operation followed by determining which note needs to be played. The final step in the *if-then* test is to actually play the note with our custom block.

Testing & Validation

It is now time to run the program and see whether or not it performs as expected. It is recommended that you have a tape measure or some other measuring device to determine whether or not the notes are being played at the expected distances from the ultrasonic sensor.

It is also recommended that you use a flat surface (called a reflector), such as a book, to ensure that the sensor has no issues with ultrasonic signals being absorbed. The flat surface will also let you see where on the tape measure the notes are changing.

Test Procedure

1. Test every 5cm beginning at 15cm in front of the mBot to see if a tone is played.
2. Test that no note is played if the reflector is below the minimum range.
3. Test that no note is played if the reflector is above the maximum range.

If your program does all of these, you have successfully the assignment. Congratulations!

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
3. There is an error in the program.
   (a) Did you create the variables correctly?
   (b) Did you initialize all of the variables?
   (c) Did you use the proper comparison operator?
   (d) Did you select the correct port for the Ultrasonic Sensor?
   (e) Did you check that your program matches the one in Figures 97 - 99?

Challenges

Modify the program to make it do the following:

1. Change the interval and the min/max values to find a length of *scale* that works best for you.
2. Find the frequencies for the accidentals (sharps/flats) and add them for more complex melodies.
3. Use the LEDs to change colors with each different note.

Things to Ponder

The program demonstrates one method of creating an electronic musical instrument. Are there other applications that you can think of that would use a change in sound based on the distance from an object?

10 | Baby You Can Drive My Bot

Program Description

You have likely seen or heard about robotic competitions where the participants drive their *robots* via a remote control. As mentioned in the preface, tele-operated vehicles do meet the definition of a robot, i.e., they are not autonomous. The real value of a robot is realized when it can operate autonomously without the need for human supervision. It is, however, quite common to use a remote control to experiment with movement and in some cases to *train* a robot to be able to make movements using some sort of operator harness.

This will be our first program to make the mBot move using the remote control to make it travel around based on the buttons we push on the remote. We will keep it simple and just plan on moving forward, backward, and turn left or right.

Requirements

The requirements for the program are:

1. The mBot shall move forward when the *up* arrow on the remote is pressed.
2. The mBot shall move backward when the *down* arrow on the remote is pressed.
3. The mBot shall turn left when the *left* arrow on the remote is pressed.
4. The mBot shall turn right when the *right* arrow on the remote is pressed.
5. The mBot shall stop when none of the arrow buttons are pressed.
6. The mBot shall not respond to any button presses that are not listed above.

Analysis

The first topic that needs to be understood is how robots move and turn. As you can probably guess with a two-wheeled robot, you will need to ensure that both wheels are turning in the same direction to go forward or backward.

Where things become interesting is when you want to have the robot turn. There are two types of turns that the robot can make and each will have a different effect on how the robot turns and by how much the robot will turn.

The first type of turn is often called “steering” because it causes the robot to turn in the fashion as an automobile. To accomplish this type of turn, one wheel must turn and the other wheel must not turn. This will have the effect of turning the robot in an arc based on the wheel that is not turning. In the case of an automobile and other vehicles, what really happens is that the “inner” wheel turns more slowly than the “outer” wheel.

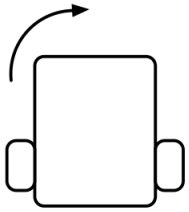


Figure 100: Automobile Steering Turning

The other type of turn operates in the same manner as a tracked vehicle or tank. One wheel will turn one way and the other will turn the opposite direction. This type of movement will spin the robot in place.

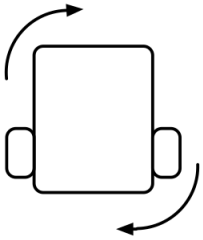


Figure 101: Tank Turning

Each type of movement has its place, and it depends on what you are attempting to accomplish.

Another factor that comes into play when making turns is the speed of the turn. Turns made at a slower speed will be more accurate than those made at a higher speed. The speed issue holds true for forward and backward movement as well.

Another factor that comes into play involve the charge level of the battery (or batteries if you are using the battery pack). Fully charged batteries will make all movement involving motors much faster than movements with a partially discharged battery (or batteries in the case of the battery pack). If your mBot seems to be performing erratically, it might be time to change (or charge) the batteries.

You also need to be aware that all remotes use the same frequency to communicate with the mBot. If you are in a classroom setting, only one mBot at a time can be operating. Otherwise, the mBot will happily respond to any other remote signal!

The mBlock environment has preset blocks for moving and turning based on the arrow keys that we will use for this program. However, in later programs it will be necessary to look more closely at ways to move the robot, especially when turning.

Design

The flowchart below depicts the logic the program will use to accomplish the task of responding to button presses on the remote.

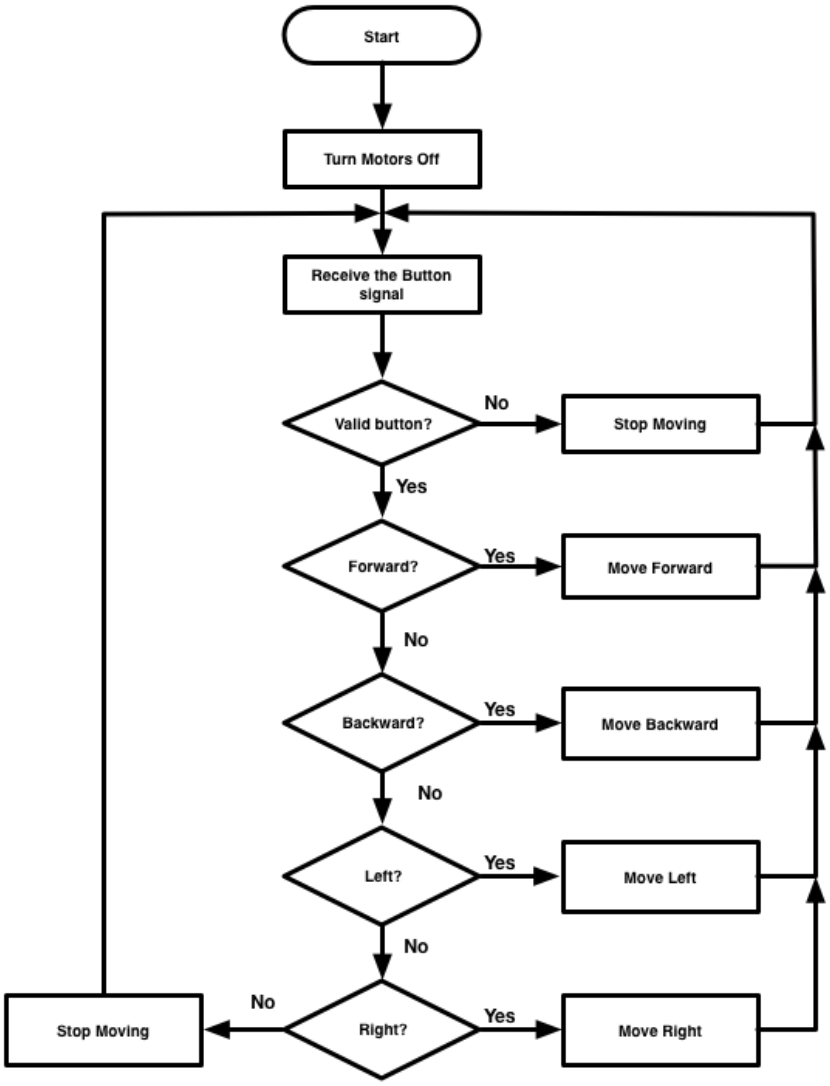


Figure 102: Baby You Can Drive My Bot

Let's take a moment to walk through the flowchart to ensure that everything happening is well understood. The program starts by turning off all of the motors to ensure that the robot starts in a known state. From there the program goes into an endless loop waiting for button presses from the remote.

If a button press is received, the program tests to see if it is a valid movement button. This is an example of using *guard code* to filter out unwanted inputs by only accepting valid inputs. If the button pressed is one of the buttons we accept for movement, the program will determine which movement to make based on the button and will turn on the motors to make the movement. If the button is not a valid movement button, the program will make the robot stop by turning off the motors.

Program Code

The following code (Figure 103) implements the logic described in the flowchart.

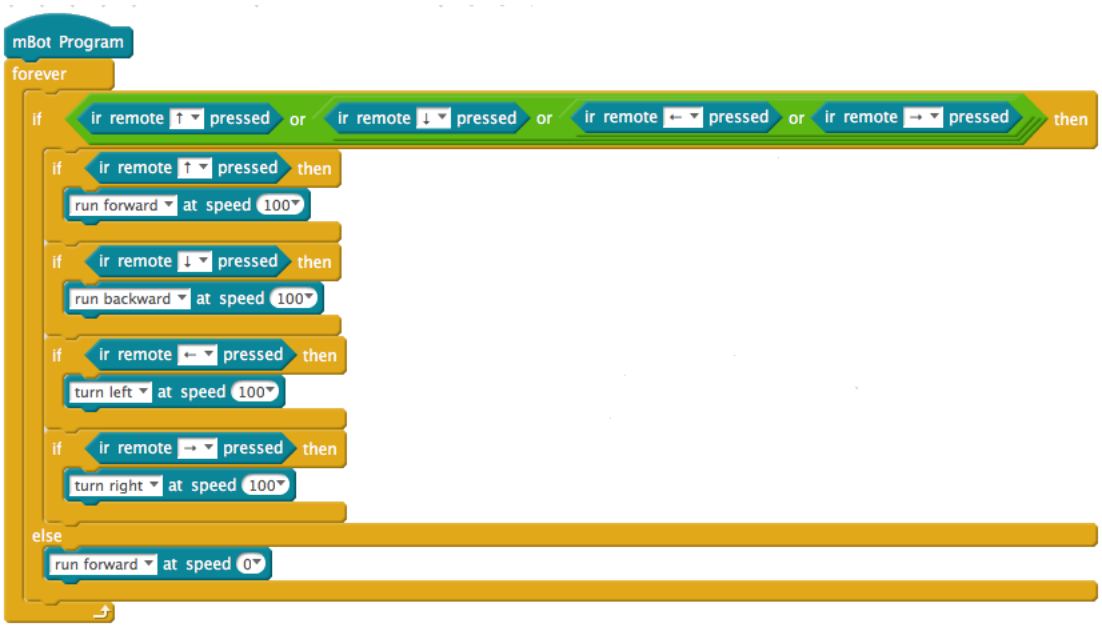


Figure 103: Baby You Can Drive My Bot Program

As you see, the test for the outer *if-then-else* block uses nested logical operators to implement the guard code. It may be easier to build each of the nested blocks from right to left, beginning with the leftmost logical comparison (left arrow or right arrow), followed by adding each layer of logical operators. Written with parentheses, the equation would be:

(up arrow pressed or (down arrow pressed or (left arrow pressed or right arrow pressed)))

Note that when a button is pressed that we are not processing for movement the motors are set at speed 0, effectively stopping the mBot's movement.

Testing & Validation

It is now time to compile and run the program to determine whether or not it performs as expected.

Test Procedure

1. Press the up arrow button on the remote. Does the mBot move forward?
2. Press the down arrow button on the remote. Does the mBot move backward?
3. Press the left arrow button on the remote. Does the mBot turn to the left?
4. Press the right arrow button on the remote. Does the mBot turn to the right?
5. Press any other button other than the arrow buttons. Does the mBot move?
6. Does the mBot stop when no buttons are being pressed?

If the mBot passes all of the tests, you have successfully completed the assignment. Congratulations!

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
2. The mBot runs backwards when I want it to go forward
   (a) Switch the motor plugs on the mCore board.
3. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
4. Compilation or Upload failed
   (a) Did the program compile without any errors?
   (b) Were you able to upload the program to the mBot?
5. There is an error in the program.
   (a) Did you create the variables correctly?
   (b) Did you initialize all of the variables?
   (c) Did you use the proper comparison operator?
   (d) Did you check that your program matches the one in Figure 103?

Challenges

Modify the program to do the following:

1. Set up additional buttons to increase and decrease the speed of the robot.
2. Set up an additional button to make a sound on the buzzer based on some action, such as a beeper when the robot is backing up.

11 | Where's My Line?

Program Description

One of the features of the mBot is the line following module (see Figure 104). This module will enable the mBot to determine whether or not it is directly over a line or not as well as determining if it has gone to the left or right side of the line.

Note that there are two pairs of emitter/receiver sensors at the front of the module. This arrangement makes it simple to determine whether the mBot is completely on the line, partially off to one side, or completely off of the line.

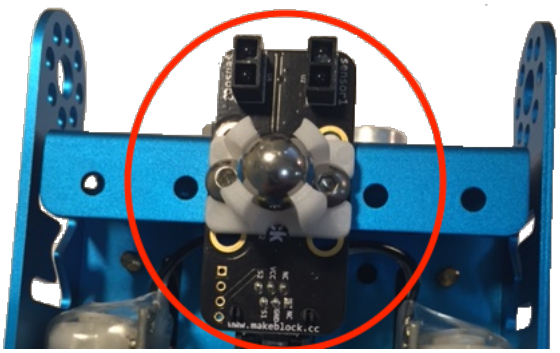


Figure 104: Line Follower Sensor

This program enables the mBot to follow a path marked by a dark (preferably black) line on a light (preferably white) surface.

Requirements

The requirements for the program are:

1. The mBot shall detect a path marked by a dark line.
2. The mBot shall move forward along the path.
3. The mBot shall turn as needed to stay on the path.
4. The mBot shall execute a strategy to find the path if it strays off of the path.

Analysis

The program requires a line-following sensor to detect differences in light intensity. These differences will tell the mBot whether or not it is over the line or not. The use of the line following sensor

assumes that the line will be of a different color than the rest of the course surface. Ideally, the line will be black with the course surface being white to provide the highest contrast for the sensor.

The line-following sensor on the mBot is very cleverly designed to have two pairs on sensors. Each pair of sensors has an emitter that radiates infrared light and a matching receiver. This arrangement makes it simple to tell if the robot is on the line (both sensors return the same value), partially on the line (one sensor returns a different value than the other), or completely off of the line (both sensors return the same value). The table for the values is shown below:

Value	Description
0	Both Sensors Over Line
1	Left Sensor Over Line / Right Sensor Off Line
2	Right Sensor Over Line / Left Sensor Off Line
3	Both Sensors Off Line

Table 7: Line Follower Sensor Values

This approach greatly simplifies the problem of writing a line-following program for the mBot (as compared to a robot with a single sensor pair, such as a LEGO™ Mindstorms EV3™).

Movement Strategy

Our strategy for moving along the path is relatively simple:

1. If both sensors are over the line, move forward.
2. If the right sensor is off of the line, turn left slightly.
3. If the left sensor is off of the line, turn right slightly.

However, the requirements don't tell us what to do if the mBot has moved completely off the line. Let's assume that the mBot traveled off the line while moving forward and simply have it back up until it senses the line again.

Strategy for Locating the Path

Let's assume that the line is within the turn radius of the mBot if it spins in place and have the mBot spin in place until the line following sensor detects the line. This is a calibration step that puts the mBot into a known position and state.

After the mBot has located the line, we can begin the actual line following process that has the mBot make a turn in the appropriate direction to stay on the line. We also need to handle the case of the mBot going completely off of the line, so let's just have it back up until it (hopefully) finds the line.

Design

The following flowchart captures our approach to movement and our strategy for re-finding the path if the mBot goes off of the line.

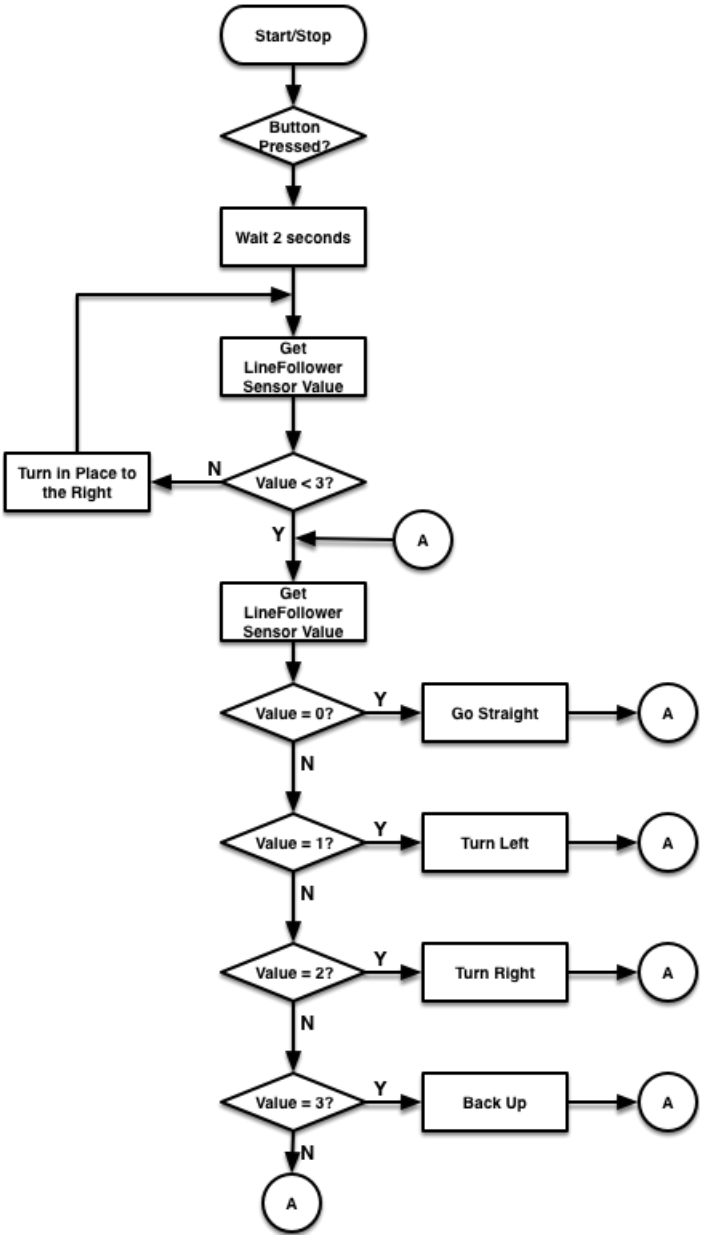


Figure 105: Line Follower Flowchart

The flowchart depicts the calibration loop that is used to find the line based on our assumption that the line is within the turning radius of the mBot. The second loop causes the robot to move according to the value returned by the line following sensor. This final loop executes until you turn the mBot off (or the battery runs out).

Program Code

As with previous programs, let's walk through the code piece by piece starting with the Calibrate block.

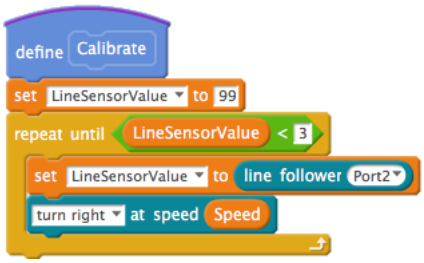


Figure 106: Where's My Line? Calibrate Block

The Calibrate block sets the LineSensorValue variable to a known (impossible) value so that it will enter the calibration loop in a known state. From that point, the block reads the line follower sensor and causes the mBot to turn to the right.

Figure 107: Where's My Line? Move Block

The Move block is fairly straightforward in operation. The block begins by getting a current reading from the line follower sensor and making a decision on how to move.

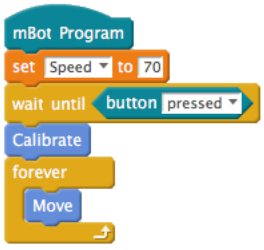


Figure 108: Where's My Line Main Logic

By using the two custom blocks, the main logic of the program is kept tight and readable. It uses a single initialization for the motor speed. After the initialization is a *wait until* block that will wait *button pressed* event on the mBot occurs. This will keep the mBot from starting to move until the button is pressed. The Calibrate block is executed and is followed by a *forever* loop that contains the Move block.

Testing & Validation

After compiling and uploading the code to the mBot, you are ready to test the program to see if it performs as you expect.

Test Procedure

1. Place the mBot on the track and press the button. Does the mBot follow the path?
2. Press the reset button on the mBot that is by the USB connector and place the mBot slightly off of the track.
3. Press the button on the mBot and see if the mBot can find the path.

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
3. Compilation or Upload failed
   (a) Did the program compile without any errors?
   (b) Were you able to upload the program to the mBot?
4. There is an error in the program.
   (a) Did you create the variables correctly?
   (b) Did you initialize all of the variables?
   (c) Did you use the proper comparison operator?
   (d) Did you select the correct port for the Line Following Sensor?
   (e) Did you check that your program matches the one in Figures 106 - 108?

Challenges

Modify the program to do the following:

1. Make noise when turning left/right.
2. Use colors on the LEDs to indicate which part of the program is being executed.
3. Develop an alternate strategy for if the mBot cannot find the line.
4. Develop a strategy for when the line is not within initial turn radius.

12 | Robotic Movement Part 1

Controlling Movement Distance

The mBot uses inexpensive analog motors that are controlled by the level of power applied to them. Programs that require the mBot to travel a specified distance require us to think like engineers. We need to measure, make calculations, and then test to see whether or not the results of the calculations need to be modified.

Thinking Like an Engineer

The mBot wheels of the mBot are 6.4 centimeters in diameter. To determine the distance traveled in a single rotation of the wheels, we need to determine the diameter of the wheels using a little math:

$$6.4cm * \pi = \sim 20.1cm$$

However, we don't have any measurements that tell us how long it takes to travel that distance. We need to do some experiments and measure how far the mBot travels over time. We also know that it is likely that the mBot will travel farther over the same time increment at a higher power level than at a slower power level so we should construct our experiment to account for that factor as well. Finally, we should take several measurements for each reading and use the mean of the measurements to factor out any minor differences from the motors.

We can record the distance traveled for each duration of time and power level in the chart provided below:

Power	**1 sec**	**2 sec**	**3 sec**	**4 sec**	**5 sec**	**Mean**
100						
125						
150						
175						
200						
225						
250						

Table 8: Straight Line Power/Time table

Note: Each mBot is slightly different, so it is very important to take measurements for your mBot. Also be aware that the battery charge level will affect movement distance.

Program Description

Engineers often create tools to enable them to make measurements to support their calculations. Let's create a simple program that will allow us to measure the distance traveled for a known time so that we can fill out the chart. When you run the program, you should take multiple measurements and then determine the arithmetic mean to get a reasonable value for each power level and duration.

Even with simple tools, you will still need to follow the process because after all, it's still a program.

Requirements

The requirement for our program are relatively simple:

1. The program shall run the motors in a straight line for a user specified power level and duration.

Analysis

As a matter of course, you should isolate the power level and duration into their own variables so that you only have to change them in one spot. You should also have the mBot wait for a button press and then a short pause afterwards so that your finger can get out of the way!

Design

The flowchart below is a simple sequential diagram that fulfills the needs of the tool.

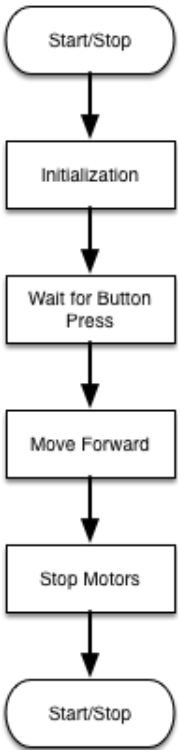


Figure 109: Straight Line Time and Power Program

Program Code

The program below contains the implementation of our straight line movement design. Note how the wait until block is followed by a short delay so that your finger can move out of the way. Pay attention to the final block in the program. Setting the power to zero for the block allows you to make the motors stop.

Figure 110: Move Forward based on Time and Power Program

Testing & Validation

For this type of program, testing is fairly limited because there are not many places that it can fail. Without access to the timer on the mBot, you cannot accurately measure the time so we will have to assume that the hardware timer on the mBot works as designed.

Challenges

Having to constantly reconnect to the mBot to change the duration or power level can be tiresome. You might want to consider making these changes to the code:

1. Use the buttons on the IR remote to increase/decrease the duration and power levels.
2. Sound a tone indicating whether the increase or decrease button has been pressed.
3. Use the LEDs to indicate the current duration and power level being used.

Now that you've written the program, you will need to make several runs (the number is up to you) for each time period and power level on the chart. When you have completed gathering your measurements, you are ready to tackle any programs that require the mBot to make a forward or backward movement. Although getting the measurements takes some time, it is data that you can, and will, reuse many times so it really is time well spent.

That's thinking like an engineer!

13 | We Like to Move It!

Program Description

One of the key elements of robotics is the ability to accurately control movement. Accurate movements require the use of either an motor encoder or the use of a type of motor called a *stepper motor*[20]

The mBot does have either encoded motors or stepper motors (although you can buy those to add on if you wish) but uses what is called a hobby gearmotor. As you should have noted, the motor blocks do not allow you to precisely control the rotations, so the distance can only be controlled through the power level and the time the engine is running, hence the point of the exercise in the last chapter.

For this program, we will utilize the movement data we gathered and put it to use to make the mBot move in a controlled fashion.

Requirements

The requirements for this program are:

1. The mBot shall move forward 25cm.
2. The mBot shall pause for one second after the first movement.
3. The mBot shall move forward 10cm.

Analysis

Obviously this is a fairly simple program and the challenges at the end of the chapter will have you do additional experiments to make the mBot forward and backward using straight-line movement. The goal of the exercise is to ensure that you know how to calculate the movement time based on the power level.

Let's tackle the calculation. You may or may not have a power/time measurement that is exactly 10cm or 25cm. If you don't, you will need to perform a calculation to determine the distance to travel. If you are using a power level of 100 and the distance traveled in one second is 12.9cm, then you would perform the following calculation to get a close estimate of the time needed to travel that distance:

$$Distance \div CmPerSecond = Time$$

The rest of the program should be relatively straightforward to design.

[20] There is a nice discussion of the differences between an encoded motor and a stepper motor on the Electronics board at Stack Exchange (http://goo.gl/N5qibQ) .

Design

After thinking through the problem and our analysis, we could develop a flowchart that looks like the one in Figure 111.

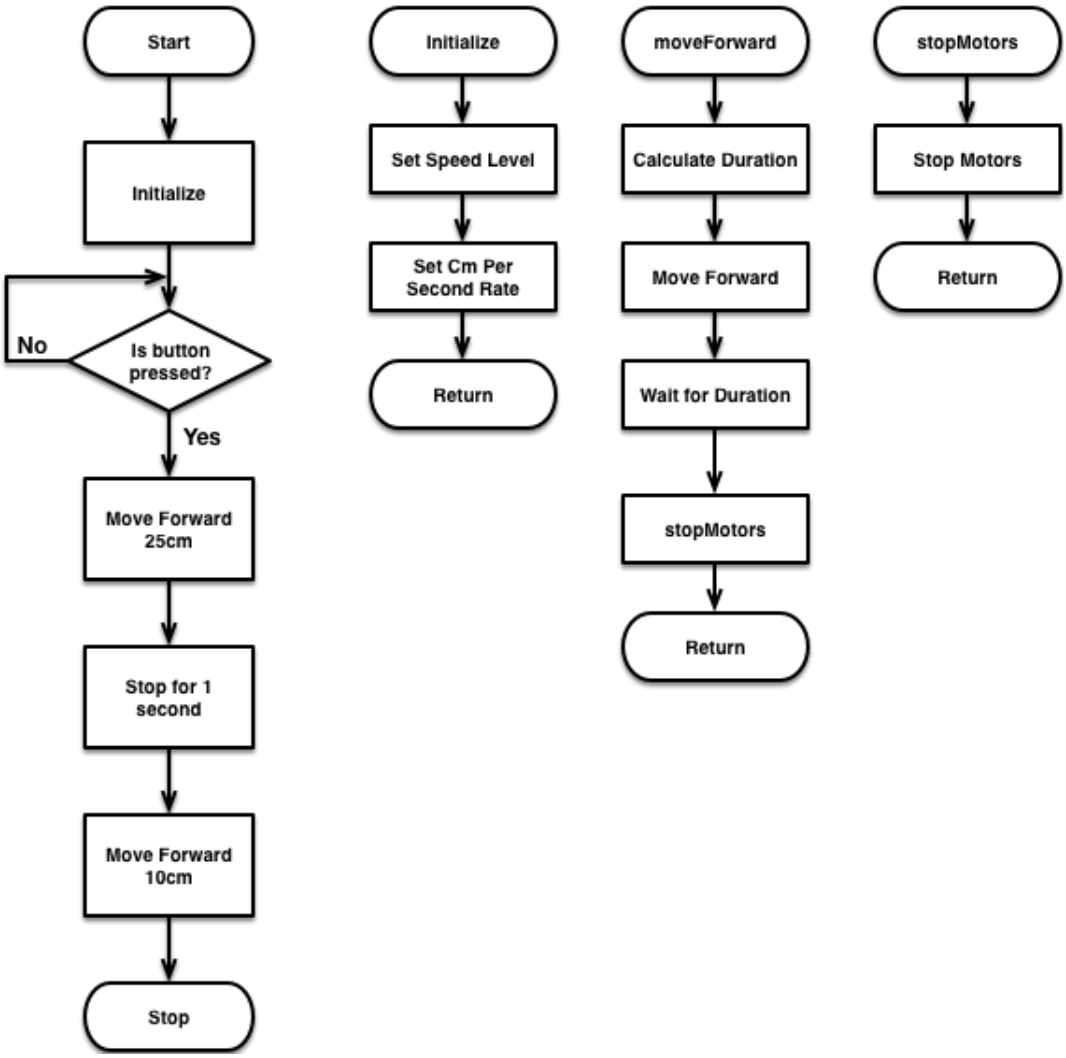


Figure 111: We Like To Move It! Flowchart

As you can see, we have created several custom blocks to encapsulate functionality to keep the main line of the program clean and readable. We are also choosing to have the mBot wait until the button is pressed.to begin its movement.

As usual, all initialization code has been captured in the Initialize block. In this case, we are using our data from the last chapter (from the Mean column for the power level we use) to set the Centimeters per Second rate of movement based on the power level we set as the Speed level.

In the moveForward block, we calculate the duration of the forward movement based on the formula from our analysis. We then tell the mBot to turn on the motors and move forward. When the duration expires, we tell the mBot to stop its motors.

The stopMotors block is somewhat contrived, but it is there for consistency.

The program design is complete so it is time to write the program.

Program Code

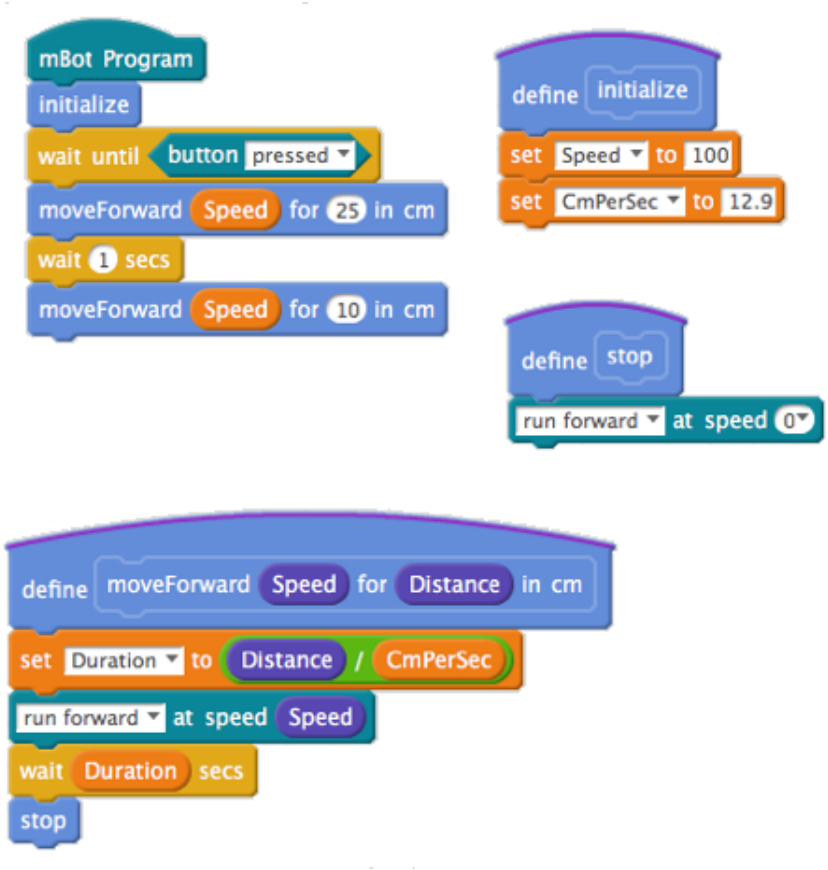


Figure 112: We Like to Move It! Program

The code accurately reflects the flowchart design, and by breaking down the logic into custom blocks (again, as per the flowchart), we end up with well-encapsulated logic that is easy to read and debug if necessary. The moveForward and stop blocks could form the basis of a movement library that you can reuse for programs that require the mBot to move.

This type of reuse is not only a great time saver, but is highly encouraged to reduce software bugs. "Why will it reduce software bugs?," you might reasonably ask. When you have code that has already been debugged and is known to work, you can rely on it to continue to work unless you (or someone else) changes something in it. After all, there is no point in doing the same work over and over again.

Testing & Validation

You will need some sort of measuring device (like a tape measure) that is divided into centimeters to test your program. You may want to mark off the distances on a big piece of paper (you can make this by taping several pieces of paper together lengthwise) to simplify measurement.

Test Procedure

1. Compile and load the program into the mBot.

2. Press the button on the mBot.
3. The mBot should move forward 25cm.
4. The mBot should pause after the first movement.
5. The mBot should move forward 10cm.
6. the mBot should stop.

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
3. Compilation or Upload failed
   (a) Did the program compile without any errors?
   (b) Were you able to upload the program to the mBot?
4. There is an error in the program.
   (a) Did you create the variables correctly?
   (b) Did you initialize all of the variables?
   (c) Did you use the proper arithmetic operators?
   (d) Did you select the correct port for the Line Following Sensor?
   (e) Did you check that your program matches the one in Figure 112?
5. The mBot moved too much or too little.
   (a) Are your batteries fully charged?
   (b) You may need to tweak your movement rates slightly.

Challenges

1. Modify the program to make the mBot flash the LEDs in different colors or make a sound based on the different intervals.
2. Modify the program to make the mBot move forwards and backwards using the LEDs and sounds to indicate distance and direction.
3. Modify the program to move forward in three increments, then move backwards in two increments to reach the starting point.

14 | Robotic Movement Part 2

Now it is time to continue our discussion about robotic movement. Having covered straight line movement, we will turn our attention to making accurate turns with the mBot. We had briefly covered turns in Chapter 10: Baby You Can Drive My Bot, especially the difference between a steering turn and a tank-like turn. Let's dig a little deeper into their differences.

Steering Turns

You may recall from the earlier discussion that a steering turn involves the wheel on the outer (longer) arc of the turn rotating faster than the wheel on the inner (shorter) arc of the turn. In many cases, you may choose to either have the inner wheel not move at all (resulting in a turn that pivots on that wheel) or to move less (resulting in a car-like steering maneuver). Let's investigate this behavior right now.

The sharp turn that pivots on the inner tire would be achieved by setting the motor for the pivot point to zero and having the motor on the outer wheel run for a designated period of time. There are two approaches for determining the time required to make the turn to a particular angle:

1. Calculate the distance the outer wheel needs to travel based on the speed used.
2. Calculate the total time required for the outer wheel to make a 360° turn based the speed used and take the fraction of time required by the angle of the turn.

In the first method, we would need to know the measure of the turn and the rate of movement in centimeters per second. Let's first calculate the distance that would need to be traveled with the following formula:

$$Distance = (((WheelDiameter * \pi) \div 360) * AngleInDegrees) * (WheelBase/WheelDiameter)$$

The formula determines the circumference of a wheel then divides it by 360 to get the length of one degree. This value is then scaled by multiplying it by the ratio of the wheel base to the wheel diameter to scale the distance the wheels must turn to support the larger arc of the wheelbase of the mBot. To get the time required to make the turn, you would need to divide the distance to be traveled by the rate of travel:

$$TravelTime = Distance \div RateOfTravel$$

Math is awesome!

You will also likely need to do some tweaking to account for the differences in your motors, battery charge level, etc. However, this calculation should get you most of the way there.

The mBlock environment provides you with blocks to control each motor individually if you like, and that is what would be required for steering movements.

The second method of controlling turns is to measure how long it takes for the mBot to complete a 360°turn at a particular speed. You can then calculate the time as:

$$TravelTime = (DegreesOfTurn \div 360) * Turn360Time$$

As before, the actual turn time depends on the speed/power that is applied to the motor, battery charge level, etc., and will require some minor tweaking to get right.

Either of the previously mentioned calculations work for a pivot turn and for the mBot, a pivot turn for car-like steering is probably adequate. If you are interested in pursuing more accurate calculations, there is a good reference paper at the Rossum Project[21] that discusses differential steering in depth.

Tank-like Turns

The Makeblock firmware for the mBot specifies a tank-like turn. Internally the block takes the power setting and applies it to the outer wheel, negates the value and applies it to the inner wheel.

For example, if the mBot is asked to make a left turn at a power of 100, internally the right wheel (outermost wheel in the turn) will turn in a forward movement at a power of 100. The left wheel will turn in a backward movement at a power of -100.

The calculations mentioned before work with tank-like turns so we won't have to go back over that. For the sake of simplicity, let's create a table of turning times for a 360° turn at several different power levels. These turns are based on the tank-like turns provided by the run [direction] block in mBlock.

You will need some way to measure a 360°turn to complete the turning table below. There is a 360°compass in Appendix F: Resources that you can print out if you have the eBook version of the book. The diagram will also be available as a PDF file in the code repository[22]

We need to fill out the following table:

Power	Time
100	
125	
150	
175	
200	
225	
250	

Table 9: Turning Power/Time

so it is time to build a tool that let's you measure the time it takes to make a 360°turn.

Program Description

To fill out the chart you will need to write a program that allows you to make the mBot turn for a specified duration and power level.

[21] http://rossum.sourceforge.net/papers/DiffSteer/
[22] https://goo.gl/CWROM5

Requirements

The requirements for this program are:

1. The program shall cause the mBot to turn for the specified duration and power level.

Analysis

Looking at the requirements, this will be a relatively simple program requiring the parameters of the move block (set to turn right or turn left).

Design

The flowchart is equally simple, although note the use of variables to make the program more readable.

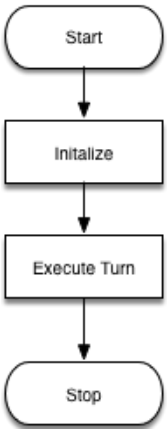


Figure 113: Turn Tool Flowchart

Program Code

The program code is also straightforward, although you should note that there is a *wait until* block used to keep the mBot from turning until you press the button on the mBot.

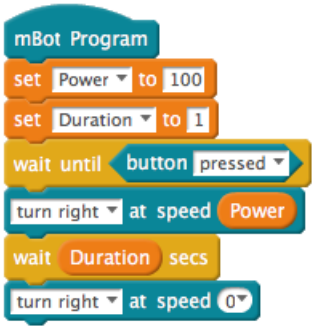


Figure 114: Turn Tool Program

Testing & Validation

You will need a stopwatch if you want to test whether the hardware clock on the mBot is accurate although this is unnecessary for this application. You will need to run the test several times to make sure that you have the correct timing for each power level.

Test Procedure

1. Set the power level at 100.
2. Set the duration for 1 second.
3. Compile and upload the program.
4. Run the program
5. Press the button.
6. Adjust the duration as necessary and repeat the process until the mBot completes a 360°turn.
7. Repeat the process for the other power levels

15 | Trace A Shape

Program Description

Now that we have the data to support controlled turns, let's do a practical exercise. In the Resources folder of the Github repository[23] associated with the book is a file named TraceAShape.pdf contains a triangle that can be printed on an 11" x 14" sheet of paper. You can likely guess that we will write a program to make the mBot trace the triangle.

Dead Reckoning

Dead reckoning is the navigational process used to calculate a new position based on the current position, a direction for the new position, and a known distance to the new position. Be aware that dead reckoning can be subject to cumulative errors over time. If your robot is making several movements, small inaccuracies in each of the movements will add up to make future movements less accurate.

Requirements

The requirements for this program are:

1. The robot shall move forward 25cm.
2. The robot shall execute a 120°turn.
3. The robot shall repeat these operations until it returns to the starting point.

Analysis

The first requirement can be satisfied by retrieving the rate of movement from the Power/Time movement table for the power level you select. You will need to use the following formula to determine the time that the motors should run:

$$RunTime = Distance \div RatePerSecond$$

The next task is to determine the time required to make a 120° turn, so retrieve the turning rate from your 360° Turn Chart from Chapter 14: Robotic Movement Part 2. You will likely need to make some small adjustments to the running times for both the straight line movement and the turning movement, but the reference numbers should get you most of the way there to start with.

Hint: *Turning at a slower speed results in a more accurate turn.*

[23] https://goo.gl/CWROM5

So where did the 120° measurement come from for the turns? When dealing with non-intersecting equi-angular polygons, you can derive the measure of the exterior angles by dividing 360° by the number of edges. For example, a triangle has three sides, so 360° divided by 3 would equal 120°.

Design

From the requirements, we know that we are dealing with a triangle and that we need to repeat a series of actions three times. Therefore we should use a loop to contain the movement logic. The resulting flowchart would be:

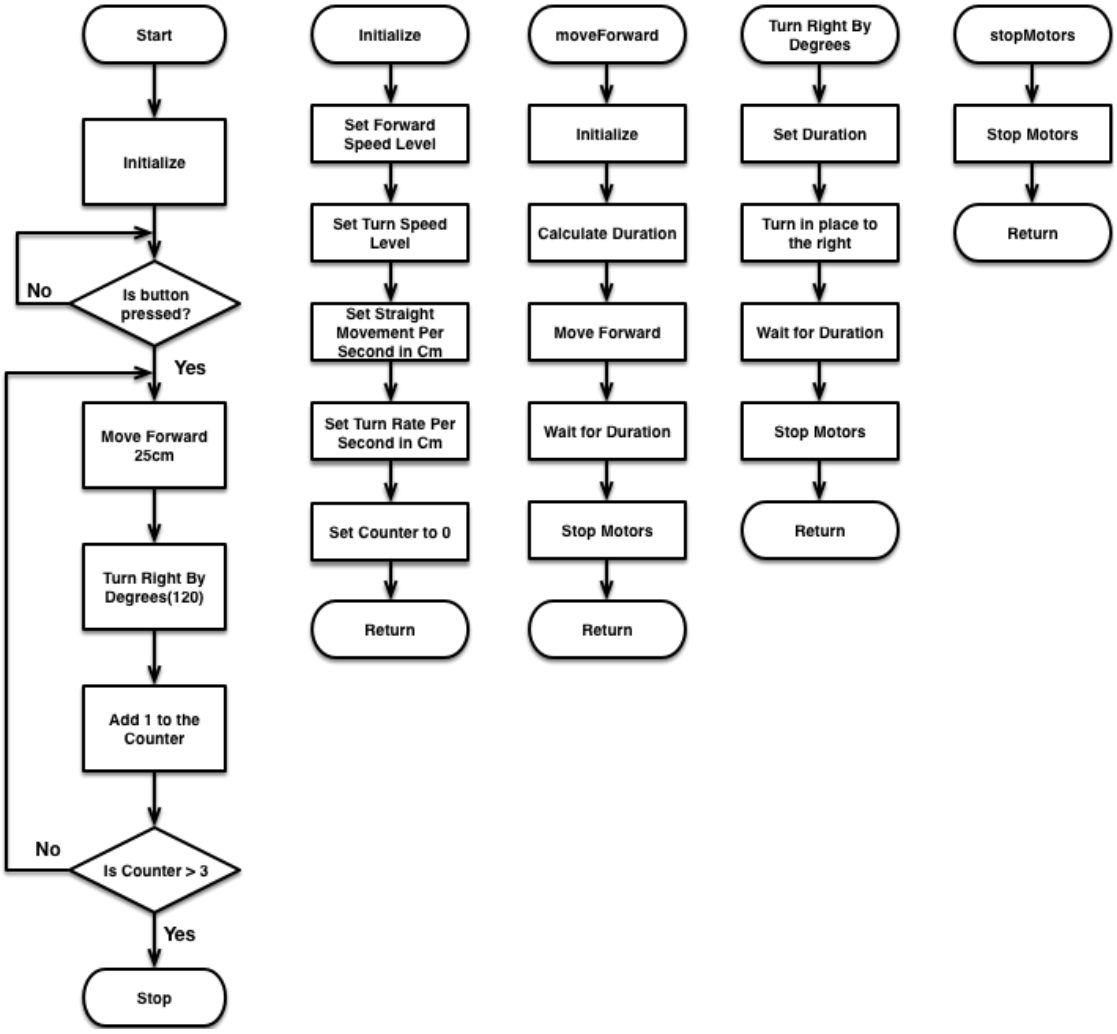


Figure 115: Trace A Shape Flowchart

Program Code

In the *initialize* block, we will set up several variables to support the program operations and to minimize the places where we need to change values.

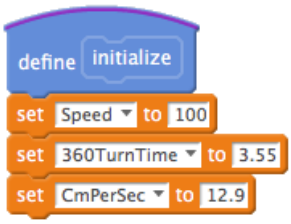


Figure 116: Trace A Shape Initialize Block

The *moveForward* block takes two parameters for speed and distance, then makes the necessary calculation to determine how long to run the motors. This approach to creating the block makes it reusable with future programs requiring this type of movement.

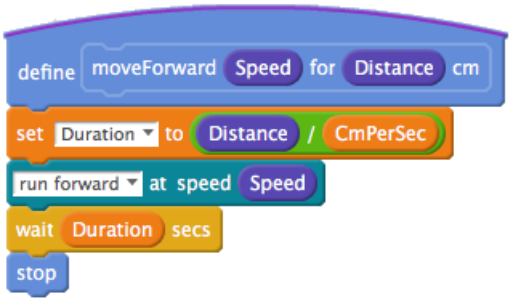


Figure 117: Trace A Shape Move Forward Block

The *turnRight* block encapsulates the calculations needed to make the mBot turn to the right for the specified number of degrees. The calculation itself is:

$$Duration = (Degrees \div 360°) * 360TurnTime$$

The block could be improved by adding a *Speed* parameter if the turns needed to be performed at a different speed than the straight line movements.

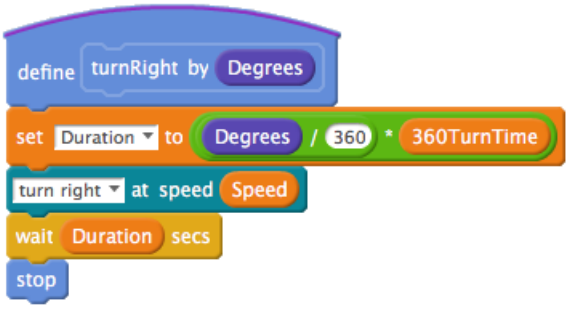


Figure 118: Trace A Shape Turn Right Block

The *stop* block is somewhat contrived for completeness. However, readability is improved by hiding the *motor* block within the *stop* block.

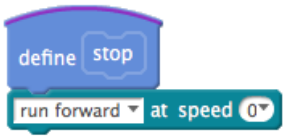


Figure 119: Trace A Shape Stop Block

Finally, the main program keeps with the practice of minimizing the amount of code while maximizing readability.

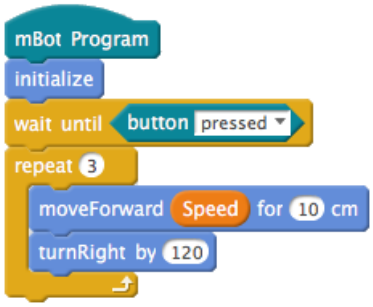


Figure 120: Trace A Shape Main Program

Testing & Validation

Test Procedure

1. Compile and upload the program.
2. Press the green button on the mBot.

Does the mBot accurately execute all of the movements and return to its starting point?

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBot runs backwards when I want it to go forward
   (a) Switch the motor plugs on the mCore board.
3. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?

(b) Have you upgraded the firmware on the mBot from the Connect menu item?

(c) Did you install the Arduino driver for the mBot?

4. Compilation or Upload failed

   (a) Did the program compile without any errors?

   (b) Were you able to upload the program to the mBot?

5. There is an error in the program.

   (a) Did you create the variables correctly?

   (b) Did you initialize all of the variables?

   (c) Did you use the proper arithmetic operators?

   (d) Did you check that your program matches the one in Figure 116 - 120?

Challenges

1. Modify the program to trace the outline of a square.
2. Modify the program to trace the outline of a pentagon.
3. Modify the program to trace the outline of a hexagon.
4. Modify the program to trace the outline of an octagon.

Optional Advanced Challenges

1. Create a version of the program that can trace a circle.

   (a) Start the mBot on the outside edge of the circle and have it trace the shape.

   (b) Start the mBot at the center of the circle and have it move the distance of the radius and execute a 90° turn before tracing the shape.

2. Create a version of the program that follows a path you make.

   Hint: *If the path requires left turns, you will need to implement a leftTurn block similar to the rightTurn block.*

16 | Obstacle Avoidance

One of the challenges for robotic movement is the occurrence of physical obstacles, such as furniture, walls, etc. Whenever a physical obstacle is encountered, the mBot must execute an avoidance strategy to go around the object. The focus of this chapter centers on developing an object avoidance strategy for the mBot.

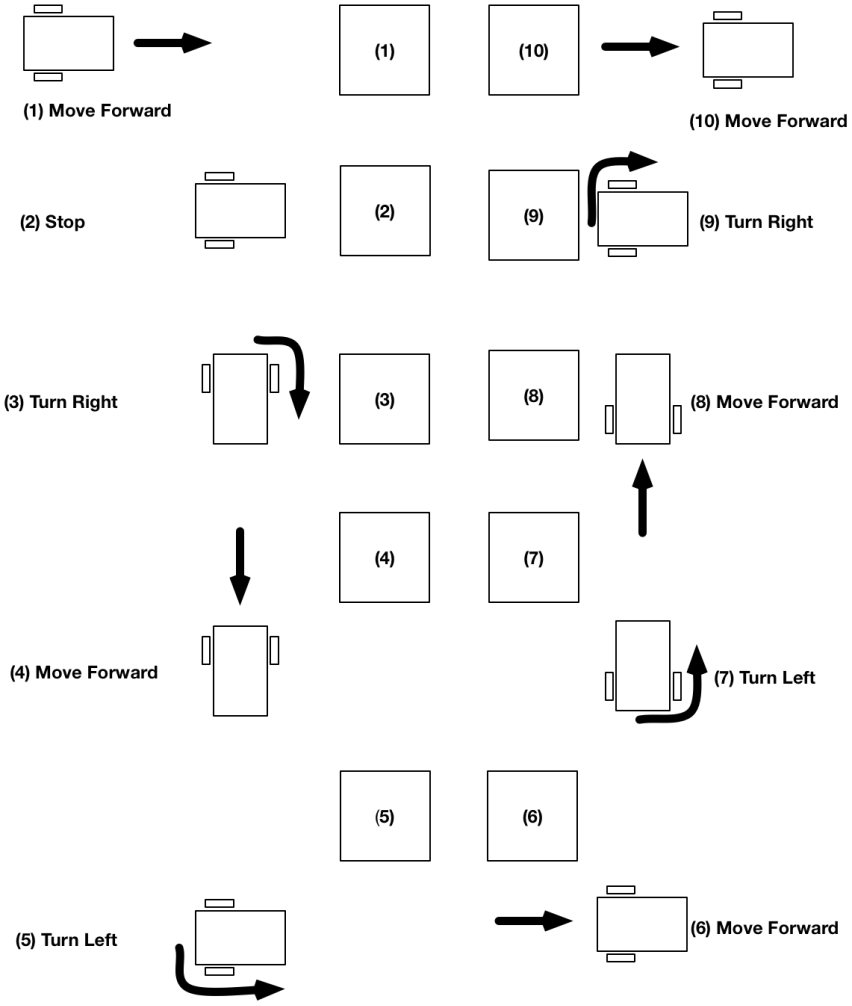


Figure 121: Avoid Obstacle Movement Strategy

Note: *This program will be concerned with obstacles in the path that protrude up from the ground, not with holes.*

Program Description

The program will enable the mBot to move forward and avoid any obstacles it encounters.

Requirements

The requirements for this program are:

1. The mBot shall execute an avoidance strategy whenever it encounters an object within 5 centimeters.
2. The avoidance strategy shall allow the mBot to navigate around the object and then continue forward movement on the same line it was on before moving around the object.

Analysis

Having already learned to control movement and turns in previous exercises, it is time to put our knowledge to work. We need to develop a strategy to detect and avoid obstacles in the path of the mBot.

From the Ultrasonic Theremin exercise, we know that the ultrasonic sensor can detect objects at range. It makes sense to use that sensor to detect any obstacles within the specified range.

Developing the actual avoidance strategy is our next objective. A strategy is a plan of actions to take so let's think through what might be a reasonable strategy. One strategy might be:

1. Back up a specified distance.
2. Turn right (or left) and move forward.
3. Turn left (or right if you turned left in step 2) and move forward.
4. Turn left (or right if you turned left in step 2) and move forward.
5. Turn right (or left if you turn left in step 2) and move forward.

This is a simple type of *dead-reckoning* strategy that assumes that the dimensions of the obstacle are known. While this strategy will work for the purposes of demonstrating detection and avoidance of an obstacle, in real life a more flexible strategy would be required. The real world that the mBot interacts with may have obstacles of varying shapes and sizes as well as being arranged in an unpredictable fashion. One of the challenges at the end of this chapter is to think of a more flexible approach to dealing with obstacle detection and avoidance.

Design

Main Logic

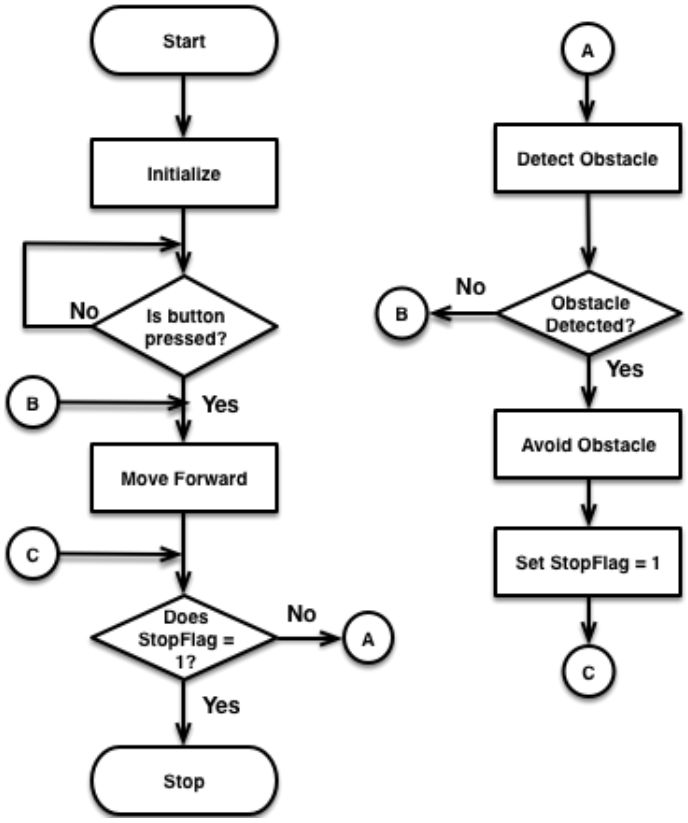


Figure 122: Obstacle Avoidance Main Logic Flowchart

The logic presented begins with an initialization block to set up any variables followed by a *wait until* block that waits for the button on the mBot to be pressed before any movement occurs. From this point, the mBot moves forward and attempts to detect any obstacles. If an obstacle is detected, the mBot executes its avoidance strategy followed by setting the exit flag so that the main loop will exit and cause the mBot to stop.

This flowchart makes heavy use of on-page connectors (the circles with letters in them) to keep the diagram clear and concise. When you are designing complex programs, you will want to take advantage of on-page connectors to keep the diagram from becoming a tangle of lines.

Detect Object Logic

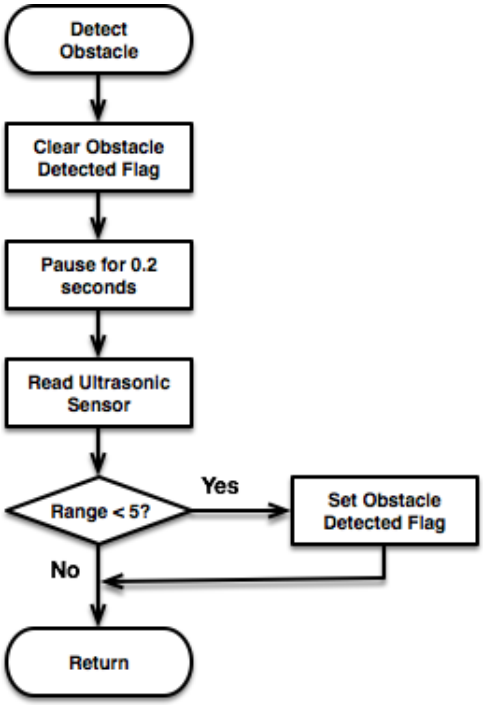


Figure 123: Obstacle Avoidance Detect Obstacle Logic Flowchart

The *Detect Obstacle* logic clears the flag denoting whether or not an obstacle has been detected, then pauses briefly to read the ultrasonic sensor to ensure an accurate reading.

The Joys (and Perils) of Inexpensive Robotics

We are living in a remarkable time where the size and cost for robotic components are amazingly low. You can assemble a fairly advanced robot for $75-$100 dollars (US Dollars) with components that are readily available both locally and from the Internet. However, the old adage, "You Get What You Pay For," should be kept in mind.

As a general rule, hobby-grade components are more than adequate for most needs. However, they will not be as accurate or as consistent as higher priced components aimed at commercial markets. The ultrasonic sensor that comes with the mBot is a hobby-grade component and in some cases you may have to find workarounds for behaviors that you might not experience with a commercial-grade ultrasonic sensor.

That being said, unless you really, really need the precision and reliability that a commercial-grade component provides, stick with the hobby-grade components. Your wallet will thank you, and you'll get to use your own ingenuity or leverage the ingenuity of others to make things work.

Avoidance Logic

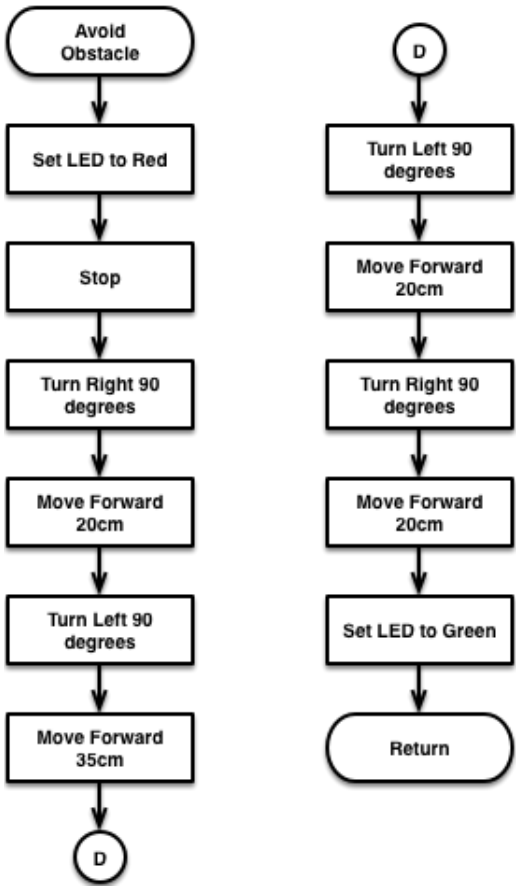


Figure 124: Obstacle Avoidance Strategy Flowchart

The *Avoid Obstacle* logic is very straightforward. After setting the LEDs to red to indicate that the avoidance logic is being executed, the mBot makes a series of movements based on the dead-reckoning strategy that is being used. When all movements contained within the strategy have been executed, the LEDs are set back to green, indicating that the avoidance strategy is no longer being executed.

The use of the LEDs to indicate the current *state* of the mBot program enable the programmer to confirm that the avoidance strategy is being executed. This type of visual marker is often used when testing a program where no display is available and it makes the mBot's movements much more colorful and interesting.

Having previously covered the logic behind moving and turning, we will not repeat those flowcharts here. You can refer to Chapter 13: We Like to Move It! and Chapter 15: Trace A Shape if you need to refresh your memory on how that logic works.

Program Code

This is the largest program that we have written so far, and like the earlier programs we will break it into smaller parts that can be quickly and easily understood. We will not revisit the movement-related blocks from earlier chapters to avoid needless repetition.

You should consider building an empty *template* program that contains those blocks so that you can easily reuse them in other programs. If you have not yet done so, now might be a good time to do that. This habit is definitely considered a best practice and it has the added benefit of letting you focus on the new parts. When this practice becomes automatic you will truly be working smart, not working hard (and great programmers are really lazy like that!!!).

Initialization

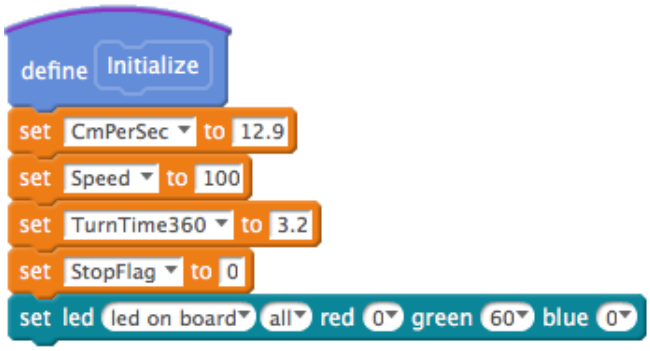


Figure 125: Obstacle Avoidance Initialization Block

The *Initialize* block sets up the initial values for our variables and ensures that the LEDs are set to green to indicate a normal, running state.

Detect Obstacle

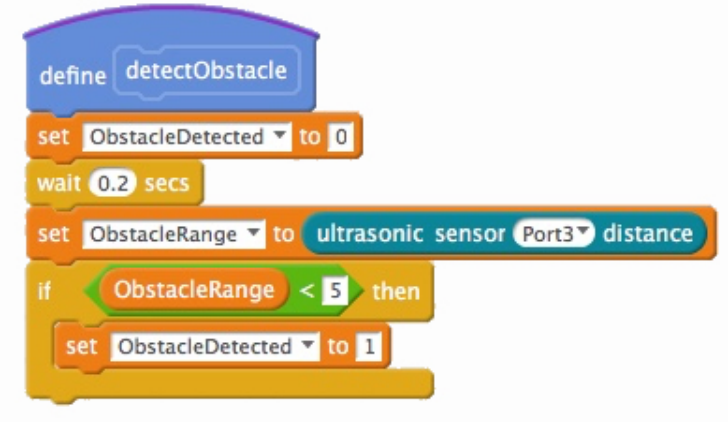


Figure 126: Obstacle Avoidance Detect Object Block

The *detectObject* block has the task of setting a flag if an obstacle is within a certain range. This is a generic enough task that it should be considered for addition into your template program. Notice that the block allows for a 0.2 second *settling* period so that the ultrasonic sensor can get a solid reading.

Avoid Obstacle

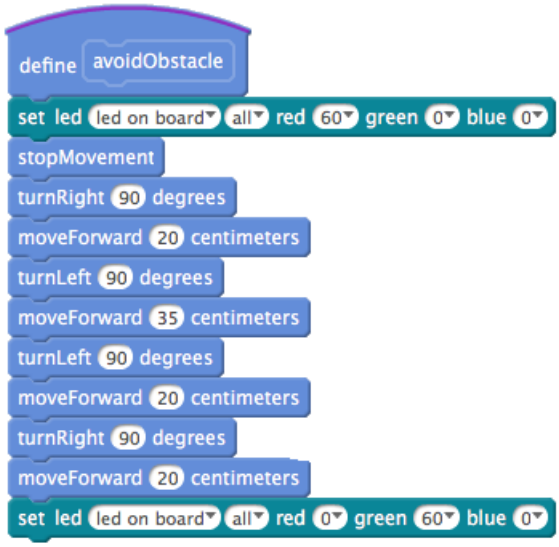


Figure 127: Obstacle Avoidance Avoid Obstacle Block

The *avoidObstacle* block contains the obstacle avoidance strategy. As mentioned earlier, a hard-coded strategy is not the way to go on a real-world application, but it suffices for our purposes here.

The block itself is straight forward in implementation, setting the LEDs to red while executing the strategy, and then setting the LEDs back to green when the strategy execution has been completed. The advantage of encapsulating the strategy in this fashion is to allow the programmer to change the strategy without the need to change the rest of the program.

Main Program Logic

Figure 128: Obstacle Avoidance Main Program

The main program logic uses the familiar *wait until* block to prevent the mBot from moving until the button on it is pressed. The rest of the program follows the logic set by the flowchart. Note that the program uses a *repeat until* loop to continue moving until the *StopFlag* is set after an obstacle is detected and avoided.

Testing & Validation

Test Procedure

1. Compile, upload, and run the code on the mBot.
2. Press the button on the mBot.

Troubleshooting & Debugging

If your mBot did not successfully avoid the obstacle, take a look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBot runs backwards when I want it to go forward
   (a) Switch the motor plugs on the mCore board.
3. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
4. Compilation or Upload failed
   (a) Did the program compile without any errors?
   (b) Were you able to upload the program to the mBot?
5. There is an error in the program.
   (a) Did you create the variables correctly?
   (b) Did you initialize all of the variables?
   (c) Did you use the proper comparison operator?
   (d) Did you check that your program matches the one in Figures 125 - 128?

Challenges

The strategy used to avoid an obstacle is hard-coded for a specific set of movements and dimensions. A more flexible approach would be to move forward for a specified distance, and continue to turn in the direction of the obstacle (left if you turned right to avoid it), and check to see if it is still in the way. Now is a good time to consider other more flexible ways of detecting and navigating around the obstacle.

1. Modify the program so that the mBot can move around obstacles of unknown dimensions.
2. Modify the program so the mBot can deal with multiple obstacles of unknown dimensions.

17 | Bashin' at the Basho

The Ancient Sport of Sumo Wrestling

You may or may not be familiar with the ancient Japanese sport of sumo wrestling. In this sport, the two wrestlers attempt to push their opponent out of the ring or to throw the opponent to the ground. The wrestlers are typically very large men who are specially trained for the sport.

Although most of the sumo wrestlers are Japanese, there are a few who come from the United States and one was even the grand champion!

The Modern Sport of Robotic Sumo Wrestling

In modern times, the sport of sumo wrestling has become a popular pastime for roboticists. The robotic sumo competitions have weight classes just like the real sumo competitions and an opponent is eliminated when it is pushed out of the ring (whether intentionally or accidentally) or if the opponent is flipped and cannot continue moving.

We won't worry about flipping the mBots, but we will focus on trying to push them out of the ring!

Scoring

- Pushing an opponent out of the ring is an automatic victory.
- A charge at an opponent that results in contact with that opponent scores 1 point.
- If no sumo robot is pushed out of the ring, the sumo robot with the most points wins.
- Ties are decided by the sumo robot owners playing Rock-Paper-Scissors in a best of five match (3 victories to win).

Robots versus Remote Control Vehicles

You've likely seen or heard of television shows that feature competitions between "robots." These human-driven machines are armed with all manner of saws, hammers, spikes, wedges, etc., and flail away at each other. While it is an exciting spectacle, the stars of the show are merely tele-operated vehicles, not robots.

"What!?!" you say, "but the announcers and participants call them robots!" You'd be right to be confused, because the popular media labels a lot of things incorrectly and this is an excellent example of that unfortunate practice.

Let's think back to the definition provided by Maja Materić in the preface:

> "A robot is an autonomous system which exists in the physical world, can sense its environment, and can act on it to achieve some goals."

The key term is the word *autonomous*, meaning the system can act on its own. In essence, if a human is actively providing the *intelligence* then the system is not a robot but a tele-operated system.

Some robots are trained via a human controlled harness, but do not rely on human intelligence after being trained and operate autonomously. However, the vehicles in the television robotic battle shows and their ilk are more akin to remote controlled vehicles — but a lot more dangerous! That being said, don't let that definition keep you from building tele-operated vehicles. Just recognize that calling them robots is misusing the term.

Program Description

The goal of this program is to create a program that allows the mBot to act as a robotic Sumo wrestler.

Requirements

The requirements for this program are:

1. The robosumo mat shall be circular with a minimum interior diameter of 24 inches (~61cm).
2. The mBot shall be able to move around within the confines of a ring.
3. The mBot shall be capable of detecting the edge of the ring.
4. The mBot shall be able to execute an avoidance strategy to stay within the ring when a ring edge is detected.
5. The mBot shall be able to locate an opponent.
6. When an opponent is located, the mBot shall charge at the opponent and attempt to push the opponent from the ring.
7. The mBot shall have the capability to stop movement after the round duration has expired.

Analysis

To fulfill the requirements of this program, you will need to solve several problems:

1. Detect the edge of the ring and move away from it.
2. Detect other robots and move towards them.
3. Track the time to know when to quit

Next you need to ask yourself:

1. Which sensor can be used to detect the edge of the ring?
2. What avoidance strategy can be utilized to stay within the ring area?
3. What strategy (or strategies) can be utilized to locate an opponent?
4. Which sensors can be used to detect an opponent?
5. What can you use to track the elapsed time?

Detecting the Edge of the Ring

The first problem requires the mBot to detect the ring edge and to turn and move away from it. You can reuse the movement modules from earlier programs to shorten development time because you have already tested those modules.

To ensure that the turns don't keep turning directly into the ring edge, we can specify that each turn will be 120°. We know from the Where's My Line? Program that the line following module returns a value ranging from 0-3, where 0 indicates that both sensors are over the line, 1 indicates the right sensor is over the line, 2 indicates that the left sensor is over the line, and 3 indicates that neither sensor is over the line.

Let's simplify our coding by stating that if both sensors are over the line, the mBot will execute a left turn before proceeding, just as if the right sensor had been over the line. If the left sensor is over the line, the mBot will execute a right turn before proceeding. We can ignore the case where both sensors are not over the line because that should be the normal case when moving.

Detecting an Opponent

When an opponent is detected in attacking range, the mBot should charge at it and attempt to push it out of the ring. In practice, there are several strategies that can come into play. The simplest strategy is to move around the ring until an opponent is detected. Since the Ultrasonic sensor is the only sensor on the mBot that is capable of detecting objects at specified distances, it will be the primary means of detecting an opponent. When there is no opponent in range, the mBot should revert to moving at its normal speed and spinning slowly at intervals to look for an opponent.

Tracking the Time

The mBot has a clock on board that counts time in milliseconds from when the mBot was turned on. This timer can be reset and effectively used to know when it is time to stop moving. It is important to note that the timer is not blocking, i.e., it will not stop the program from executing unlike a wait block which will stop the program until it is done waiting.

Design

With the increased complexity associated with this program, we will take a top-down design approach beginning with the main logic and working through to each custom block needed to support the main logic. The other approach to design is referred to as bottom-up where you design all of the blocks first and then the main logic. There are times that this approach may yield better results, but many software engineers will initially work from the top down.

Main Program Logic

In the top-down approach, the designer is concerned with the high-level details, in this case the high-level logic of the program.

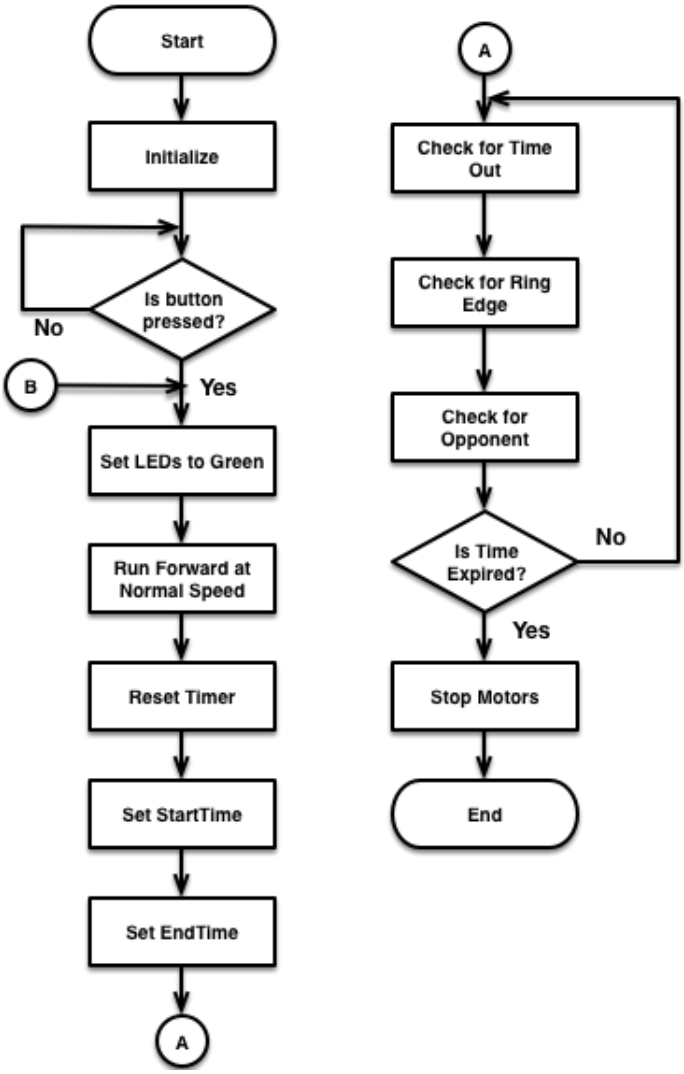


Figure 129: Bashin' At The Basho Main Program Logic

You may note that we continue the pattern of performing our initialization (*Initialize*) and have the program enter a *wait until* loop so that the mBot doesn't just start moving right out of the gate.

The program changes the LEDs to green to provide a visual indication of the current movement state, in this case indicating a normal movement state, and continues by having the mBot begin moving forward.

A reset command is issued to the hardware timer and the Start Time variable is set to the current timer value. The End Time is calculated immediately afterwards.

At this point, the program enters into its main loop. Within the loop, the first action is to check to see if the match time has expired. Following this statement, a check is made for a ring edge, and finally a check to see if an opponent it within range.

The loop will exit when time expires and will subsequently stop the motors.

Check for Ring Edge Logic

The logic for turning right and left are reused from past programs and will not be shown. Likewise the logic for checking to see if time has expired for the match will not be shown due to its simplicity.

The logic to check for the edge of the ring is shown in Figure 130:

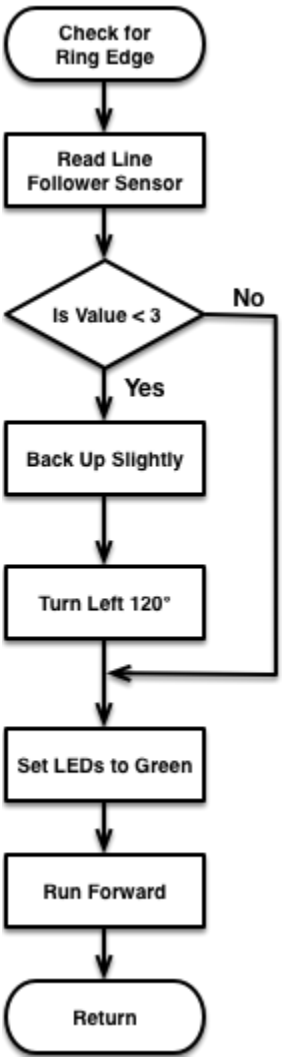


Figure 130: Bashin' At The Basho Check for Ring Edge Logic

This is a relatively straightforward strategy that uses the line sensor to detect the ring edge. If the edge of the ring is detected, the program uses a simple strategy that causes the mBot to back up slightly, turn left by 120°.

Following the turn, the LEDs are set to green to account for those occasions that the ring edge is detected during a charge. Finally, the mBot is reset to a normal forward movement state before returning to the main program logic.

Check for Opponent in Range

The next block to explore checks to see if an opponent is within the attack range of the mBot. Our requirement states that the mBot should move towards an opponent when one is in range. The program will use the Ultrasonic sensor to detect whether or not an opponent is within a designated attack range, and moving in a charge towards to opponent if one is within range.

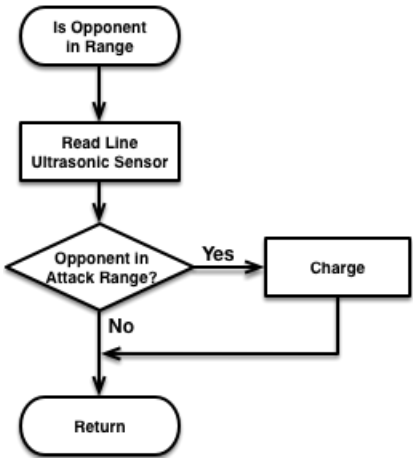


Figure 131: Bashin' at the Basho Detect Opponent Logic

Charge at Opponent Logic

The last block to explore contains the charge logic. Although it could be argued that the logic could be directly included in the Is Opponent In Range block, it's often better to isolate this type of code for testing and debugging purposes. The logic is displayed in Figure 132 below.

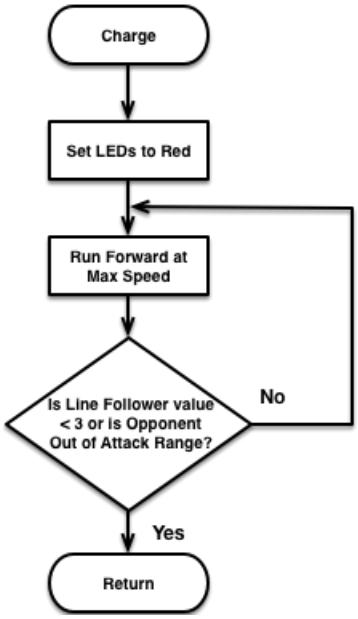


Figure 132: Bashin' at the Basho Charge Logic

Note that the if statement uses two different conditions (Line Follower Value < 3 and Opponent Out of Attack Range) to control the loop. The use of two or more conditions is referred to as a compound conditional statement by mathematicians and computer scientists. Because the compound conditional is an or, the loop will exit if either condition is met. If the statement had used an and, both conditions would have to be met to exit the loop. On that note, let's take a slight diversion to talk about logical comparative operators.

Logical Operators

The mBlock environment provides three logical operators, *and*, *or*, and *not* that can be used if statements. The previous paragraph mentioned the *or* and the *and* operators so for completeness you should know that the *not* operator negates whatever other comparison is being made. For example, the Line Follower sensor will only return the values 0-3.

The statement not LineFollower = 3 would read as not(LineFollower = 3). Because the LineFollower sensor returns integer values in the 0-3 range, the statement would be the equivalent of saying LineFollower < 3 because 0, 1, and 2 are less than three.

When trying to optimize code size and performance, it is often useful to determine whether or not a not operation can be rewritten to avoid using the not because it is an extra operation (time) and takes extra space in the PROGMEM. Not all *nots* can be eliminated in this fashion, but it is worthwhile to validate whether the *not* operator really needs to be used in a comparison.

Program Code

Main Program Logic

Let's look at the main program logic first before diving into the custom blocks this time.

Figure 133: Bashin' at the Basho Main Program

The logic follows the flowchart, but there seems to be many more blocks in the main program than in earlier programs. This is an implementation decision and is based on whether or not the

developer believes that adding a custom block to hide these details really makes sense. In this case, the developer thought it unnecessary to do so. However, you may choose to create a custom block to hide these details from the main program if you like.

The program starts with an *Initialize* block followed by a *wait until* block with a small delay to let the finger get out of the way. This pattern should be a familiar one at this point.

The next series of blocks is concerned with set the LEDs to green to indicate a normal movement state followed blocks to reset the hardware timer, assign a value based on the hardware timer as the starting time and setting the end time to a calculated value based on the start time. In this case, the start time is incremented by 30 seconds. For a three minute round, the end time should be set to a value of the starting time plus 180 seconds.

At this point the program enters its main loop, a *repeat until* loop that will continue to run until a flag (*StopFlag*) is equal to a value of one. The use of the flag allows to set the value for an exit condition from any block used within the loop. Within the loop are the three custom blocks *checkTimeOut*, *checkForRingEdge*, and *isOpponentInRange*. These blocks form the backbone of the program's operations and guide the mBot during its journey through the sumo ring. The final block ensures that the motors on the mBot stop when the main loop exits.

Initialization

The *Initialize* block to contain all of the housekeeping for the program, such as setting up the initial variables and their values.

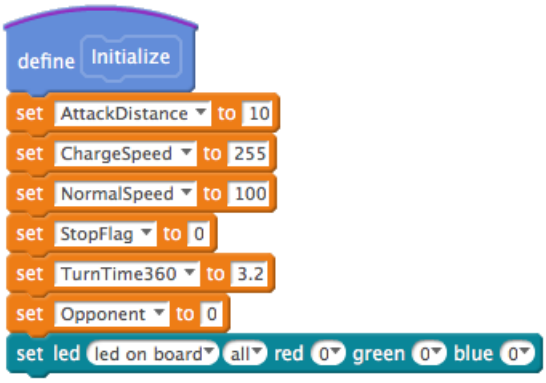


Figure 134: Bashin' at the Basho Initialize Block

Check for Timeout

The *checkTimeOut* block is responsible for setting the exit flag if the allocated time for the match has expired. It is trivially simple, but hiding the implementation in a custom block improves the readability of the main program.

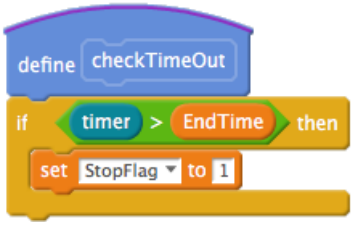


Figure 135: Bashin' At The Basho Check Timeout Block

Check For Ring Edge

The *checkForRingEdge* is a slight variation on how the line follower sensor was used in Chapter 11: Where's My Line.

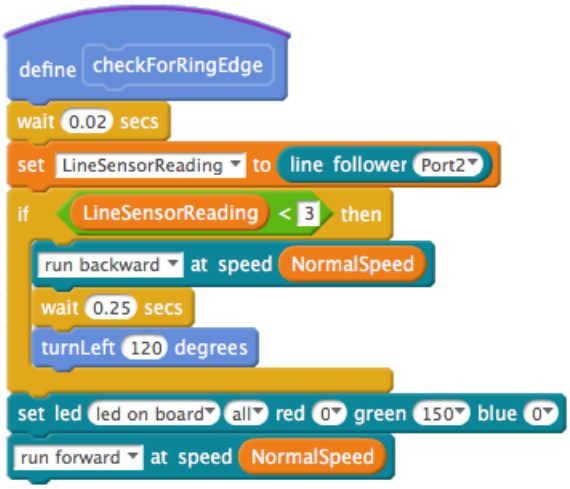


Figure 136: Bashin' At The Basho Check For Ring Edge

Note how the program makes a slight pause before reading the line follower sensor to ensure an accurate reading. From this point, the value is checked to determine whether or not the edge of the ring has been detected. If the edge of the ring has been detected, the program executes its avoidance strategy by backing up and turning. Finally, the LEDs are set to green for normal movement and forward movement resumes.

You will need the turnLeft block from one of your earlier programs to allow the mBot to make a controlled turn. This type of reuse can be easily achieved if you have created a template starter program that contains the *library* of blocks that you've already built.

Is Opponent In Range

The *isOpponentInRange* block (not shown) is the detectObject block from the chapter on obstacle avoidance with the name changed to be more reflective of the way it is being used. Please refer to Chapter 16 if you need to review the detectObject block implementation.

Charge

The *charge* block sets the LEDs to red and executes the blocks necessary to make the mBot charge at an opponent at maximum speed. Note the use of the or block to terminate the charge movement when the ring edge is detected or the opponent is no longer within attack range.

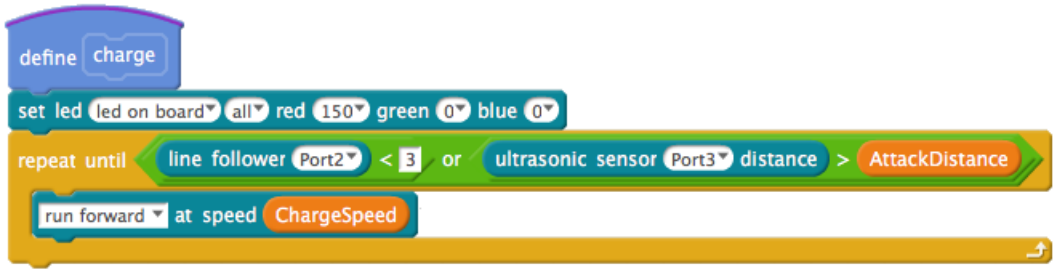


Figure 137: Bashin' At The Basho Charge Block

Testing & Validation

Test Procedure

1. Compile and upload the program.
2. Press the button.
3. The mBot should start moving forward.
4. The mBot should back up when it detects the edge of the ring and turn left before moving forward.
5. The mBot should move in a charge at maximum speed when it detects anything within the specified attack range.
6. The mBot should stop moving after the match time has expired.

You may want to test each custom block as you build it to ensure that it works before you move on to creating the next block.

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBot runs backwards when I want it to go forward
   (a) Switch the motor plugs on the mCore board.
3. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
4. Compilation or Upload failed

(a) Did the program compile without any errors?

(b) Were you able to upload the program to the mBot?

5. There is an error in the program.

   (a) Did you create the variables correctly?

   (b) Did you initialize all of the variables?

   (c) Did you use the proper comparison operators?

   (d) Did you check that your program matches the one in Figures 133 - 137?

6. The mBot does not move faster when charging at an opponent.

   (a) Make sure your batteries are fully charged.

   (b) Make sure the charge speed is different than the normal speed.

7. Why does the mBot drive right by an opponent?

   (a) Depending on the speed of the mBot and the opponent, the criteria for being in range may have been missed if the mBot and opponent moved in and out of range before the Ultrasonic sensor reading was obtained.

Challenges

Adding lights and sound is one way to make the sumo match more interesting.

1. Modify the program to make the LEDs green when the mBot is just moving around and red when the mBot charges at an opponent.

2. Modify the program to make the LEDs flash red when charging at an opponent.

3. Modify the program to make the mBot play a sound or melody when not charging at an opponent.

4. Modify the program to play a different sound or melody when the mBot is charging at an opponent.

5. The current strategy for detecting an opponent relies on random chance as the mBot moves around the ring. Can you devise and implement a better strategy for finding an opponent?

18 | A-Maze-ing!

Mankind's fascination with mazes and labyrinths seem to stretch back as far as we have recorded history. The challenge of finding ones way through a series of twisting passages continues to pose an intellectual challenge whether one is solving the maze on paper or in the physical world.

Mazes vs. Labyrinths

Because precise definitions are important, we need to distinguish between a maze and a labyrinth.

1. A maze is a collection of connected passages with at least one entrance and at least one exit. The goal of a maze is to find the exit.
2. A labyrinth is an area with a single entrance that also serves as its exit. The goal of a labyrinth is to reach a designated point within its bounds.

Program Description

The goal of this exercise is to create a program that allows the mBot to navigate a structure that combines elements of a labyrinth and a maze. This exercise will allow you to use all of the skills and techniques covered in the book to create a program that allows your mBot to solve a maze.

Requirements

The requirements for this program are:

1. The program shall enable the mBot to detect connecting passages.
2. The program shall enable the mBot to execute a maze-solving strategy to locate the designated end point.
3. The maze-solving strategy shall enable the mBot to locate the designated end point regardless of the starting point.
4. The program shall only use the sensors and devices that come with a stock mBot.

Analysis

Maze-Solving Strategies

There are several strategies for solving mazes. However, not all strategies solve all mazes equally well. We also need to be aware that the subset of Scratch that supports the mBot is fairly limited and as such limits our selection of strategy.

For this program, let's use a simple wall-following strategy. It is not the fastest approach, but it will eventually solve the maze so long as the maze does not have any loops.

The wall-following strategy works by having the mBot place an imaginary hand on a wall (right or left, it's your choice) and keeping that hand on the wall. The algorithm works as follows:

1. Check to see if mBot is at designated end of labyrinth.
2. Turn left and check for a wall.
3. If there is no wall, move forward. Otherwise turn back to the right.
4. Check to see if there is a wall straight ahead.
5. If there is no wall, move forward.
6. If there is a wall, turn right and check to see if there is a wall.
7. If there is a wall, turn right. Otherwise, move forward.

Challenges for Maze Solving

There are several challenges associated with successfully creating a maze-solving program.

1. Navigating down the center of the passageways.
2. Detecting walls.
3. Strategy for handling dead-ends.

Challenges with mBlock

The mBlock IDE only supports a limited subset of Scratch blocks and unfortunately neither a list or a string (which might be pressed into service as an array) are supported at this time[24]

Challenges with the mBot

Although you have gathered data for movement and turning, the mBot still suffers from a lack of precision when it comes to moving. The inability to make extremely accurate turns present a particular problem for navigation because the errors become additive over time. Likewise, forward movement errors become additive and you may find that your mBot is running into walls and getting stuck!

Maze Design

The mBot comes with several sensors, but to build a maze-solver will require the maze to be designed and constructed in a fashion that supports the mBot's capabilities. Normally a maze-solving robot uses a sensor array to detect walls or if the maze is a line maze it will use an array of line sensors.

We are limiting ourselves to a stock mBot, so we will have one ultrasonic sensor and one line following sensor. The maze will need to be designed with this limitation in mind. The suggested approach is to use posts to hold walls and to put lines on the floor of the maze (Figure 138).

[24]Remember, this constraint applies to programs directly running on the mBot, not programs run from within the mBlock environment.

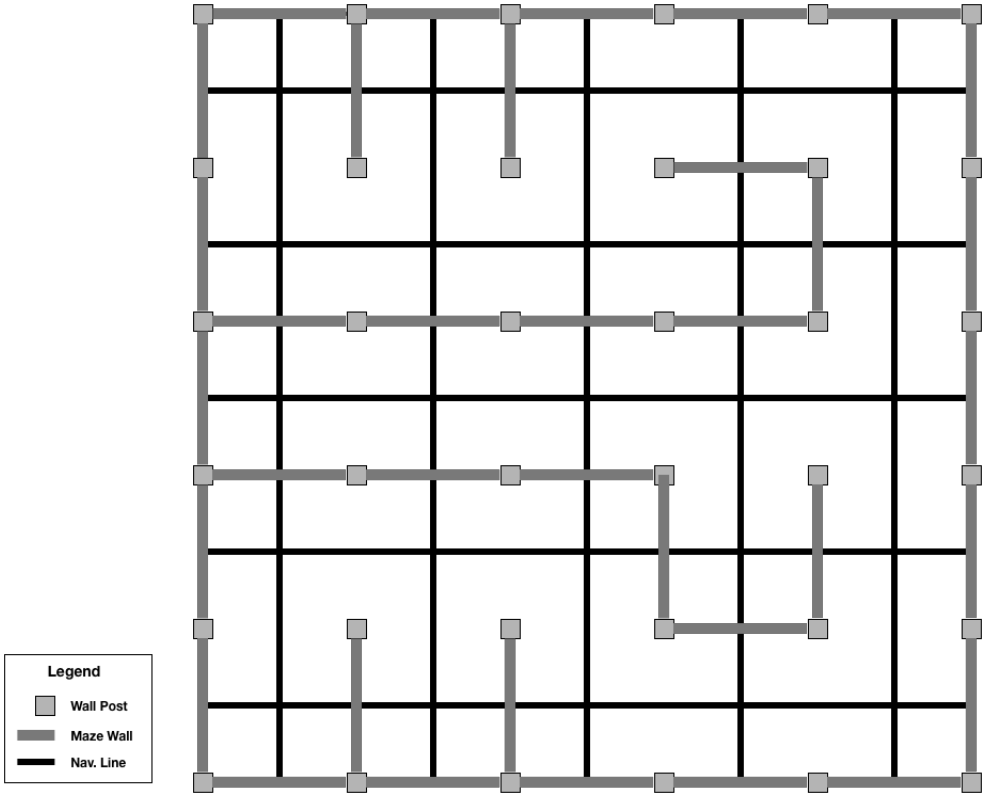


Figure 138: Sample Maze

In the sample maze, you'll note that there is not a designated starting and ending point. The starting can be any square in the maze, but marking the ending point might appear to be a bit tricky. In reality, You can mark the end point of this maze by covering the uppermost vertical line and the two horizontal lines with something of the same color as the maze floor. This has the virtue of allowing you to designate any room as the final room.

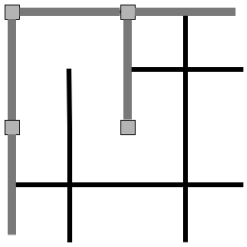


Figure 139: Marking The End of the Maze

Breaking Down the Boulder

You can't solve this problem all at once, so let's break it down into smaller problems. The first problem is navigation. You need to enable the mBot to travel in straight lines and to make accurate turns to the left and right. This is a problem that can be solved using the line following sensor so long as we have lines on the maze floor.

Forward Movement

You have already gathered the data to be able to move a designated distance in a straight line from earlier work, so you can reuse the block for forward movement. You will need to position the mBot so that the axles are centered on the horizontal line and move forward until they are centered on the next horizontal line. You may need to do several runs to get the exact number (and the number will depend on how big your rooms are). You should also refresh your memory on how to make the mBot follow a line.

Turns

While we could use our turn blocks from the past, those depend on timing and as mentioned several times, there will likely be additive errors that may cause the turns to become increasingly inaccurate. A more accurate approach is to make the mBot turn off of the line in the desired direction and then continue turning until both pairs of sensors on the line following sensor are over the line. This approach almost always guarantees a fairly accurate turn. If you need the mBot to continue moving forward after the turn the line following part of the program should correct for any off-center positions.

Detecting the Walls

The walls of the maze are solid, vertical surfaces that lend themselves well to being detected by the ultrasonic sensor. However, the relatively inaccurate movements of the mBot suggest that the mBot should treat anything that is less than the distance from the center of one room to the center of the next room as a wall. This approach will relieve the need to check for a specific wall distance from the mBot.

Dead-End Strategy

The strategy for handling dead ends is straight forward. Let's walk through an example using Figure 140.

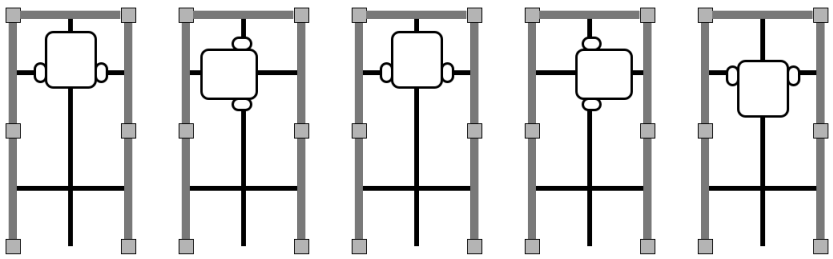


Figure 140: Dead End Strategy

1. Wall detected in front of mBot
2. mBot executes a left turn but detects a wall
3. mBot executes a right turn but detects a wall
4. mBot executes a right turn but detects a wall
5. mBot executes a right turn

While there are many approaches to the dead-end strategy, such as using variables to keep track of what walls were detected, this method keeps the program relatively simple.

Now that the challenges have been thought through, you are ready to move on to design.

Design

The flowchart for the program turns out to be much simpler than one might expect. Figure 141 depicts the design based on the analysis of the problem. The simplicity of the design is possible because we are not tracking any variables for the walls or the path and are simply reacting to pre-programmed patterns.

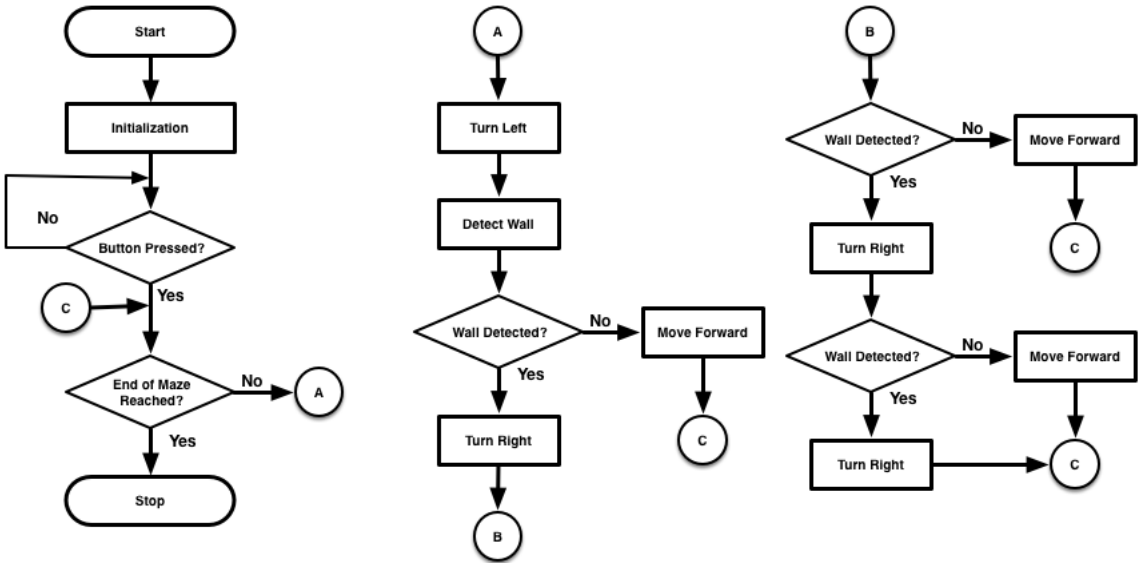


Figure 141: Flowchart for the A-Maze-ing Program

Note that the flowchart accounts for having the mBot wait until the button on it is pressed before it begins movement. The flowchart also makes extensive use of on-page connectors to avoid a spiderweb of overlapping lines.

Program Code

Main Program

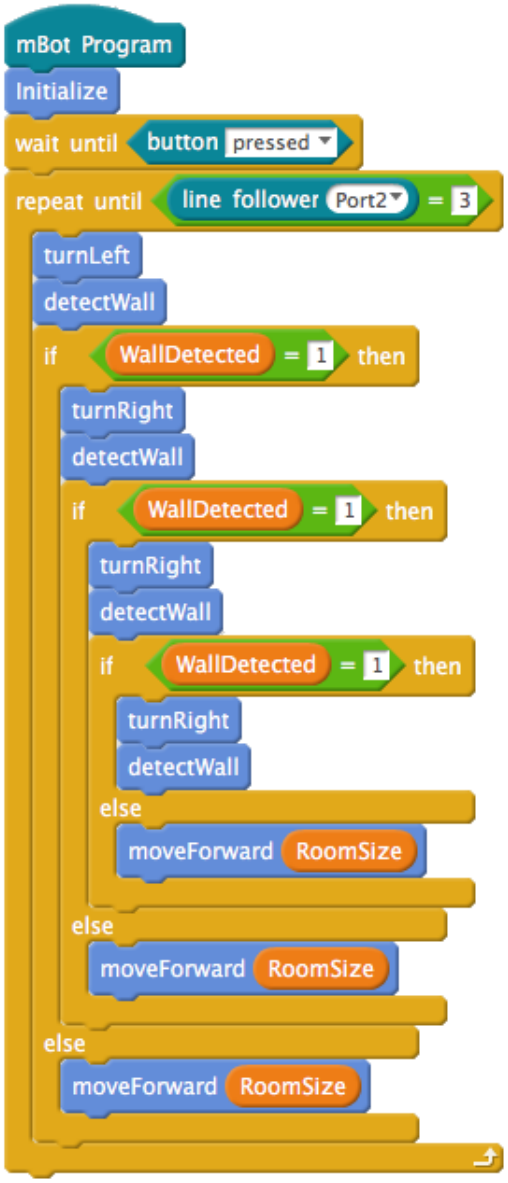


Figure 142: A-Maze-ing Main Program

After calling the *Initialize* block to set up variables and assign them their initial values, the program uses the now-familiar *wait until* block to wait for a button press before beginning any movement. From that point, the program enters a *repeat until* block that executes until it encounters a room marked as the endpoint as indicated by both sets of sensors on the line following sensor being off of the line.

Within this loop, the mBot begins by turning to the left to see if there is a wall. This may seem odd at first, but the program makes no assumptions about the starting position or orientation of the

mBot.

If the mBot detects a wall, it will execute a right turn. If it does not detect a wall, the mBot will execute a forward movement to go to the center of the next room. Note that the mBot will always execute this type of forward movement if a wall is not detected.

This pattern is followed through the rest of the program. If the mBot is in a dead end, it will execute a total of one left turn followed by three right turns to turn itself around. Upon locating an opening, it will execute a forward movement, effectively backtracking the way it came.

Initialize

The *Initialize* block sets the center-to-center size of the rooms (*RoomSize*), the rate of travel (*CmPerSecond*), and the power level (*Speed*) used by the mBot.

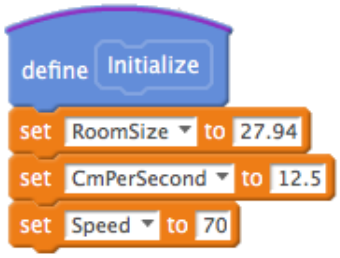


Figure 143: A-Maze-ing Initialization Block

Moving Forward Along the Line

The *moveForward* block calculates the duration of time that the motors should run to cover the distance specified by the *distance* parameter. Next the timer is reset and the motors are set to run forward at the speed set by the global variable *Speed*. The *repeat until* loop will execute until the timer indicates that the *Duration* calculated earlier has been exceeded, at which point the motors are turned off.

Within the loop, the block checks to see if the line sensor indicates that the mBot is veering off of the line and makes an adjustment according to whether or not the mBot is veering left or right. If the mBot line sensor indicates that the mBot is moving along the line, the motors continue to move the mBot forward.

You should note that there is a subtraction of .15 from the CmPerSecond variable. This minor adjustment is due to the inexact movement that results from timing errors. As previously mentioned, the mBot uses toy gearmotors that are not encoded to provide precise movements, unlike stepper motors. The .15 adjustment was arrived at through trial and error and may only apply to the mBot and battery combination the author used for the program. You will need to experiment to find the value that works best for your mBot.

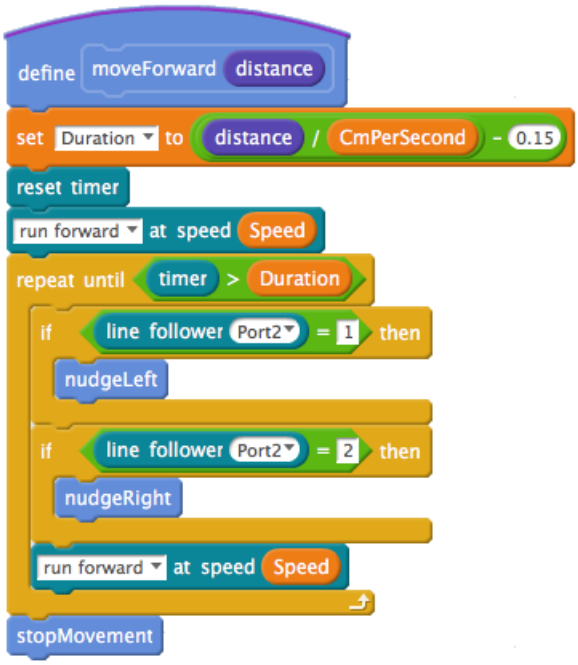


Figure 144: A-Maze-ing Move Forward Block

This program also uses a couple of helper functions that assist the mBot to stay on the line. They give the mBot a slight "nudge" in the right direction by executing a short turn at a fraction of the normal forward movement speed. To ensure that the overall forward movement time is not impacted, these "nudges" will add back their time to the end time for the forward movement.

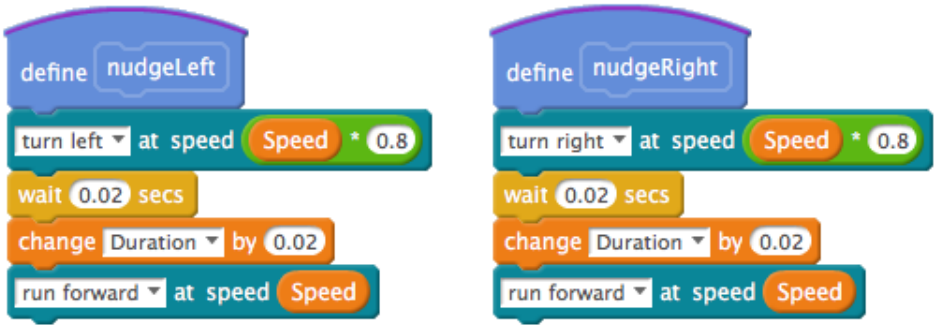


Figure 145: A-Maze-ing nudgeLeft and nudgeRight Blocks

Turning

The *turnLeft* and *turnRight* blocks are identical except for the direction of the turn. Both blocks use a 0.25 second "bump" to get the sensors of the line following sensor off of the line before starting the turn. This action ensures that the turn can happen without any issues. From that point, the block uses a *repeat until* block that exits when both pairs of sensors on the line following sensor are over the line and the motors stop. Until that point, the mBot keeps making its turn.

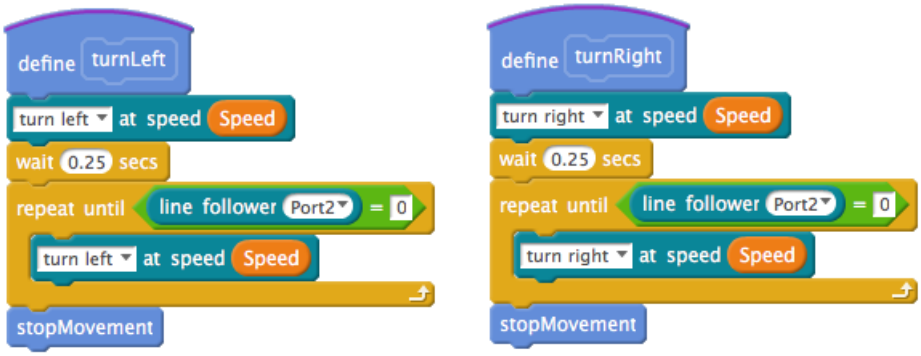


Figure 146: A-Maze-ing turnLeft and turnRight Blocks

Note that the *stopMovement* block only contains a *run forward* at zero speed block and is not illustrated here.

Detecting the Wall

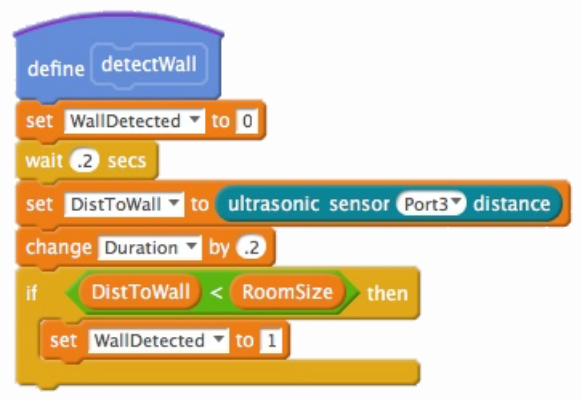


Figure 147: A-Maze-ing detectWall Block

The *detectWall* block is essentially the detectObject block from Chapter 16. However, you should note that the .2 second settling time is added back to the duration so that the timing for movement is not adversely affected.

Testing & Validation

This program involves a lot of movement and there will likely be additive errors due to the inaccuracies associated with the control of the motors. Be prepared to test everything, but be smart about your testing. For a program of this size, you will need to test each block before testing the entire program.

Test Procedure

1. Test *moveForward* block to ensure that accurate forward movements are occurring.
2. Test *turnLeft*/*turnRight* blocks to ensure that accurate turns are occurring.

3. Test ability to stop at the designated endpoint of the maze.
4. Test main program to see if mBot is solving the maze.

Troubleshooting & Debugging

Let's look at what might have potentially gone wrong by narrowing down the potential points of failure into a checklist and work through them.

1. The mBlock IDE is not connected to the mBot
   (a) Is the cable is connected to the mBot and the computer?
   (b) Did you click the Connect menu item and choose the correct serial port?
2. The mBot runs backwards when I want it to go forward
   (a) Switch the motor plugs on the mCore board.
3. The mBlock IDE cannot connect to the mBot
   (a) Is the mBot turned on?
   (b) Have you upgraded the firmware on the mBot from the Connect menu item?
   (c) Did you install the Arduino driver for the mBot?
4. Compilation or Upload failed
   (a) Did the program compile without any errors?
   (b) Were you able to upload the program to the mBot?
5. There is an error in the program.
   (a) Did you create the variables correctly?
   (b) Did you initialize all of the variables?
   (c) Did you use the proper comparison operators?
   (d) Did you check that your program matches the one in Figures 142 - 146?

Chapter 19

You Made It!

Well, here we are at the end of the book and congratulations are in order. Whether you are a complete novice to programming and robotics or an expert who has had the patience to work through to the end, I want to thank you for coming on this journey.

It is important to realize that this is just a way station, not an endpoint for your journey into robotics. You have had the opportunity to do some incredible things with the mBot and Scratch and I hope that you will continue to build upon your experience by designing programs of your own and posting them on the Makeblock forum or on YouTube for all to see. You can also extend the capabilities of your mBot by purchasing additional modules from Makeblock.

In the next volume of the series, we will revisit many of the programs from this book but we will be writing the programs in the native language of the Arduino, C/C++. While the language will look different, you will be familiar with the concepts. You will also discover that there are many more things that you can accomplish.

Regardless, I wish you well as you continue your journey!

Appendix A | Flowcharts

Flowcharting As A Design Tool

Flowcharts provide a visual map for your program logic. Why do them?

- Flowcharts allow you to map out the logic for your program before you write the code.
- Flowcharts help you to organize your thinking.
- Flowcharts help identify repeating logic that can be encapsulated in a custom block (also called a function or method in other languages).
- Flowcharts allow you to quickly explain your program logic to others.

Flowcharting Symbols

Flowcharts use the following symbols:

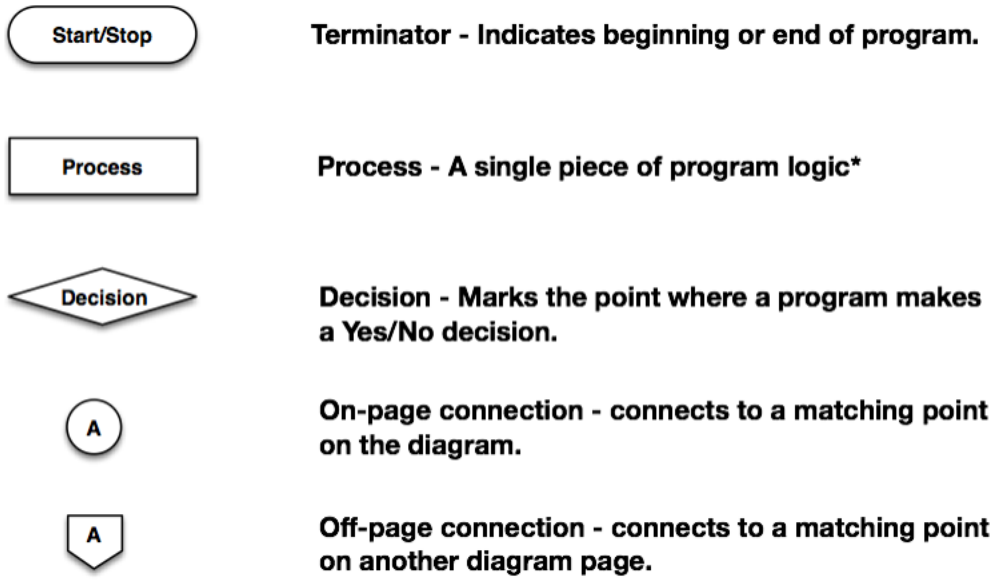


Figure 148: Flowchart Symbols

Flowcharting Connections

Flowcharts use lines to connect the symbols and arrowheads on the lines to indicate the direction of program flow.

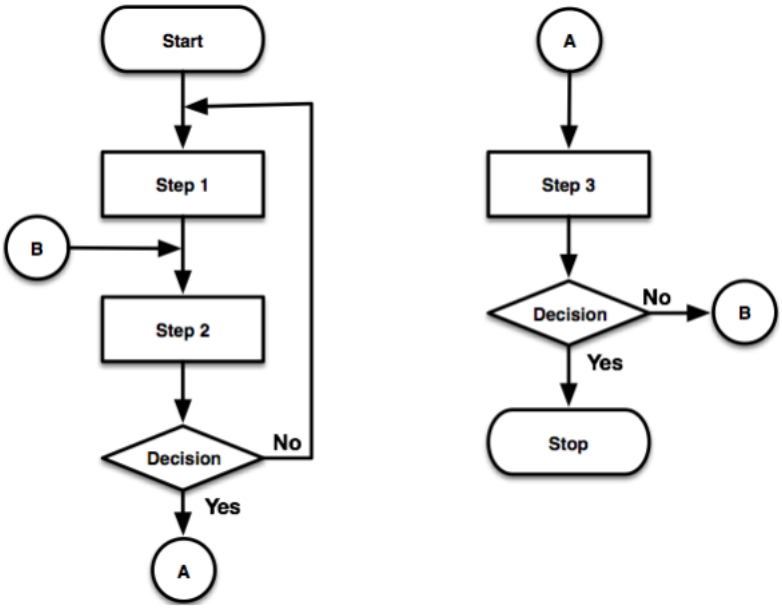


Figure 149: Flowchart Connections

Appendix B | mBot Blocks

The blocks described in this appendix appear when mBlock's Board menu has the mBot selected. If other boards are selected, such as the Orion board, many of the blocks shown here will not be displayed.

Hat Blocks

The *mBot Program* block marks the beginning of a program that can be compiled and run directly on the mBot.

The *when button [event]* is used to mark the beginning of a subroutine when running the program from within mBlock. This block does not generate code and will have no effect when running the program directly on the mBot.

The events for this block are:

Event	Description
pressed	Button is currently pressed
released	Button has been released

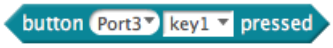
When running in the mBlock environment, this block will execute its contents when the button is pressed or released depending on the specified state. In some cases, you may want to have subroutines tied to each of the states.

Event Blocks

The *button [event]* block is similar to the *when button [event]* hat block. It will return a true or false value depending on the state of the green button on the mBot and the type of event you are using. The events for this block are:

Event	Description
pressed	Button is currently pressed
released	Button has been released

Control blocks (*if-then, if-then-else, etc.*) use this block to trigger the execution of the code contained within them.



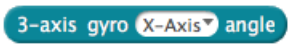
The *button on [port][key] pressed* block is used by the four button module sold by Makeblock. The block watches the designated port and returns true if the selected button [1-4] has been pressed.

The *limit switch [port][slot]* block tracks the current state of an attached microswitch, returning true if the switch has been activated and false if not activated. The *port* parameter indicates the port where the switch is connected. The *slot* parameter is used to designate which slot on the Me-RJ25 adapter is plugged into. This block is used by the control blocks to determine whether or not to execute the code within the control block.

The *[ir remote [button] pressed* block tracks the current status of the selected button, returning true if the button has been pressed or false if it has not. The *button* parameter supports buttons 0-9, A-E, the arrow keys, and the settings button (gear button). This block is used by the control blocks to determine whether or not to execute the code within the control block.

The *touch sensor [port]* block tracks the current status of a touch sensor connected to the selected port returning true if the touch sensor microswitch is closed and false if it is not. This block is used by the control blocks to determine whether or not to execute the code within the control block.

Reporter Blocks



The *3-axis gyro [axis] angle* block returns the angle in degrees for the specified axis (X/Y/Z).

The *compass sensor [port]* block will return a value indicating the number of degrees of rotation from north (north = 0).

The *flame sensor [port]* block returns a value from the flame sensor. The block returns true if a flame has been detected and false if a flame is not detected.

The *gas sensor [port]* block is used to measure concentration of flammable gases. The block will return true if a higher than normal concentration of flammable gases is detected and false if it is not.

The *humiture sensor [port][setting]* returns the current humidity level or the temperature based on the selected setting. Temperature is return in degrees Celsius and humidity is returned as a percentage.

The *joystick [port][axis]* block returns the position of the joystick attached to the selected port. The axis parameter may be set for the X-axis or the Y-axis and will return values in the range -500 to 500. Negative values for the X-axis indicate the joystick is left of center. Negative values for the Y-axis indicate the joystick is below center.

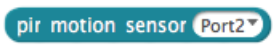
The *light sensor [port]* block returns the current light level. The port setting may be *light sensor on board* for the light sensor built into the mBot or to port 3 or 4 for additional light sensors that may be attached. A zero value represents absolute darkness and all greater values indicate some degree of light.

The *line follower [port]* block returns a value based on the status of the two pairs of infrared emitter/receivers.

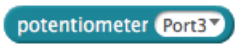
Value	Description
0	Both sensors on line
1	Left sensor on line
2	Right sensor on line
3	Both sensors off line

The *port* parameter indicates where the line following sensor is attached.

The *mBot's message received* block returns a string received via the infrared receiver on the mBot.



The *pir motion sensor [port]* block is used to detect changes in infrared radiation caused by a human or other heat-emitting body. The block will return true if something is detected and false if it is not.



The *potentiometer [port]* block reports the current value of the potentiometer module attached to the designated port. The values will range from 0-1024.

The *sound sensor [port]* block returns the level of sound detected. The *port* parameter identifies where the sound sensor is installed.

The *temperature [port] [slot] C°* block returns the temperature in Celsius as measured by the sensor. The *port* parameter designates the port where the sensor is attached, and the *slot* parameter designates the slot on the Me-RJ25 adapter module where the sensor is attached.

The *timer* block returns a time based on the hardware timer on the mCore board. This time is the number of seconds elapsed in seconds since the mBot was turned on.

The *ultrasonic sensor [port] distance* block returns the distance in centimeters to the nearest detected object. As of mBlock 3.3.5, the sensor returns 400 if it cannot detect anything. Earlier verisons returned a zero if nothing was detected.

Stacking Blocks

The *[direction] at speed [power level]* block controls the direction and speed of the mBot motors. The *direction* parameter determines whether the mBot is moving forward, backward, or turning left or right at the rate indicated by the *power level* parameter.

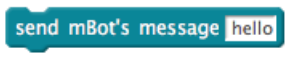
A positive *power level* will cause the mBot to move forward. A negative value will cause the mBot to move backwards.

The turns use a tank-style turn that negates the *power level* value for the inner wheel on a turn while applying the same positive power level to the outer wheel.

The *play tone on note [frequency] beat [duration]* block plays the specified *frequency* for the listed *duration*. See Appendix D: Musical Note Values for a list of the available note frequencies.
The *duration* has the following default values although you are free to type the value of the desired frequency directly into the parameter slot.

Label	Length
Zero	0 ms
Eighth	125 ms
Quater	250 ms
Half	500 ms
Whole	1000 ms
Double	2000 ms

The *reset timer* block sets the internal counter to the current time of the hardware timer.



The *send mBot's message [string]* block sends the *string* over the infrared transmitter.

The *set 7-segments display [port] number [value]* block enables the display of a number to a 7-segment display attached to the port designated by the *port* value. The range for the displayable numbers are -999.5 through 9999.5.

The *set camera shutter [port] as [state]* block enables a program to control the camera module on the port specified by the *port* parameter. This block has the following states:

Value	Description
Press	Open camera shutter
Release	Close camera shutter
Focus On	Activate focus
Focus Off	Deactivate focus

The *set led [port][device] red [rvalue] blue [bvalue] green [gvalue]* block enables a program to turn LEDs on or off in a variety of colors.

The onboard LEDs for the mBot are addressable by setting the *port* parameter to *led on board*. LEDs connected to other ports may be addressed by specifying the port where those devices are connected.

The *device* parameter enables the selection of one or more LEDs to control. If the parameter is set to *all*, then the command will be sent to all connected LEDs on the specified port. The program can also specify a specific LED to control.

The colors are defined by a combination of red-green-blue (RGB) values where each individual value can range from 0-255. Setting all values to zero results in black, effectively turning the LEDs off. Likewise, setting all values to 255 results in a white color.

The *set led strip [port][slot][device] red [rvalue] blue [bvalue] green [gvalue]* block enables a program to control the LEDs on an LED strip attached to the port specified by the *port* variable. The *slot* parameter refers to the slot on the Me-RJ25 adapter module where the LED strip is attached.

The *device* parameter enables the select of one or more LEDs to receive the command.

As with the *set led* block, the colors are defined using RGB values, with all zeroes turning off the LED or LEDs.

The *set light sensor [port] led as [state]* block enables the program to turn the LED of an off-board light sensor on and off. This block does not interact with the mCore onboard light sensor. The *port* parameter specifies the port where the light sensor connects to the mCore board.

The *set motor [motor] speed [power level]* block enables a program to have control over each of the mBot's motors. The *motor* parameter specifies which motor will be affected and the *power level* parameter determines the speed and direction that the motor will turn.

Note that a positive number will make the motor move in a forward direction and a negative number will make the motor move in a backward direction.

The *set servo [port][slot] angle [value]* block enables a program to control the movement of a servo connected to the mBot on the port designated by the *port* parameter.

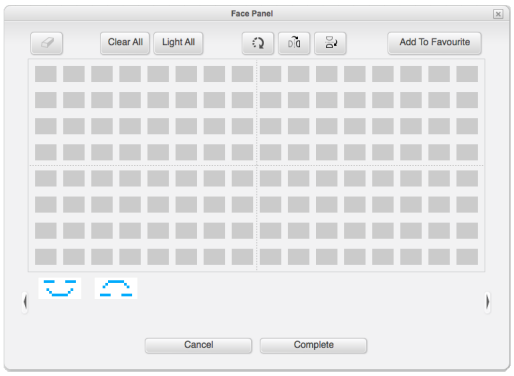
The *slot* refers to the slot on the Me-RJ25 adapter module where the servo is attached. The angle *value* accepts a range of 0-180 degrees which corresponds to most non-continuous servos.

The *show drawing [port] x: [xvalue] y: [yvalue] draw []* block enables a program to display a drawing on the faceplate add-on module.

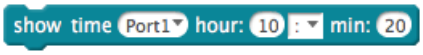
The *port* value defines where the faceplate is connected to the mBot.

The X/Y coordinates (*xvalue* and *yvalue]* determine the upper left hand corner of the drawing on the faceplate.

The input slot after **draw:** brings up a dialog that allows the developer to create a drawing by setting/unsetting blocks on the dialog.



The *show face [port] x: [xvalue] y: [yvalue] characters: [string]* block enables a program to display letters or numbers on the add-on faceplate display. The *port* parameter specifies where the faceplate display is connected to the mBot. The X/Y values determine the upper left corner where the contents of the *string* parameter will be displayed.



The *show time [port] hour: [hour] [separator] min: [minutes]* block enables a program to display the time on the add-on faceplate module. The *port* parameter specifies where the faceplate is connected to the mBot. The *hour* and *minutes* values are used to display the time and the *separator* allows the program to insert a colon, space, or other separator between the hours and minutes.

The *stop tone* block turns off the buzzer on the mBot. With the advent of the *play tone on note* block, the *stop tone* has less use and may be deprecated in the future for the mBot portion of the Robots palette.

Appendix C | Best Practices

This appendix contains a brief list of best practices that will assist in creating solid programs. The biggest key to success, however, is consistency. Consistency in each step of the process will ensure that your software development occurs in an orderly, least time-consuming fashion.

Process

1. Make sure that you have a clear understanding of the problem to be solved.
2. Make sure that you have a consistent process that you are following *consistently*.
3. Keep a journal that captures your thoughts and discoveries over the course of the project.
4. Be sure to review your journal after the project is over to see if there are ways to do things better on the next project.

Requirements

1. Make sure that each requirement covers one thing that can be tested. If you discover that you have a requirement that is doing multiple things, break it up into smaller chunks that can be tested.
2. Be precise in your wording.

Analysis

1. Take the time to carefully read the requirements.
2. Make sure that you understand the problem to be solved.
3. Identify all of the internal and external dependencies and relationships for the problem you are solving.

Design

1. Design in the big (top-down) for the initial cut.
2. Design in the small (bottom-up) for the next cut.
3. Always break large problems down into smaller ones.
4. Make sure that each custom block does one thing.
5. Be thinking about how each custom block can be tested

6. Take the time to make diagrams to walk through your design.
7. Explain your design to someone else to see if they can see something you can't.
8. Be consistent in your naming practices.

Implementation

1. Be sure to use an initialization block for all initializations.
2. Don't Repeat Yourself (DRY). If you are repeating the same code in many places, you may want to put it into it's own custom block.
3. If you find that you are repeatedly recreating the same blocks consider building a *template* project (called a library in other languages) with all of the blocks
4. Use the simplest possible way to achieve the desired functionality.
5. Use comments to convey intent or complicated details.

Testing

1. Always test the *golden path* to ensure the desired functionality works under normal circumstances.
2. Test as many side cases as possible.
3. Automate testing where possible.

Appendix D | Musical Note Values

Note	Value	Note	Value	Note	Value	Note	Value
B0	31	A2	110	G4	392	F6	1397
C1	33	B2	123	A4	440	G6	1568
D1	37	C3	131	B4	494	A6	1760
E1	41	D3	147	C5	523	B6	1976
F1	44	E3	165	D5	587	C7	2093
G1	49	F3	175	E5	659	D7	2349
A1	55	G3	196	F5	698	E7	2637
B1	62	A3	220	G5	784	F7	2794
C2	65	B3	247	A5	880	G7	3136
D2	73	C4	262	B5	988	A7	3520
E2	82	D4	294	C6	1047	B7	3951
F2	87	E4	330	D6	1175	C8	4186
G2	98	F4	349	E6	1319	D8	4699

Appendix E | Arduino Uno Specifications

The Makeblock mCore board for the mBot is based on the Arduino Uno[25].

Technical specs

Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
Length	68.6 mm
Width	53.4 mm
Weight	25 g

Figure 150: Arduino Uno Specifications

[25] http://goo.gl/VcV68F

Appendix F | Resources

Books

Scratch-related Books

Learn to Program with Scratch by Majed Marji

mBlock Kids Maker Rocks with the Robots link

Scratch 2.0 - The Adventures of Mike link

Scratch Programming in easy steps by Sean McManus

Super Scratch Programming Adventure! by the LEAD Project

Robotics-related Books

The Robotics Primer by Maja J. Matarić

Web Sites

Makeblock Forums	link
mBot Introduction	link to Makeblock Learn site
Official Scratch Site	link
Scratch Resources	link
Scratch Wiki	link
Redware	link
YouTube	link to Scratch videos
	link to mBot videos

Materials

Degree Wheel

Figure 151: Degree Wheel

Movement Table

Power	1 sec	2 sec	3 sec	4 sec	5 sec	Mean
100						
125						
150						
175						
200						
225						
250						

360° Turn Table

Power	Time
100	
125	
150	
175	
200	
250	