NETWORKING WITH THE MICRO:BIT

WRITTEN BY

Cigdem Sengul

&

Anthony Kirby

Authors: Cigdem Sengul and Anthony Kirby

https://microbit.nominetresearch.uk/networking-book/

Acknowledgements:

"BBC", "micro:bit" and the micro:bit emojis are trademarks of the BBC

CONTENTS

Preface 8

About the book 8

About the authors 9

Acknowledgements 9

Resources & updates 9

Outline 9

1 Communication over Wires 11

1. Introduction 11

1.2 What you'll need 11

1.3 Background 11

1.4 Programming: A Simple Heart Transfer 12

1.5 Task 1: Watch the "Simple Heart Transfer" 13

1.6 Task 2: Connect your micro:bits and test telegraph 13

1.7 Task 3: Test "Simple Heart Transfer" Hex files 14

1.8 Task 4: Program a heart transfer 14

1.9 Extended activity 15

1.10 Problems 15

1.11 Resources 15

2 Wireless and Broadcast Communication 17

2. Introduction 17

2.2 What you'll need 17
2.3 Background 17
2.4 Programming: Receiving and sending broadcast messages 19
2.5 Task 1: Configure your radio 19
2.6 Task 2: Receive a broadcast message 20
2.7 Task 3: Send a broadcast message 20
2.8 Extended activity 21
2.9 Problems 21
2.10 Resources 21

3 Group communication 23
3. Introduction 23
3.2 What you'll need 23
3.3 Background 23
3.4 Programming: Creating groups and messaging within groups 24
3.5 Task 1: Create groups 25
3.6 Task 2: Send and receive messages 25
3.7 Extended activity 25
3.8 Problems 25
3.9 Resources 26

4 Game 1: Shakey Donkey 28
4. Introduction 28
4.2 What you'll need 28
4.3 Programming: Playing Shakey Donkey 28
4.4 Problems 30

5 Unicast communication: One to One 32
5. Introduction 32
5.2 What you'll need 32
5.3 Background 32
5.4 Programming: Sending and receiving unicast messages 34
5.5 Task 1: Configure your radio 34
5.6 Task 2: Design your header 34
5.7 Task 3: Create your packet and send 35
5.8 Task 4: Receive a packet 35
5.9 Challenge: Filter senders 35

5.10 Extended activity **35**
5.11Problems **36**
5.12 Resources **36**

6 Two-way unicast 38

6. Introduction **38**
6.2 What you'll need 38
6.3 Background **38**
6.4 Programming: Ping 41
6.5 Task 1: Prepare for unicast 41
6.6 Task 2: Send a Ping 41
6.7 Task 3: Receive a Ping 41
6.8 Task 4: Receive a Pong and calculate round-trip-time 41
6.9 Exercises **41**
6.10 Problems **42**
6.11 Resources **42**

7 Game 2: Rock, paper, scissors over the radio 44

7. Introduction **44**
7.2 What you'll need 45
7.3 Programming: Rock, paper, scissors **45**
7.4 Task 1: Start with the simple game 45
7.5 Task 2: Hands over the radio with unicast 45
7.6 Task 3: Fill the table of rules 45
7.7 Task 4: Play the game 46
7.8 Exercises **47**
7.9 Problems **47**
7.10 Resources **47**

8 Handling Errors: Retransmissions 49

8. Introduction **49**
8.2 What you'll need 49
8.3 Background **49**
8.4 Programming: Retransmissions **51**
8.5 Task 1: Create packet errors 51
8.6 Task 2: Send a sequence of messages 51
8.7 Task 3: Retransmit by default 52

8.8 Extended activity **52**
8.9 Problems **53**
8.10 Resources **53**

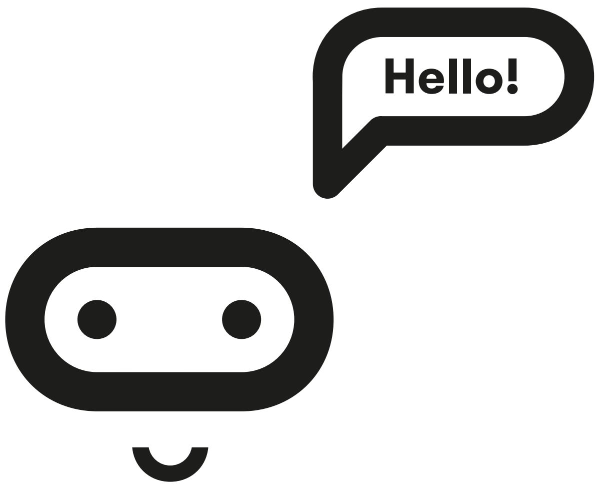
9 Handling Errors: Acknowledgements 55

9. Introduction **55**
9.2 What you'll need 55
9.3 Background **55**
9.4 Programming: Stop and Wait! **58**
9.5 Task 1: Design your data and ACK packets 58
9.6 Task 2: Timeout and retransmission 58
9.7 Task 3: Testing the reliability of Stop-and-Wait 58
9.8 Extended activity **59**
9.9 Problems **59**
9.10 Resources **59**

10 Game 3: Battleship over Radio 61

10. Introduction **61**
10.2 What you'll need 62
10.3 How the game works 62
10.4 A sample game 63
10.5 Programming: Battleship **64**
10.6 Task 1: Initialization of the game 64
10.7 Task 2: Firing a shot 65
10.8 Task 3: Receiving a shot 65
10.9 Task 4: Receiving the shot result: "Hit" or "Miss" 65
10.10 Extended Activity **66**
10.11 Problems **66**
10.12 Resources **67**

Index 70



PREFACE

About the book

This book presents a series of activities to teach the basics of computer networks. While you will not learn all aspects of computer networking, we hope that it will serve as a good starting point.

To network micro:bits, we use custom micro:bit radio to radio communication. When one hears the word radio, what comes to mind is the radio that blasts out tunes from your favourite radio broadcasting channel. But, a radio, or a radio transceiver (transmitter/receiver), is used in communications to generate and receive radio waves that contain information such as audio, video or digital data. And all micro:bits have built-in radios[1].

Each chapter presents interesting challenges in radio communications and networking with micro:bits. After every few chapters, there's an exciting game to look forward to! In the programming sections, you will use the JavaScript Blocks Editor at https://makecode.microbit.org/ to develop solutions to overcome those challenges[2].

Writing this book, we have assumed no knowledge of radio communications or networking.

However, we assume that you have written programs with a micro:bit. For example, we expect that you are familiar with variables, if-then-else statements, and loops. The activities in each chapter will provide ample opportunity to put this knowledge into practice.

[1]The CPU on the micro:bit is a Nordic Semiconductor nRF51822 and contains a built-in 2.4GHz radio module. This radio can be configured to run Bluetooth Low Energy (BLE) protocol but in this book, we will use the simpler micro:bit to micro:bit communication.

[2]This version of the book uses JavaScript Blocks Editor; we are also working on a MicroPython version.

About the authors

Cigdem and Anthony are researchers, which means we work on new ideas and products. We work for a company called Nominet, which runs the part of the Internet that controls how names (like www.bbc.co.uk) are used when people, computers, or devices like tablets or smartphones connect to other computers over the Internet. We're very excited to have the opportunity to work with micro:bits and the Micro:bit Foundation.

Understanding how computers talk to each other is something that we think is important, which is why we wrote this book! We've enjoyed designing the tasks and challenges in the book, and we hope you do too.

Anthony & Cigdem

Acknowledgements:

- Thank you to the BBC and creators of the micro:bit, and to Zach Shelby and Jonny Austin of the Micro:bit Foundation for your support.
- Thank you to David Whale (@whaleygeek) for all your help and inspiration.
- Thank you to James Burgin and Anna Adolphson at Nominet, for help with the videos and graphical layout, and Alistair Braden for review & suggestions.
- Special thanks to Adam Leach, director of our research team at Nominet, for giving us the opportunity to work on this.

Resources & Updates

Updates to this book, alternative formats, and links to online versions are available at https://microbit.nominetresearch.uk/networking-book/

Additional teaching resources (lesson plans for each chapter) will also be published here.

Outline

Communication over wires

This chapter is an introduction and a fun demonstration of networking. Micro:bits can communicate when connected with wires. Via wires, you will send images between micro:bits.

Wireless and broadcast communication

You will start using radio communication in this chapter and learn about broadcast communication. With broadcast communication, one micro:bit can send messages to many other micro:bits. But, be cautious! If all micro:bits do that, it's like everyone is speaking at once.

Group communication

By forming small groups, you will send to and receive from a limited number of micro:bits. This is more manageable than broadcast. But, selecting a unique identifier for your group will be an interesting challenge.

Game 1: Shakey Donkey

This is a game that uses the micro:bit radio. See whether you can figure out how to play the game, and how it works.

Unicast Communication

Broadcast and group communication are fun. But sometimes you want to talk to only one person. This is called unicast communication. To do this, you discover that you will need a unique identifier for your micro:bit.

Two-way Unicast

It's no use talking with somebody if you don't get a response back. In this chapter, you will program your micro:bit to send a message and to get a reply. Also, you will work out how long it takes for a reply to come back. Doing this, you will also program one of the most important tools used in the Internet: Ping.

Game 2: Rock-Paper-Scissors over Radio

This is not like the traditional Rock-Paper-Scissors game. It works over the radio!

Handling errors: Retransmissions

Nothing is perfect, especially radio communication. What happens if your message gets lost on the way? In this chapter, you will test methods for dealing with message loss. For instance, does it help if you send your messages more than once? This is called a retransmission.

Handling errors: Acknowledgements

It's a waste to retransmit if the other side already received the message! The receiver needs a standard reply (or an acknowledgement) to avoid this. At the sending side, if you do not receive an acknowledgement, you can assume that your message wasn't received. In this chapter, you will test how well acknowledgments work to improve reliability.

Game 3: Battleship over Radio

You have come far. Now you are ready for another classic game! You will write a version of the famous Battleship game using your micro:bits. Your experience with radio communication and networking will help you along the way.

Let's start!

1. COMMUNICATION OVER WIRES

1. Introduction

Everything is connected nowadays! Computers and devices connect to each other to form networks. And these networks connect to form the Internet. When we say *computers* or *devices*, these can be anything from a traditional laptop to a cellphone, to a washing machine, to a humidity sensor. Of course, it can also be your micro:bit. More and more, the Internet is becoming an *Internet of Things.*

In this chapter, you will form your own network using wires to connect two micro:bits. Doing this, you will learn:

- **the concept of a *communications medium*, and *signals***
- **the concept of *binary* and *bit***
- **the concept of a network**

1.2 What you'll need?

- **2 micro:bits**
- **4 crocodile clip leads**
- **1 battery holder, and 2 AAA batteries**
- **1 teammate**

1.3 Background

For two micro:bits to be able to send messages to each other, they somehow need to be connected,whether by wires or wirelessly - we call this a *communications medium*.

Definition 1. — Communications medium.
A communication medium is the physical path over which a signal is transmitted.

A message could be a string like "Hello", a number like "9", or an icon image. The micro:bits convert each message to a *signal* to send it over the communications medium.

Definition 2. — Signal.
Signals are the electromagnetic voltages or waves transmitted on a physical wired or wireless medium.

For example, take the case when we say "Hello" into a landline telephone. The telephone handset converts the sounds into an electrical voltage signal. Then, this signal is transmitted to the receiving telephone by wires; and at the receiver, it is converted back into sound.

Exercise 1.
What is the wireless physical medium that makes radio communication possible?

Computers, and also your micro:bit, cannot process signals without converting them to binary data: 0s and 1s. Also, the binary data processed by computers need to be converted into signals before they can travel a communication medium.

Definition 3. — Bit.
A bit is the smallest unit of data in a computer. It is like an atom. A bit can be either a 1 or a 0.

A group of 8 bits is a *byte*. Table 1 shows other example groupings. By connecting computers or any device through different communications mediums, we create *networks*.

Name	Size
Byte (B)	8 bits
Kilobyte (KB)	1024 bytes
Megabyte (MB)	1024 kilobytes
Gigabyte (GB)	1024 megabytes
Terabyte (TB)	1024 gigabytes

Table 1: Groupings of bits.

Definition 4. — Network.
A computer network is a collection of computers or devices, which are connected to communicate with each other. In a computer network, there are at least two computers. Two or more networks can connect to form a larger network: a network of networks. Internet is a massive network of networks!

In this chapter, you will create a network of two micro:bits, connected via wires.

1.4 Programming: A Simple Heart Transfer

In this section, you will connect two micro:bits via wires. You will send a Heart icon from one micro:bit to another. Figure 1.1 shows how a heart icon should look like on the micro:bit display[1]. This activity is best done with a teammate. In the following, you will go through four tasks to program your micro:bits.

[1]This image is by micro:bit Educational Foundation at www.microbit.org

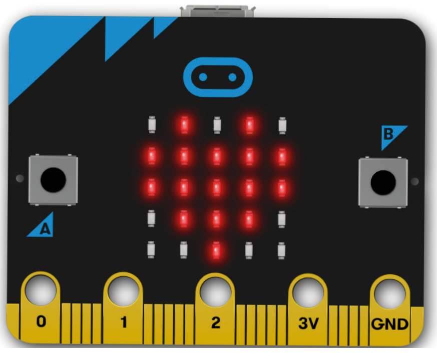


Figure 1.1: Micro:bit displaying a heart icon.[1]

1.5 Task 1: Watch the "Simple Heart Transfer"

Description: We have created a video to show how your connections and program should work in this activity.

See the video at https://microbit.nominetresearch.uk/networking-book/simple_heart_transfer.html

Instruction: Watch the Simple Heart Transfer in the video.

Important: Do not skip this task. It will help you to test whether the files you downloaded for Task 2 work. It will also help you to write your program for this chapter.

1.6 Task 2: Connect your micro:bits and test telegraph

Description: You will connect your micro:bits using wires, and use a program to check the connections. You can follow the instructions below, or there's more detailed step-by-step instructions in the micro:bit telegraph activity[2] on the micro:bit website.

Instruction: Using crocodile clips, connect the 3V pin between the two micro:bits, and connect the GND pins. Then connect pin 1 on one micro:bit to pin 2, and vice-versa. Be careful to get the crocodile clip connections right: two of the wires connect straight (3V→3V and GND→GND) but the other two cross over (1→2 and 2→1).

See Figure 1.2 for an example, and look at the colours carefully (you don't need to use the same colours of course, but they must make the same connections).

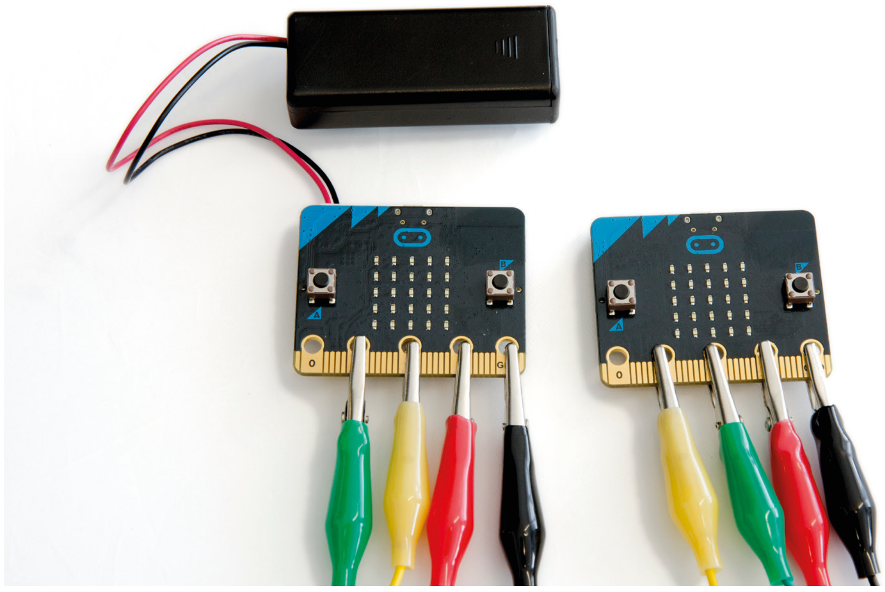
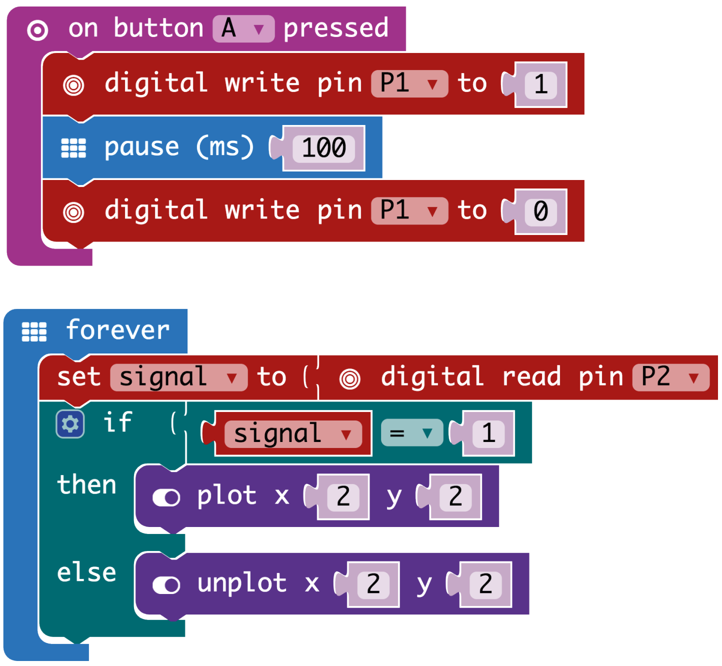


Figure 1.2: Wiring micro:bits. Two of the wires connect straight (3V→3V and GND→GND) but the other two cross over (1→2 and 2→1).

[2]https://www.microbit.co.uk/td/lessons/telegraph/activity

To test, use the program from Figure 1.3; press button A on each micro:bit and check that the LED illuminates on the other one. You will use the blocks from the Pins menu. This menu is under Advanced. Click on the More link to see all the options.



```
on button A pressed
    digital write pin P1 to 1
    pause (ms) 100
    digital write pin P1 to 0

forever
    set signal to digital read pin P2
    if signal = 1
    then
        plot x 2 y 2
    else
        unplot x 2 y 2
```

Figure 1.3: Telegraph program. Pressing button A sends a signal to the other side using Pin 1. The receiver micro:bit listens on Pin 2 to check if a signal is received. If there is a signal, it lights up the (2,2) pixel on the display.

1.7 Task 3: Test "Simple Heart Transfer" Hex files

Description: We provide two files at https://microbit.nominetresearch.uk/networking-book/microbit1_wired_simpleheart_secret.hex. and https://microbit.nominetresearch.uk/networking-book/microbit2_wired_simpleheart_secret.hex for you to test how the final program should work. These files will run on your micro:bits, but you will not be able to display the code using the JavaScript Blocks editor.

Instruction: Download the Simple Heart Transfer code into your micro:bits. There are two different hex files for micro:bit 1 and micro:bit 2. Test the program by tilting your micro:bits and checking when the heart icon is displayed.

1.8 Task 4: Program a heart transfer

Description: In this task, you will program your micro:bits to get a similar behaviour to what you observed in the Tasks 2 and 3. To do this task, you will need to think about the following questions:

1. Which input will the micro:bits react to in your program?
2. How do the microbits send data to each other?
3. Hint: Do you think they are sending each other an actual Heart icon?

Instruction: For question 1, look at the options under the JavaScript Blocks editor Input menu. For question 2, use the example Telegraph program in the Figure 1.3. For question 3, here is another big hint.

Hint: Assume micro:bit 2 knows that it will be receiving a Heart icon from micro:bit 1.

Program your micro:bit 1 so that:

1. It displays a heart icon until it is tilted over the micro:bit 2.
2. When tilted over micro:bit 2, it sends a pulse to micro:bit 2 over the correct pin.
3. When micro:bit 1 receives a pulse on its correct pin, it displays a heart icon .

Program your micro:bit 2 so that:

1. It displays a heart icon when it receives a pulse on its correct pin.
2. When tilted over micro:bit 1, it sends a pulse to micro:bit 2 over the correct pin.

1.9 Extended activity

Exercise 2.
Watch the video at https://microbit.nominetresearch.uk/networking-book/pixel_heart_transfer.html. Based on this video, discuss with your teammate how you can send more complex data across wires. Make a proposal and discuss with others.

Exercise 3.
Watch the two videos under the Resources section. How are they related to your activity? Discuss.

1.10 Problems

Problem 1.1 What is a bit?

Problem 1.2 How many bits are there in a kilobyte?

Problem 1.3 Explain the use of Ground (GND) and 3V pins in your micro:bit.

Problem 1.4 How many bits did you send to the receiver in your “Simple Heart Transfer” program?

Problem 1.5 How are the bits sent over the wire in your program?

1.11 Resources

- **Video: What is the Internet (Code.org) - https://youtu.be/Dxcc6ycZ73M**
- **Video: The Internet: Wires, Cables andWifi (Code.org) - https://youtu.be/ZhEf7e4kopM**
- **BBC Bitesize, Introducing Binary - http://www.bbc.co.uk/education/guides/zwsbwmn/revision**

2. WIRELESS AND BROADCAST COMMUNICATION

2. Introduction

Wireless (radio) communication, for example WiFi and mobile phones, is a popular way to connect to the Internet. In Chapter 1, you connected two micro:bits via wires. In this chapter, you will connect your micro:bits using radios.

Doing this, you will not only learn how to use your micro:bit's radio but also, broadcast communication. Wireless communication is typically broadcast: one micro:bit can send messages to all micro:bits. In summary, this chapter covers:

- ***wireless* communication and how to configure the micro:bit radio**
- **the concept of *broadcast* and *broadcast address***
- **receiving and sending different message types (for example, a number or a string) using broadcast**
- **when broadcast is useful, and when it isn't**

2.2 What you'll need

- **2 micro:bits**
- **2 battery holders, and 4 AAA batteries**
- **1 teammate**

2.3 Background

Wireless communication uses electromagnetic radiation - radio waves and microwaves - to send information. Radio waves are essentially electromagnetic waves radiating from an antenna (like the antennas of a WiFi router). So, wireless communication is always broadcast. In other words, the signals from the WiFi routers can be heard by other WiFi devices tuned into the same radio frequency.

Read more about frequency in the Further Reading section at the end.

Definition 1. — Broadcast.
In networking, broadcast communication means the message of a single sender is transmitted to all receivers in a network.

But, does this mean that broadcast is only possible with wireless communications? No, but it is more cumbersome. For instance, in wired communication, broadcast is possible by repeating the same message over all the wires.

Finally, receivers may refuse to receive broadcast messages if they are not labeled with a broadcast address.

Definition 2. — Broadcast address.
A broadcast address is a special address which says all devices in the network should receive this message.

In a micro:bit, the broadcast address can be configured by setting the group ID of micro:bit's radio. All the micro:bits need to have the same group ID for the broadcast to work. You will experiment with broadcasting with micro:bits in Section 2.3 .

Further Reading

Let's look at wireless communication in a bit more detail. You already learned that radio waves are essentially electromagnetic waves. Scientists have found that electromagnetic waves can be arranged together on a scale called electromagnetic spectrum. Figure 2.1 shows the electromagnetic spectrum, and the different electromagnetic waves [a].

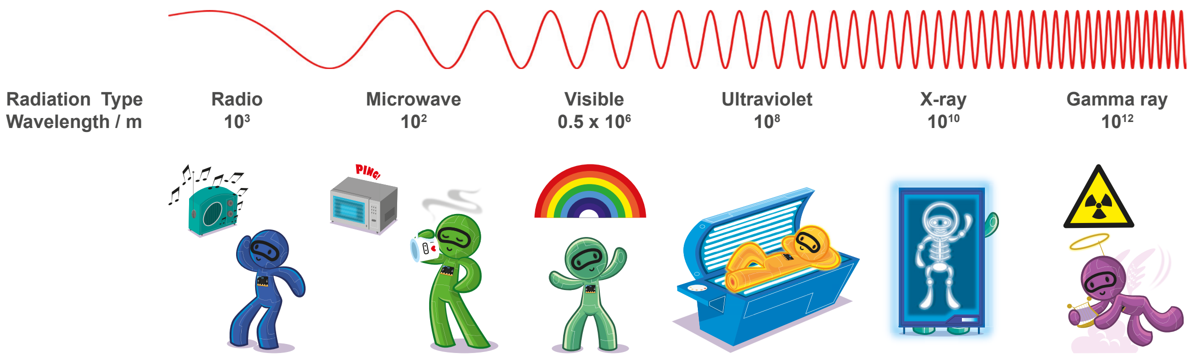


Figure 2.1: Electromagnetic spectrum.

One thing to notice in Figure 2.1 that radio waves are within the frequencies 30KHz and 300 GHz in the electromagnetic spectrum. Radio waves include microwaves, which have frequencies between 300MHz and 300GHz. Radio waves travel fast - they move at the speed of light, which is around 300,000 km per second! Let's define frequency more formally. The frequency of a wave is the number of waves passing a point in one second. The unit of frequency is hertz (Hz). Like the examples above, you will typically see that frequencies are given as megahertz (MHz) or gigahertz (GHz). 1 MHz is equal to 1 million (10^6) Hz. 1 GHz is equal to 1 billion (10^9) Hz. Your micro:bit's radio operates in the frequency range of 2402 MHz to 2480 MHz. What other wireless technologies operate in the same range as the micro:bit's radio?

Hint: The resources section at the end of this chapter will be useful too. In addition to frequency, another important parameter of electromagnetic waves is wavelength. The wavelength of a wave is the distance between a point on the wave and the same point on the next wave. The unit of wavelength is meters. Figure 2.2 shows an example of a wavelength.

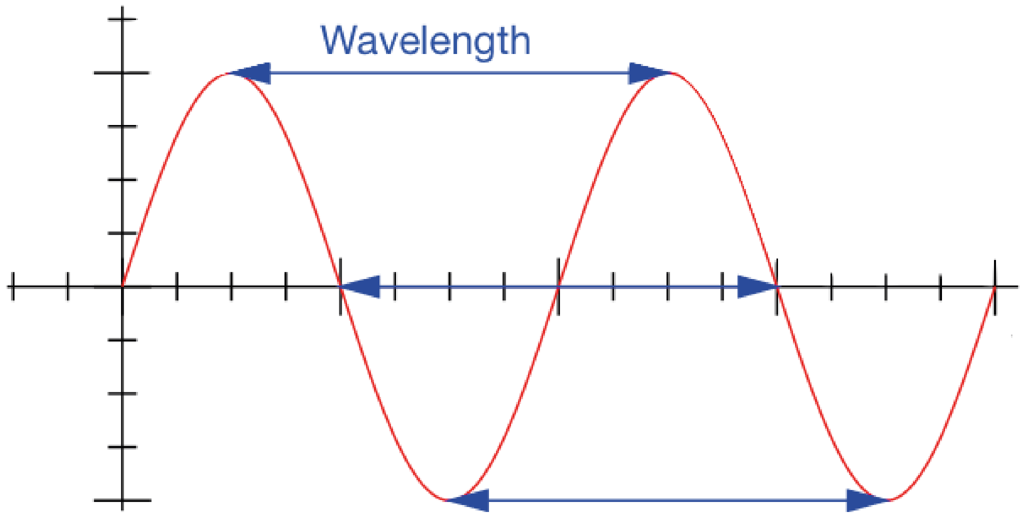


Figure 2.2: Wavelength is the distance between a point on the wave and the same point on the next wave. It is measured as the distance between two peaks.

Frequency and wavelength are related. The relationship between frequency and wavelength is given by a formula:

$$\text{wavelength (meter)} = \frac{\text{Speed of light (meter/second)}}{\text{Frequency (Hertz)}} \qquad (2.1)$$

From Equation 2.1, we see that the higher the frequency, the shorter the wavelength. You can see this also in Figure 2.1. How long do you think your micro:bit's radio waves are?

2.4 Programming: Receiving and sending broadcast messages

In this activity, you will learn how you can receive a message from a broadcasting micro:bit. Also, you will send broadcast messages yourself. If you are running this activity with your teacher in a classroom, your teacher's micro:bit will be the broadcast sender and you will try to receive from this micro:bit. If you are running this activity alone or with a friend, you can find the example codes for the broadcasting micro:bit under https://microbit.nominetresearch.uk/networking-book/. You can use these examples to test your receiver code by downloading it to a second micro:bit. These files will run on your micro:bits, but you will not be able to display the code using the JavaScript Blocks editor. You will complete the following three tasks to experiment with broadcasting.

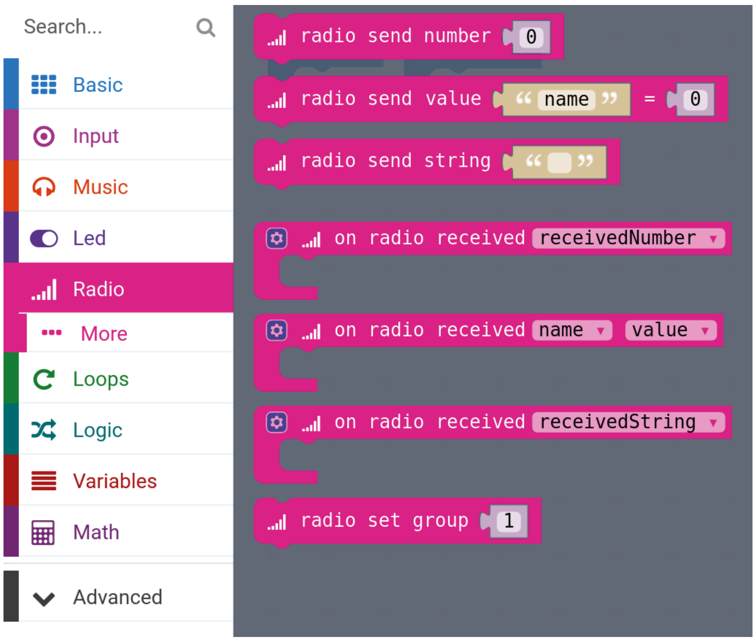
2.5 Task 1: Configure your radio

Description: For broadcast communication, you need all your micro:bits to have the same radio group ID. This group ID will be the broadcast address. This is like tuning into the correct channel to receive a TV broadcast.

Instruction: Program your receiver micro:bit's group ID to 0. This is the group ID used in the example broadcast sender programs [1]. For this, use the code block for setting the radio group in the JavaScript Blocks editor. It's under the Radio menu, as shown in Figure 2.3. You can learn about the radio blocks in more detail at https://makecode.microbit.org/reference/radio.

[1]If you are using your own programs to send a broadcast, you can select the group ID as you like.

[a]Image by Dicklyon (Richard F. Lyon) - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7184592



```
radio send number 0
radio send value "name" = 0
radio send string ""
on radio received receivedNumber
on radio received name value
on radio received receivedString
radio set group 1
```

Figure 2.3: Setting the Radio group in the JavaScript Blocks editor.

2.6 Task 2: Receive a broadcast message

Description: In this task, you will program your micro:bits to receive a message from a broadcasting micro:bit. You will use the example broadcast sender programs to test your receiver program.

When writing your receiver programs, there are two questions you need to think about.

1. Which blocks in the JavaScript Blocks editor do you need to use to receive a radio message?
2. Using these blocks, can you receive any type of message, for example, a number or a string?

Instruction: First, you will start by programming micro:bits to receive a number.

Download https://microbit.nominetresearch.uk/networking-book/SendNumber.hex into your sender micro:bit.

This sender program uses the radio group 0 to broadcast and sends a number between 0 and 9, whenever button A is pressed Program your micro:bit to receive and display a number. Test your program using the sender micro:bit.

Second, you will program your micro:bit to receive a string. Download https://microbit.nominetresearch.uk/networking-book/SendString.hex into your sender micro:bit. This program also uses radio group 0 and sends a string, whenever button A is pressed. Program your micro:bit to receive and display the string. Test your program using the sender micro:bit. What did you receive?

2.7 Task 3: Send a broadcast message

Description: Now it is your turn sending broadcast messages. If you run this exercise in a large group, with several micro:bits, you should notice that you are receiving a lot of messages! Can you guess who is sending which message?

Instruction: Program your micro:bit so that it can send a number when you press the button A and a string if you press button B. Extend your receiver program so that you can receive either a string or a number. For this, you will use a neat trick for the “on radio received” block. Pressing the little settings button on the block brings up a menu. This menu will allow you to drag additional values under Packet block. You will notice that the original “on radio received” block will extend to show these additional values. Figure 2.4 shows how the trick works.

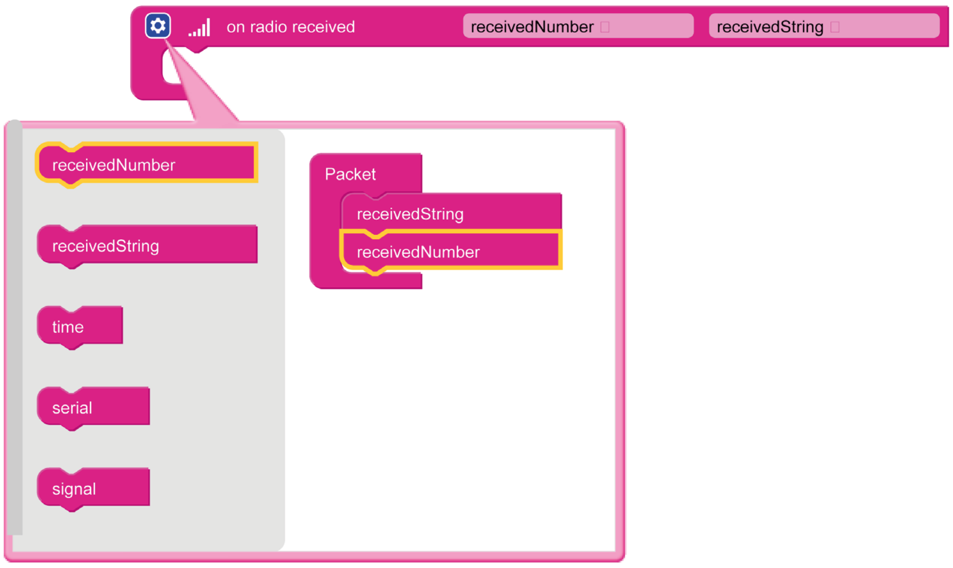


Figure 2.4: Making micro:bit radio to receive either a string or a number.

2.8 Extended activity

Exercise 1.
Extend your program in Task 2 for receiving a string. Display a "Sad" face on your micro:bit's display until you receive a "Hello" message. Then display a "Happy" face for 2 seconds.

Exercise 2.
Discuss some issues with broadcast communication. Is it always useful or necessary to send messages to everybody? What about privacy? Is this a problem that everybody receives all messages?

2.9 Problems

Problem 2.1 Which frequency range does your micro:bit's radio work in?

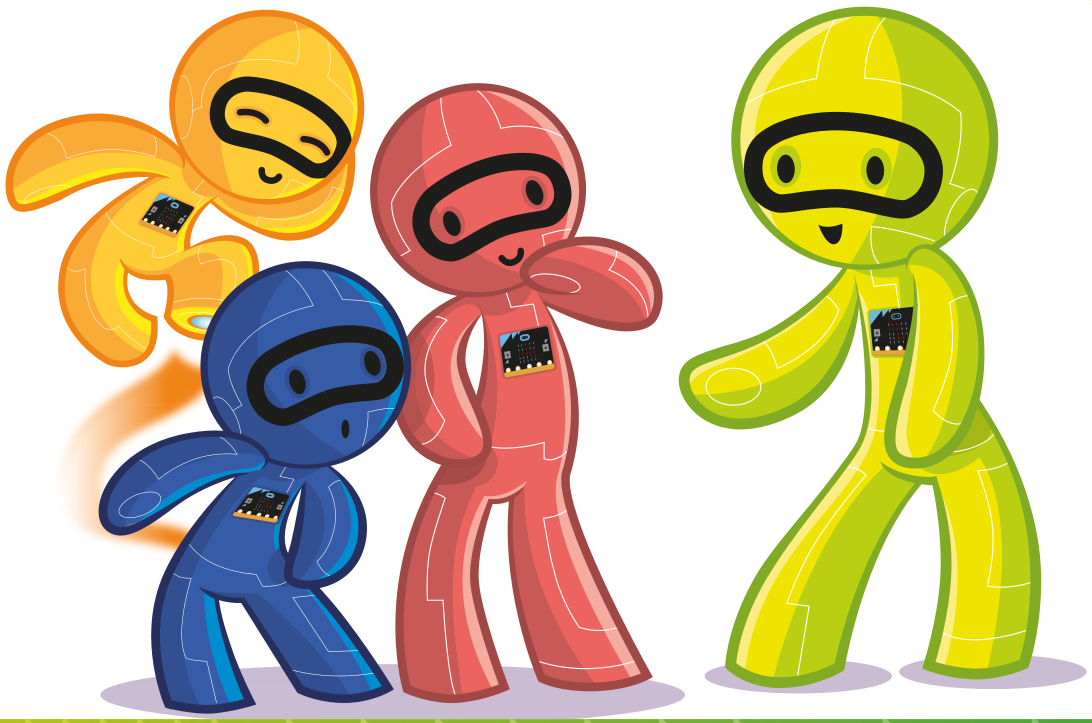
Problem 2.2 What is the speed of light?

Problem 2.3 Using the Equation 2.1, calculate the wavelength of your micro:bit's radio.

Problem 2.4 Is it easier to broadcast using wired or wireless communication? Why?

2.10 Resources

- **BBC Bitesize, The electromagnetic spectrum - http://www.bbc.co.uk/schools/gcsebitesize/science/edexcel/electromagnetic_spectrum/electromagneticspectrumrev1.shtml**
- **BBC Bitesize, An introduction to waves - http://www.bbc.co.uk/schools/gcsebitesize/science/aqa_pre_2011/radiation/anintroductiontowavesrev2.shtml**
- **Video: How does Wi-Fi Work? (Brit Lab) - https://youtu.be/xmabFJUKMdg**
- **Wired, Why Everything Wireless is 2.4GHz?- https://www.wired.com/2010/09/wireless-explainer/**



3. GROUP COMMUNICATION

3. Introduction

In the previous chapter, you experimented with broadcast: sending messages to everybody. In this chapter, you will learn about sending a message so that it just goes to a smaller group of people. This is an activity that is best carried out with a large group of friends or class mates so that you can experiment with different groups and group sizes.

Group communication (also known as multicast) is an interesting concept, and enables several of today's Internet technologies. For example, it enables sending videos as fast as possible over the Internet. In this chapter, you will learn:

- **The concept of *group communication* and *group* or *multicast address***
- **When group communication is useful and when it isn't**

3.2 What you'll need

- **2 micro:bits**
- **1 whiteboard/board**
- **board markers/post-it notes**
- **1 teammate**

3.3 Background

In the previous chapter, all micro:bits received messages from all the other micro:bits. This might have got confusing (or amusing!). Now, let's try limiting who you can send messages to and receive messages from. This is called group communication. Group communication is used in the Internet to send to many people at the same time. For example, Internet television and video conferencing use group communication.

Definition 1. — Group communication.
In group communication or multicast, a message is sent only to the computers in the group.

For this, the messages need to be labeled with a group or multicast address.

Definition 2. — Group address.
A group or multicast address is a special address which says all devices in the group should receive this message.

To configure the group address (or group ID) in your micro:bit's radio, you will again use the "radio set group" block under the Radio menu like in Chapter 2. The main challenge of this chapter is creating the groups for communication. How do computers learn about and join these groups?

What happens when they leave a group? In this chapter, you will have a chance to think about these questions when you experiment with creating groups.

Further Reading

When configuring group IDs for micro:bits, you will notice that the group IDs range from 0 to 255. This is the decimal (base 10) representation of group IDs. But we can also write these group IDs in binary (base 2). For the binary case, we will need 8 bits to get to a maximum group ID of 255.

Let's think about the binary representation of group IDs. Figure 3.1 shows an example for the group ID 172 in 8 bits: 10101100. Notice that, we start reading bits from right to left. Each bit is numbered 1 to 8, corresponding to a power of two. The rightmost bit, bit 1, means $2^0 = 1$. Bit 2 means $2^1 = 2$ and we continue like this until we reach bit 8, which means $2^7 = 128$. Each bit location may contain either 0 or 1. To find the decimal value of 10101100, we need to do some maths: For bit location x, we multiply its bit value by 2^{x-1}. For the first bit location 8, the bit value is 1, and we need to multiply 2^8-1 = 2^7 = 128) by 1. After doing this for all bit locations, we add all the values we found. The result of this addition is 172. Now, check for the case 11111111. Is it really equal to 255? To learn more, see the BBC Bitesize, Binary revision page in the Resources section.

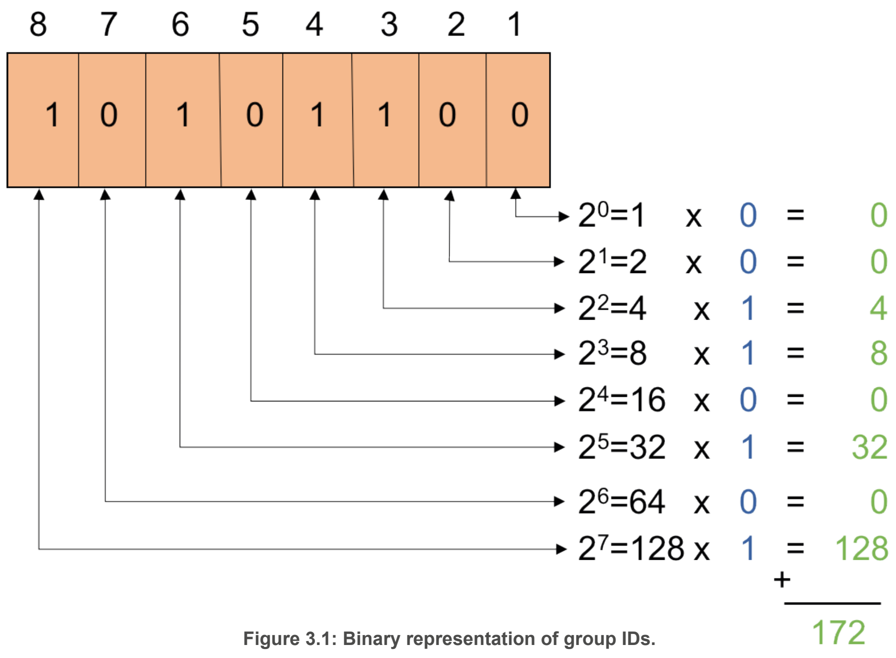


Figure 3.1: Binary representation of group IDs.

3.4 Programming: Creating groups and messaging within groups

In this chapter, you need to work together in pairs or small groups, with at least 2 micro:bits in each group. You will complete two tasks to program your micro:bits to send messages to and receive messages from your group.

3.5 Task 1: Create groups

Description: In this task, you will choose a unique group ID for your group, and configure your radios with this group ID. You will use the radio block *radio set group* in your program in the JavaScript Blocks editor. When choosing group IDs, you have to think about the best way to choose this number. **Hint: What would happen if two groups choose the same number, and how would you make sure that doesn't happen?**

Instruction: Use the board and post-it notes to choose a group ID. Make sure your group ID is not the same as any other group ID.

3.6 Task 2: Send and receive messages

Description: You will use the programs from the previous chapter to send and receive messages to your group. You will change these programs to count the number of messages you receive. This way, you will test whether you receive messages that only come from your group. **Instruction:** Write a sender program that sends a random number between 0 and 9, when you press the button A. Write a receiver program that increments a counter each time it receives a number. When you press the button A at the receiver, it displays the value of the counter. With your group, test that you are receiving the correct number of messages. Test together with other groups that you are not receiving their messages.

3.7 Extended activity

Exercise 1.
How easy or difficult would it be if micro:bits could create groups automatically themselves? How would they pick a group ID? How would they make sure nobody else had that number? Would broadcast be useful? Discuss with your teammates.

Exercise 2.
Can a micro:bit be part of two groups or more? How would you program your micro:bit to do that?

3.8 Problems

Problem 3.1 Fill in the blank in this sentence: "A one-to-many communication between one sender and a group of receivers is --- communication."

a) unicast

b) multicast

c) broadcast

d) none of the above

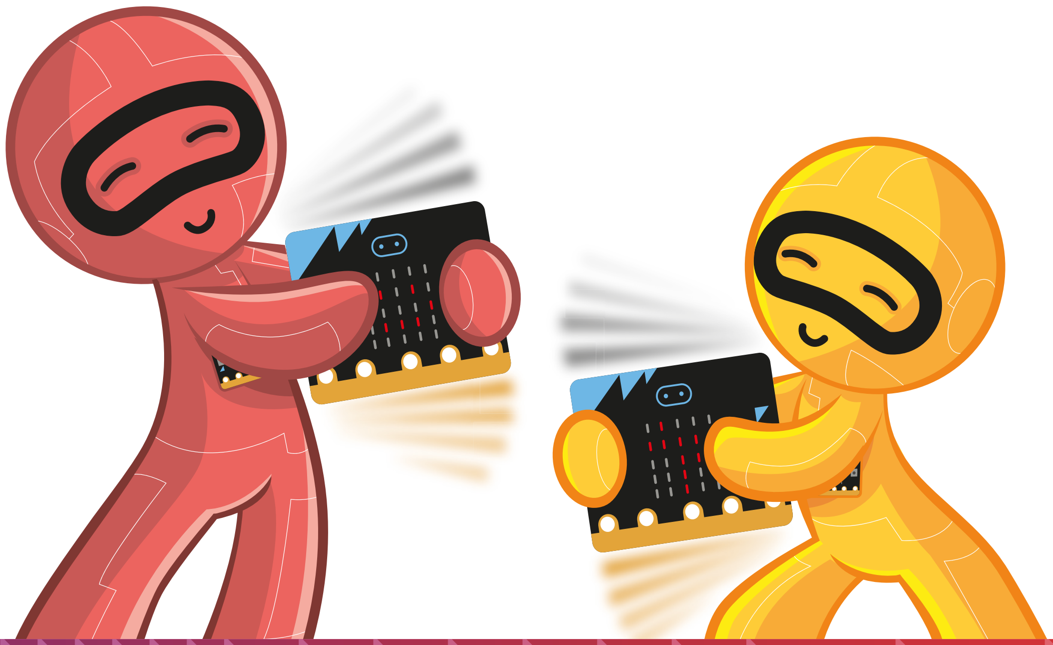
Problem 3.2 Let's assume the group ID is 3 bits. For example, 010 is a group ID. What is the maximum number of groups can you have in a network?

Problem 3.3 If the group ID were 6 bits, what is the largest group number you could choose for your micro:bit?

Problem 3.4 "Compared to broadcast, the receivers in group communication receive more messages. "True or False?

3.9 Resources

BBC Bitesize, Binary revision - http://www.bbc.co.uk/education/guides/z26rcdm/revision



4. GAME 1: SHAKEY DONKEY

4. Introduction

Let's put everything you have learned so far into practice with a fun game. If you have not already seen it, Shakey Donkey is a micro:bit game that uses the radio [1]. Shakey Donkey is played with two players, and it measures how fast you react to a Donkey appearing in your micro:bit. The game starts with shaking micro:bits. The moment your micro:bit displays a Donkey, you should shout "Donkey!" and shake your micro:bit to make it disappear. At the end, when you press the button A, if your micro:bit displays a happy face, you won!

In this chapter, you will practice:

1. **the concept of *group communication***
2. **using *group* or *multicast address***
3. **sending and receiving messages**
4. **shake and button inputs**
5. **program variables and random numbers**

4.2 What you'll need

- **2 micro:bits**
- **1 whiteboard/board**
- **board markers/post-it notes**
- **1 teammate**

4.3 Programming: Playing Shakey Donkey

Description: To be able to play this game in groups of 2, you will set a unique group ID for your pair. Then you will program the Shakey Donkey game given to you in three parts in the Figure 4.1.

[1]This game is by David Whale. We thank him for allowing us to use it in this book.

Instruction: To set your groups, repeat the activity from Chapter 3. Make sure your group IDs are unique!

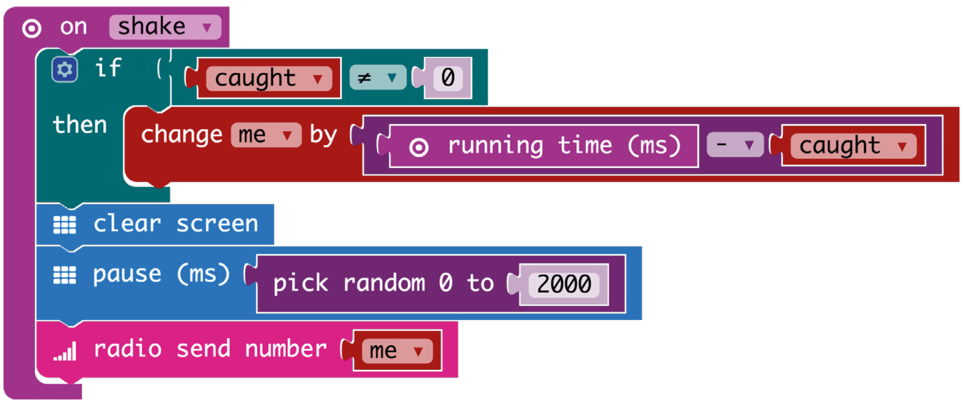
The game is played by shaking your micro:bit each time the donkey appears on your display, to get rid of it.

So, the first thing to do is to program what your micro:bit should do "On shake". This is shown in Figure 4.1a. Notice that, in this first part, your program sends a number.

So, you need a piece of code for handling a received number. This second part is shown in Figure 4.1b. Add it to your JavaScript Blocks editor program.

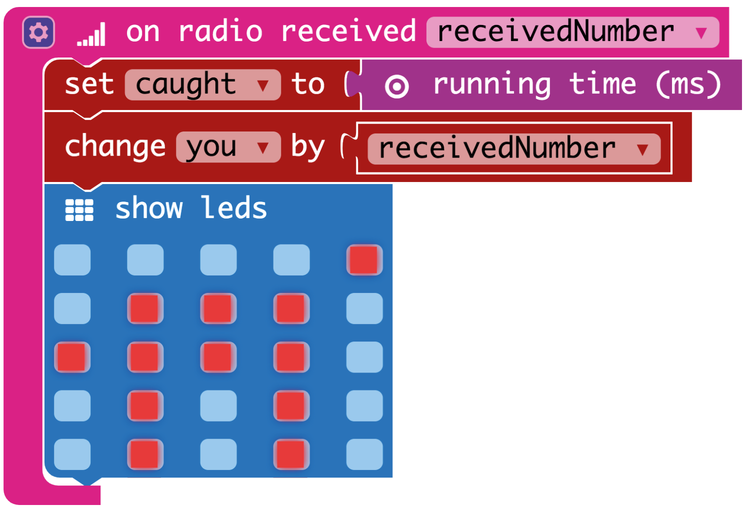
The third part, shown in Figure 4.1c, handles the case when the button A is pressed. This part of the program decides whether you won or not. Add this part into your program too.

Download the program into your micro:bits. Play the Shakey Donkey game with your teammate. Then, go through the problems to explain how your program works.



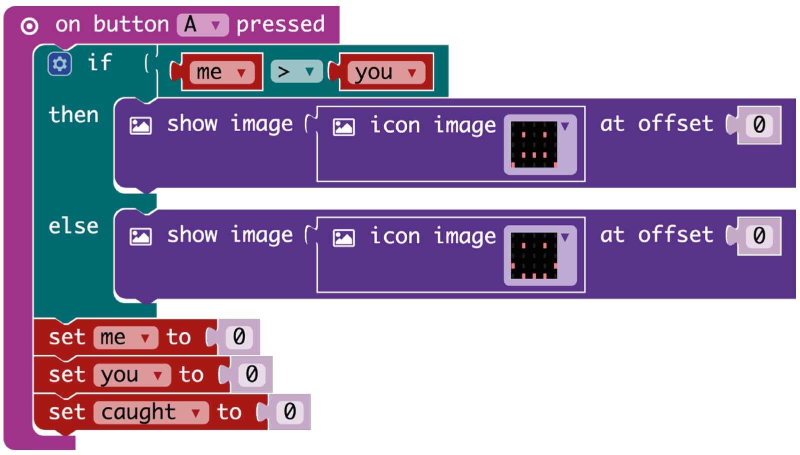
```
on shake
    if caught ≠ 0
    then
        change me by running time (ms) - caught
    clear screen
    pause (ms) pick random 0 to 2000
    radio send number me
```

(a) Shakey Donkey program - Part 1: Shake your micro:bit to send your reaction time.



```
on radio received receivedNumber
    set caught to running time (ms)
    change you by receivedNumber
    show leds
```

(b) Shakey Donkey program - Part 2: Receive the other player's reaction time, and display the donkey.



```
on button A pressed
  if me > you
  then
    show image icon image at offset 0
  else
    show image icon image at offset 0
  set me to 0
  set you to 0
  set caught to 0
```

(c) Shakey Donkey program - Part 3: Press button A to learn the result.

Figure 4.1: Parts of the Shakey Donkey game

4.4 Problems

Let’s first look at Part 1, in Figure 4.1a.

Problem 4.1 At the beginning, what is the value of the *caught* variable for both players? Does anybody need to change the *me* variable?

Problem 4.2 Who gets to send their *me* variable first?

Next, let’s look at Part 2, in Figure 4.1b.

Problem 4.3 When you receive a number, you set the *caught* variable? What does the *caught* variable mean?

Problem 4.4 You also change the you variable by the *received Number.* What does the you variable track?

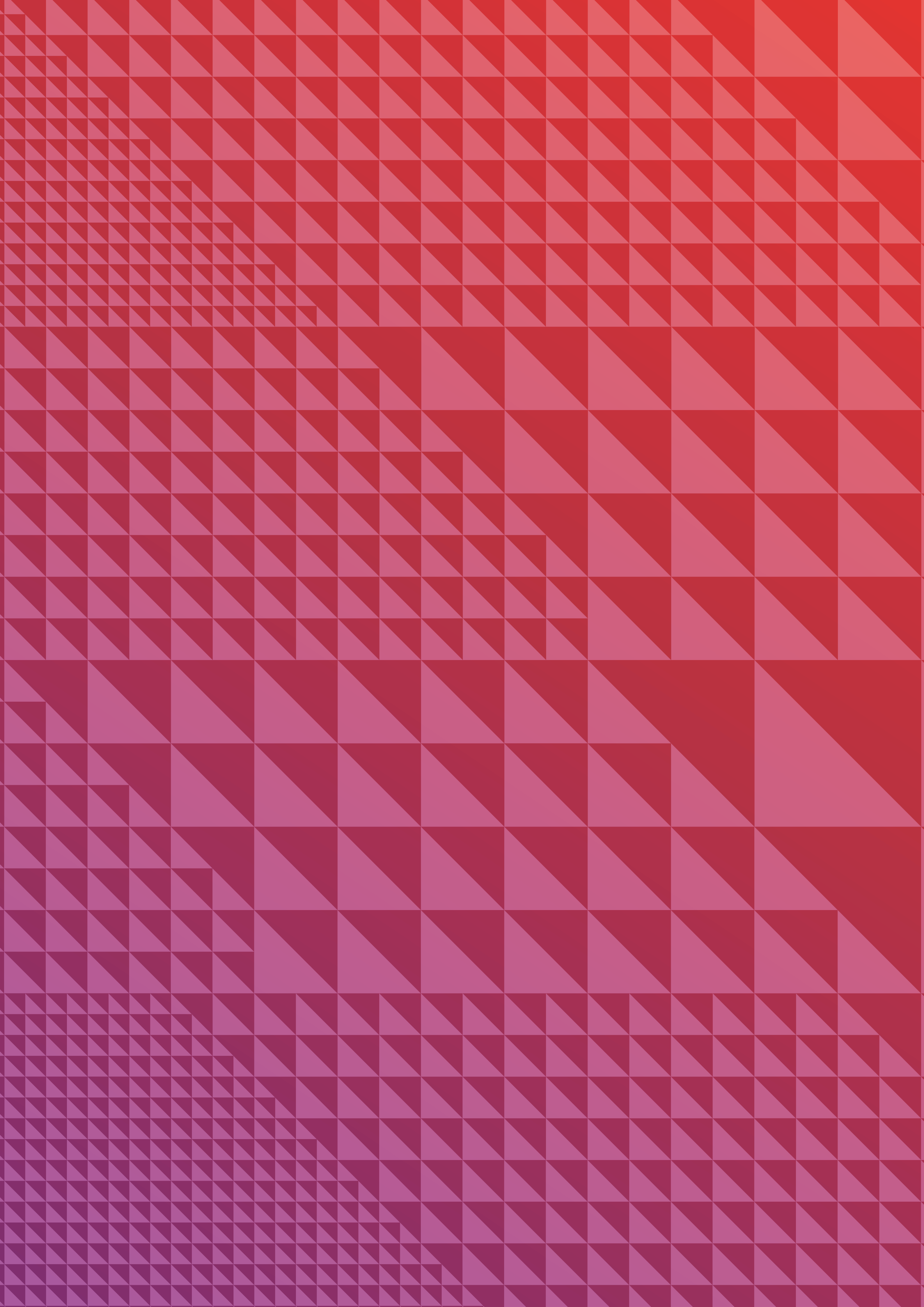
Now, let’s look at both Parts 1 and 2.

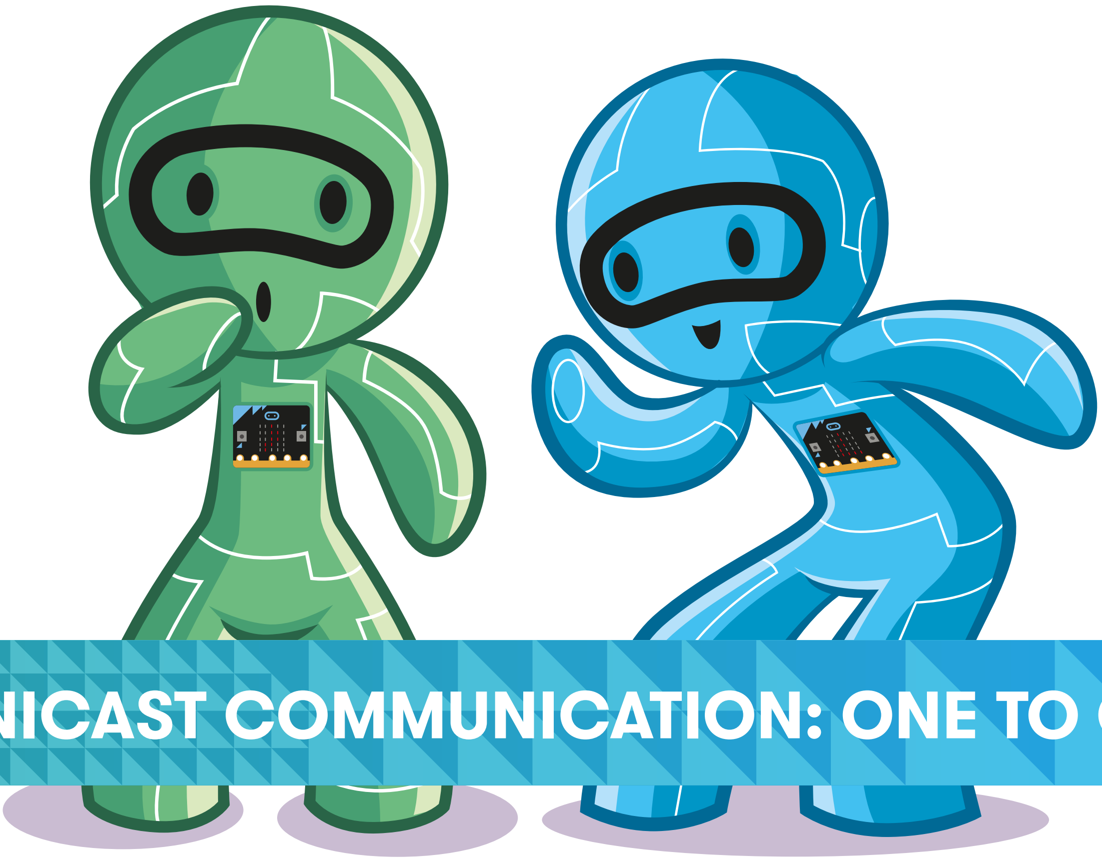
Problem 4.5 Imagine you already started playing the program. You saw some donkeys appear on your display, and you shook them away. How did your *me* variable change? What is it equal to?

Finally, let’s look at Part 3, in Figure 4.1c.

Problem 4.6 How do you know you won? Does the other player know the result? How? Explain how the *me* and *you* variables are used to decide the winner.

Problem 4.7 How would you make sure you win this game?





5. UNICAST COMMUNICATION: ONE TO ONE

5. Introduction

Unicast, sending messages to a single receiver, is the typical way we communicate on the Internet. For example, to view a web page, we send unicast messages to a server, which in turn sends us the page to display on our browser. In this chapter, you will send unicast messages, for example to a friend's or teammate's micro:bit. Doing this, you will learn some basic ideas of computer networking, including:

- **the concept of *unicast***
- **the concept of a *protocol***
- **the concept of an *address* and *IP address***
- **the concept of a *data packet* and a *header***

5.2 What you'll need

- **2 micro:bits**
- **1 whiteboard/board**
- **board markers/post-it notes**
- **1 teammate**

5.3 Background

This chapter covers unicast communication. So, what is unicast?

Definition 1 — Unicast.
Transmission of a message to a single receiver.

When transmitting messages to each other, computers use *protocols*.

Definition 2 — Protocol.
A set of rules for how messages are sent across networks.

Simply, protocols define how computers should send messages and what they should do when they receive a message. On the Internet, every computer or device follows the Internet Protocol (IP). According to Internet Protocol, each device is given a unique *address*, called an *IP address*. Remember you have already used special addresses for broadcast and multicast. In this chapter, we consider unicast addresses. IP address is used for unicast on the Internet[1].

Definition 3 — IP address.
A unique string that identifies computers that use the Internet Protocol to communicate over a network. This string is made up of 4 decimal numbers, that range between 0 and 255. Each decimal is separated by dots. For example, 213.248.234.11 is an IP address.

Your micro:bit also has an address (but it is a bit different). You already partly changed your micro:bit's address, by changing the group ID.

When two computers communicate, the sender sends a data packet to the receiver.

Definition 4 — Data packet.
A data packet is a piece of data sent over a network. This piece of data has an actual message part (for example, an image or a text) and one or more header parts. A header contains helpful information for protocols like the sender and receiver IP addresses.

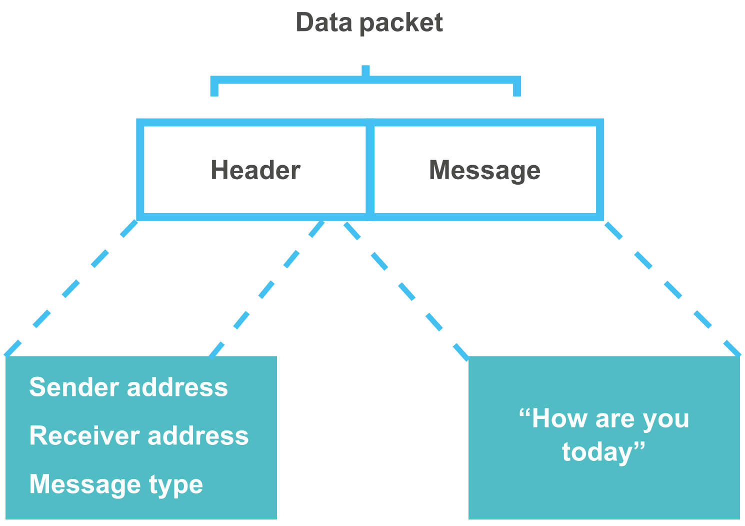


Figure 5.1: A data packet contains a message and a header. A header contains information to help a protocol such as sender and receiver addresses, and message types. Different protocols may add different headers to a message.

Figure 5.1 shows how the data and one header forms a data packet. In this figure, as well as the sender and receiver addresses, the example header also includes a message type. Message type tells the receiver whether it is receiving, for example, a text or an image. Remember, in the previous chapters, you programmed your receivers to receive a specific type of message. If your packets contained a header with the message type, then it would be easier to write the receiver program. In this chapter, to unicast to other micro:bits, you will create a data packet by adding a header with source and destination addresses.

[1]There are also special range of IP addresses that can be used for multicast and broadcast

5.4 Programming: Sending and receiving unicast messages

In this section, you will program your micro:bits to send and receive unicast messages completing four tasks. To start with, you need two micro:bits. For unicast to work, your radio should receive all messages sent, but your program should read only the ones that are addressed to you. This is like seeing all the post coming into your house, but only opening the envelopes with your name on.

5.5 Task 1: Configure your radio

Description: To receive any packet, sent by anybody, you need to use broadcast as the underlying communication.

Instruction: Set your radio group ID like you did in Chapter 2 for broadcast communication.

5.6 Task 2: Design your header

Description: The sender micro:bit needs to add a header to each message before sending. The message header will include:

- **sender address**
- **receiver address**

For the message header, you will create a special string.

Instruction: First construct the sender and receiver addresses. With your teammate, pick two-letter strings as micro:bit addresses. You need one address for your micro:bit, and one address for your teammate's micro:bit. For example, you can use your initials: These are "CS" and "AK" for the authors of this book. **Important! Your addresses should be unique across all the addresses of micro:bits that are in the same room with you.** Next, join the strings for sender and receiver addresses to create a header. Use the blocks under the *Text* menu in the JavaScript Blocks editor (see Figure 5.2), for example, *join*.

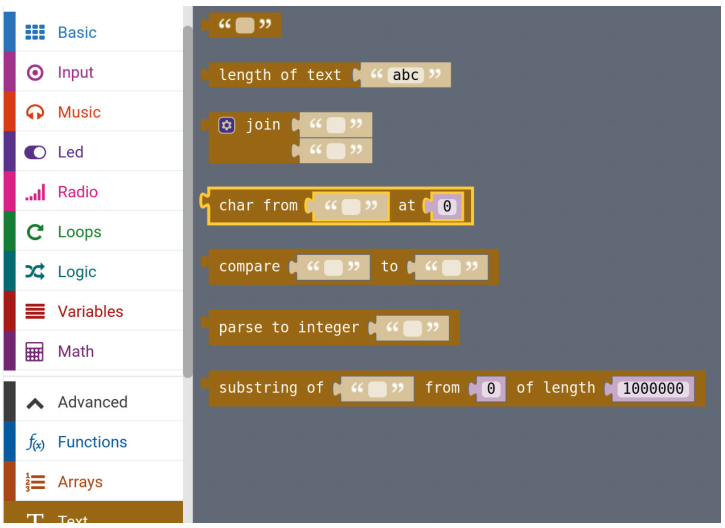


Figure 5.2: Text menu in the JavaScript Blocks editor

5.7 Task 3: Create your packet and send

Description: Now it is time to create your packet. As shown in the Figure 5.1, a header and a message form a packet. Your final packet will have the following information:

- **sender address**
- **receiver address**
- **your message**

Instruction: Pick a string as your message. For example: "Hello". Use the Text blocks to join your message string with your header. Now, your sender micro:bit is ready to send unicast packets.

5.8 Task 4: Receive a packet

Description: When the receiver micro:bit receives a packet, it decides whether to receive or ignore the packet. Notice that the receiver micro:bit receives a single string, but it knows that this string is made up of:

- **Sender address: first 2 letters**
- **Receiver address: next 2 letters**
- **Sender's message: the rest of the string**

The receiver needs to use this information to decide which packets are for itself.

Instruction: Divide the received string into the *sender address*, *receiver address*, and *sender's message* variables. Use the Text blocks, for example *substring* & *compare*. Check if the receiver address is equal to your micro:bit's address. If it is, then your micro:bit is the rightful receiver. Display the sender address and the message on your display. If your micro:bit is not the receiver, be a good citizen and ignore the message.

5.9 Challenge: Filter senders

Description: Sometimes, you may not want to receive messages just from anybody. For this, you will write a program so that you only receive messages from two people you know. We will call this your *allow-list* (often referred to as a *whitelist*).

Instruction: Extend the receiver program to also check the *sender address* field in the header. Check whether this address is in your allow-list. If yes, display the *sender address* and the message. If not, ignore the message. Test your program with addresses in and outside your allow-list.

5.10 Extended activity

Exercise 1.
You may have written two separate programs: one for the receiver and one for the sender. Change your program so that both micro:bits can send and receive.

Exercise 2.
Did you try listening out for messages sent from other micro:bits in your class? How could your program achieve this? Is this the right thing to do? How might you protect your messages from others snooping?

Exercise 3.
In this chapter, we have covered one way to do a unicast: Putting sender and receiver addresses in a data packet header. But there is another way. Remember Chapter 3. If you set your group to be only for your pair of micro:bits, then this is like you are unicasting. To unicast like this, choose a unique group ID, like you did in Chapter 3. Announce it on the board so that no one else uses it. Write programs for your pair of micro:bits that send and receive using this radio group ID. What are the limitations of doing unicast like this? **Hint: Think about how many possible group IDs there are. Would this be enough for everyone in the world who has a micro:bit?**

5.11 Problems

Problem 5.1 In what ways is unicast like broadcast and group communication? In what ways is it different?

Problem 5.2 Which ones are not IP addresses?

a) -1.0.0.1

b) 278.0.10.0

c) 104.20.14.61

d) 127.0.0.1

e) 161.23.84;18

f) 161.73.246.13

g) 104.20.14.61.15

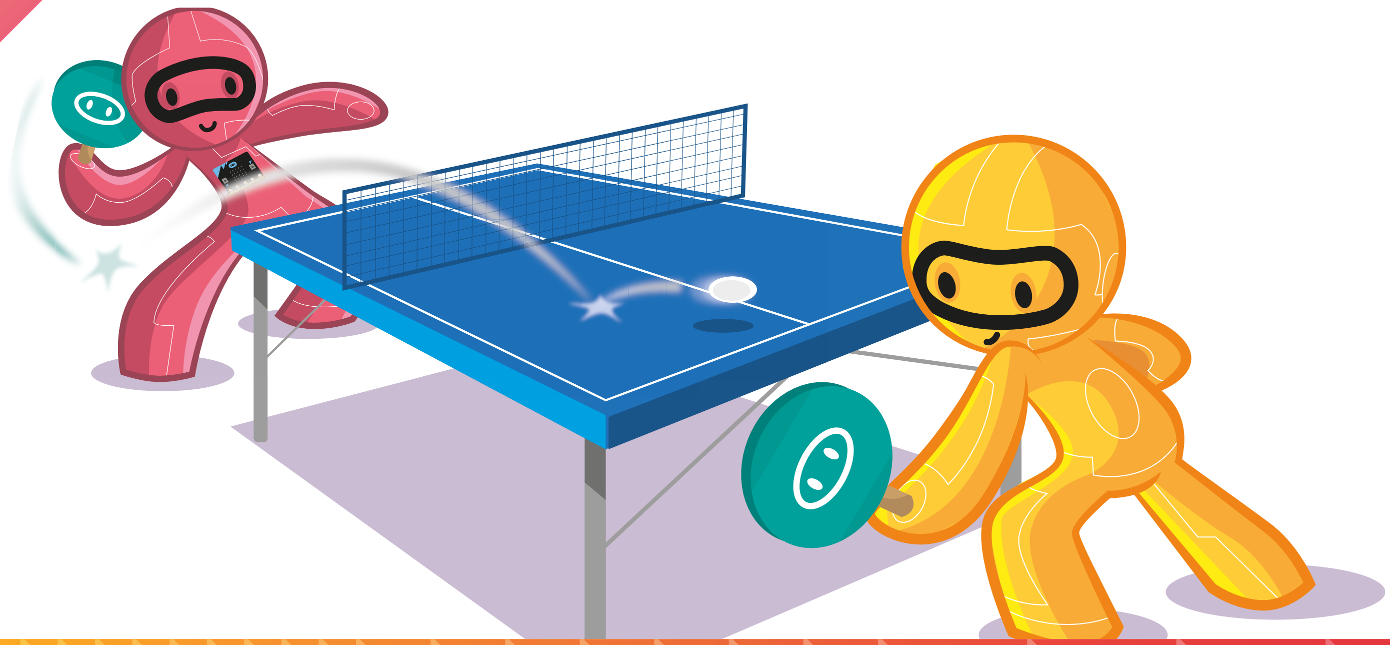
Problem 5.3 In this chapter, you used two-letter strings for your addresses. How many different people can you unicast using this address size?

Problem 5.4 When selecting an address size for your message header, can you pick any size you like? In your program, what happens if you increase your address size to 10 letters? Do you see any benefits? Or are there any limitations?

Problem 5.5 How does the size of a data packet header affect the actual packet size? If your data packet size were 100 Bytes, and your header size were 10 Bytes, how big could your messages be? What happens if the header size increases to 50 Bytes?

5.12 Resources

- **Video: IP addresses and DNS (code.org) - https://youtu.be/5o8CwafCxnU**
- **Video: IP addresses (CommonCraft) - https://www.commoncraft.com/video/ip-addresses**
- **BBC bitesize networks - http://www.bbc.co.uk/education/topics/zjxtyrd**



6. TWO-WAY UNICAST

6. Introduction

In this chapter, you will learn about *bidirectional communication*: sending a message to another micro:bit and getting a response to your message. You will also learn about the Ping program, which is a commonly used tool to check if computers are still connected to the Internet. This chapter will build on the learnings from Chapter 5. The new ideas are:

- **The idea of 2-way communication (*bidirectional communication*)**
- **The Ping program**
- **The concept of *round-trip-time***

6.2 What you'll need

- **2 micro:bits**
- **1 whiteboard/board**
- **board markers/postit notes**
- **1 teammate**

6.3 Background

Bidirectional communication enables two-way communication between two computers.

Definition 1 — Bidirectional communication.
This is a communications mode in which data is transmitted in both directions but not necessarily at the same time.

In the previous chapter, your micro:bits had clear roles: there was a sender and a receiver. In bidirectional communication, either of the micro:bits can send and receive messages. This way, it becomes possible to create two-way protocols. In these protocols, when a computer sends a message, it waits for a certain response to its message.

Definition 2 — Ping.
Ping is an example of a two-way protocol. It is widely used in the Internet to test whether a networked computer is on and connected OK.

A ping program sends a *Ping* message to test whether computers are OK. It expects this message to be echoed back, for example with a *Pong* message. This is like playing ping pong but with computers and over networks. If the sender does not receive a response to its *Ping*, this shows there is a problem with the receiver. It is also a problem if it takes a long time before the sender receives a *Pong* response. So, a ping program measures the *round-trip-time* between the two computers to point out these problems.

Definition 3 — Round-trip-time (RTT).
Ping is an example of a two-way protocol. It is widely used in the Internet to test whether a networked computer is on and connected OK.

In other words, the sender measures the difference in time when it sent the Ping and when it received the *Pong*.

RTT = Time_receive -Time_send (6.1)

Figure 6.1 shows the relationship between, Ping, Pong, and round-trip-time

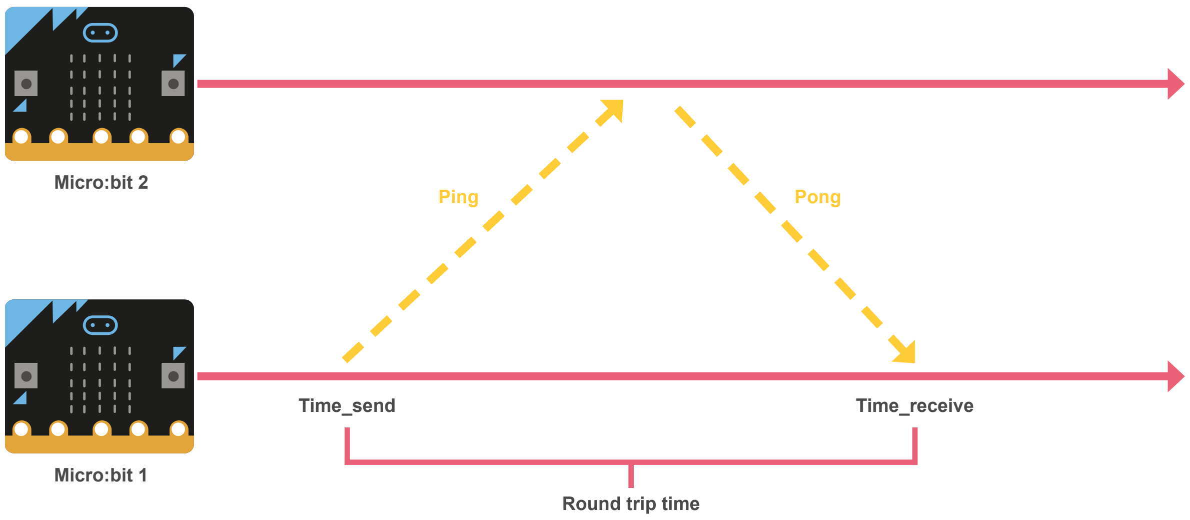
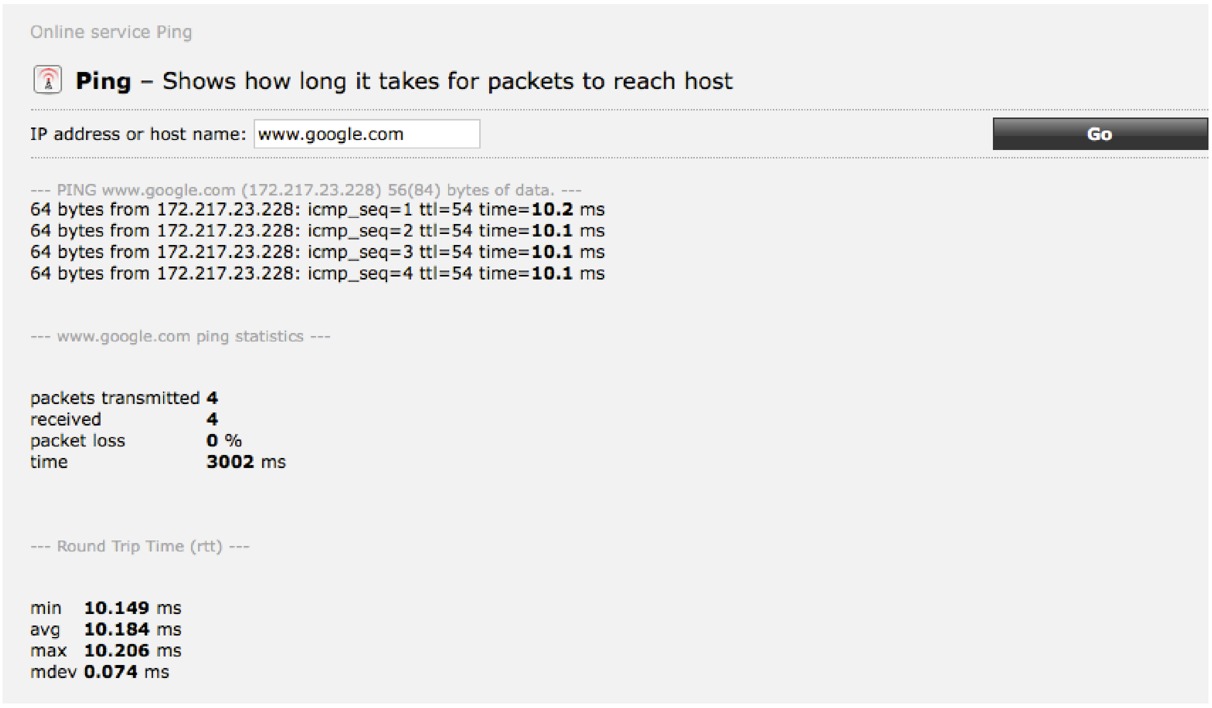


Figure 6.1: Round-trip-time. Micro:bit 1 sends a Ping message to Micro:bit 2 at Time_send. The Micro:bit 2 responds with a Pong message. Micro:bit 1 receives the Pong message at Time_receive. The difference between these two times, Time_receive and Time_send is the round-trip-time.

Besides round-trip-time (RTT), the Ping program reports statistical information. Figure 6.2 shows an example output as a result of using the command:

ping www.google.com

on the http://ping.eu/ping website. In the example in Figure 6.2, four Ping messages were sent to www.google.com. The round-trip-time for each message is given with the time value in each line. For example, for the first ping, the RTT is 10.2 ms (milliseconds). The program also reports ping statistics. For example, 4 packets were sent, 4 packets were received. This means 0% packet loss. The average RTT (shown as avg) is 10.184 ms.



Online service Ping

Ping – Shows how long it takes for packets to reach host

IP address or host name: www.google.com Go

```
--- PING www.google.com (172.217.23.228) 56(84) bytes of data. ---
64 bytes from 172.217.23.228: icmp_seq=1 ttl=54 time=10.2 ms
64 bytes from 172.217.23.228: icmp_seq=2 ttl=54 time=10.1 ms
64 bytes from 172.217.23.228: icmp_seq=3 ttl=54 time=10.1 ms
64 bytes from 172.217.23.228: icmp_seq=4 ttl=54 time=10.1 ms


--- www.google.com ping statistics ---


packets transmitted 4
received            4
packet loss         0 %
time                3002 ms


--- Round Trip Time (rtt) ---


min   10.149 ms
avg   10.184 ms
max   10.206 ms
mdev 0.074 ms
```

Figure 6.2: The output of running ping to send four messages to www.google.com. The http://ping.eu/ ping online program reports round-trip time and a statistical summary of the results.

With a micro:bit, to calculate the round-trip-time of your messages, you will use the *running time* variable.

Definition 3 — micro:bit running time.
A variable that keeps record of how long has passed since the micro:bit was turned on or reset (measured in milliseconds).

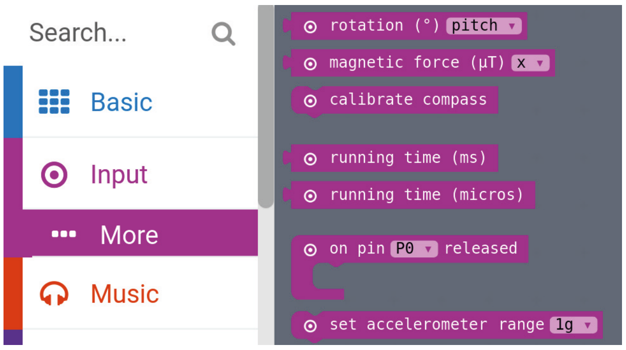


Figure 6.3: Running time

In the rest of this chapter, you will use the running time variable to calculate the round-trip time. It will be very useful to record the time when you first sent a message, and also when you received a response.

Hint: Recording running time means setting a variable equal to the current running time. You will need to combine the set item block in the JavaScript Blocks editor *Variables* menu with the running time block in the *Input* menu.

6.4 Programming: Ping

This activity is best done with a team of two. You will together program your micro:bits to run the Ping program. For this, you will need to complete four tasks.

6.5 Task 1: Prepare for unicast

Description: Ping uses unicast between the sender and the receiver micro:bits. Look at your notes for Chapter 5 and your unicast program to remember how to do unicast.

Instruction: Start with using your unicast program from the Chapter 5 as a basis. In this program, decide which micro:bit is going send the *Pings*, and which micro:bit is going to respond with *Pongs* Set the address variables based on your decision. Design your message header, *Ping* packet, and *Pong* packet.

6.6 Task 2: Send a Ping

Description: The ping sender records the time before it sends out a *Ping* packet. It unicasts the *Ping* packet.

Instruction: Use running time to record the *Ping* sending time. Send a *Ping* packet to the receiver micro:bit.

6.7 Task 3: Receive a Ping

Description: The receiver micro:bit responds a *Ping* message with a *Pong*.

Instruction: Program the receiver micro:bit to unicast a *Pong* packet when a *Ping* packet is received.

6.8 Task 4: Receive a Pong and calculate round-trip-time

Description: When the sender micro:bit receives the *Pong*, it calculates the round-trip-time.

Instruction: Program the sender to receive a *Pong* packet. When the *Pong* is received, record the time using the running time variable. Show the difference between receiving and sending times on your display. Run your program 5 times, and write down the send times that you see in your display.

Answer these two questions:

1. What is the minimum and maximum round-trip-time (RTT)?
2. What is the average RTT?

6.9 Exercises

Exercise 1.
Extend your ping program to send automatically more than one *Ping* message. Test it with 10 *Pings*. Calculate the average round-trip time of these *Ping* messages.

Exercise 2.
The ping program reports the round-trip-time. What if you wanted to calculate the time the message took one-way? Is it possible to calculate one-way times? In other words, is it possible to calculate how long it takes to send a *Ping* to the receiver? Or how long a *Pong* takes from the receiver to the sender?

6.10 Problems

Problem 6.1 In Figure 6.2, what is 216.58.213.100?

Problem 6.2 What is round-trip-time, and how is it calculated?

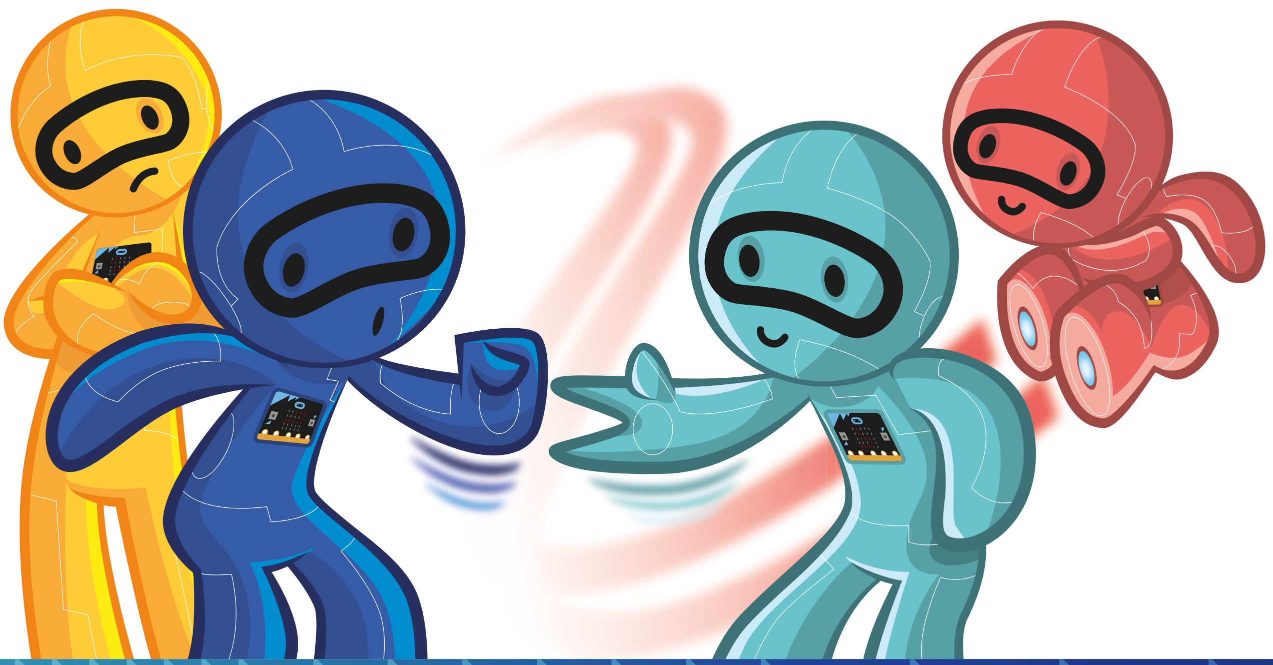
Problem 6.3 Think about the following scenario: micro:bit 1 sends a *Ping* to micro:bit 2 at time 5. If the round-trip-time is 10, at what time did the micro:bit 1 receive the *Pong* message?

Problem 6.4 In Figure 6.2, what are the minimum and maximum round-trip-times (RTTs)?

Problem 6.5 Figure 6.2 shows 0% loss. What would the loss percentage be if 2 Ping messages were lost out of 5?

6.11 Resources

- **Video: What is a Ping? - https://youtu.be/N8uT7LNVJv4**



7. GAME 2: ROCK, PAPER, SCISSORS OVER THE RADIO

7. Introduction

Let's play a game of rock, paper, scissors! This game is played with two players. Each player, at the same time, forms one of the three shapes (rock, paper or scissors) with their hands. Then, they use these rules to decide who wins:

- **The rock blunts the scissors.**
- **The scissors cut the paper.**
- **The paper covers the rock.**
- **If both players choose the same shape, it is a tie.**

Figure 7.1 shows these rules.

In this chapter, you will program this game using your micro:bits. Doing so, you will practice:

- ***Unicast communication***
- **Programming with variables**
- **Programming with conditionals**

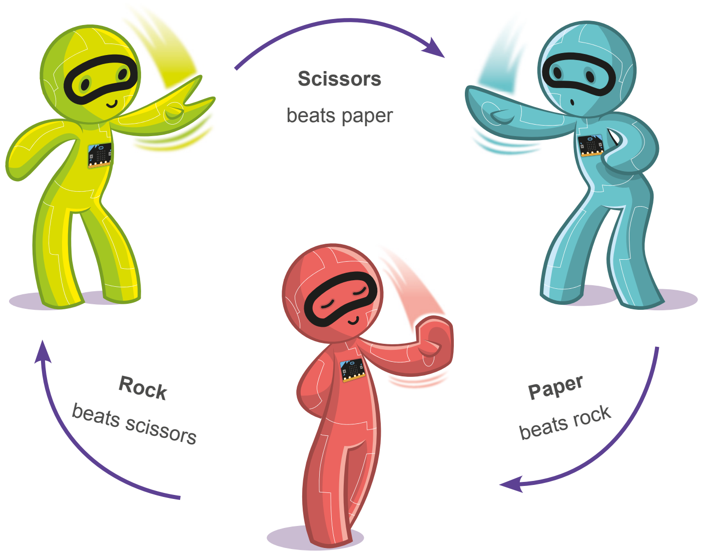


Figure 7.1: Rock paper scissors game: Rock beats Scissors. Scissors beats Paper. Paper beats Rock.

7.2 What you'll need

- **2 micro:bits**
- **1 whiteboard/board**
- **board markers/post-it notes**
- **1 teammate**

7.3 Programming: Rock, paper, scissors

To program this game, it is best to work with a teammate. Task 1 is for familiarising you with the game and will not use the radio. Starting from Task 2, you will start writing the parts of your program to play this game over the radio.

7.4 Task 1: Start with the simple game

Description: To familiarize yourself with the game, try the Rock-Paper-Scissors activity at https://www.microbit.co.uk/blocks/lessons/rock-paper-scissors/activity. Notice that the program gives a number to rock, paper, and scissors. For example, paper=0, rock=1, and scissors=2.

Instruction: Program the code shown on the Rock-Paper-Scissors activity page, and download it to your micro:bits. Play the game with a friend. You will each shake your micro:bits at the same time and then decide who wins using the rules shown in Figure 7.1.

7.5 Task 2: Handshapes over the radio with unicast

Description: To play the game over the radio, you will use button A to select paper, rock or scissors. You will use button B to confirm your selection and send it over the radio. Like in Task 1, use paper=0, rock=1, and scissors=2. You will send one of these numbers over the radio depending on the selection.

Instruction: Write a program to do the following:

1. Use button A to select paper, rock or scissors. Each time you press button A, it should alternately show an icon of either paper, rock or scissors.

2. Use button B to confirm your selection, and unicast it to your friend's micro:bit over the radio like you did in Chapter 5.

3. Add code for receiving a number. When you receive a number, show the corresponding icon on the display. For example, if you received 0, display the paper icon.

Test with your teammate that you can send and receive your handshape values over the radio.

7.6 Task 3: Fill the table of rules

Description: Your program, when it receives a number from your teammate's micro:bit, decides who wins.

Instruction: To decide who wins, compare the number you picked with the number you received. We have provided an incomplete example table to help you decide the result in Figure 7.2. Using this table, you compare *My hand* to *Opponent's hand*. For example, if both of these numbers mean *Paper,* it is a tie, and the result is a surprised face. But, if *My hand* is for *Paper* and the *Opponent's* hand is for Scissors, the result is a sad face. In contrast, if *My hand* is for *Scissors* and the *Opponent's hand* is for *Paper*, then the result is a *happy* face. Using the rules in Figure 7.1, fill the rest of the table.

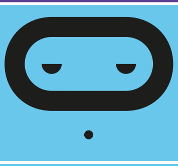
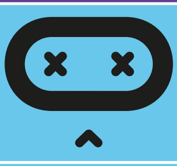
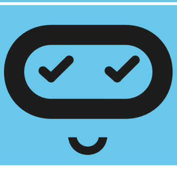
		Opponent's hand	
My hand	**Paper (0)**	**Rock (1)**	**Scissors (2)**
Paper (0)			
Rock (1)			
Scissors (2)			

Figure 7.2: Incomplete Rock paper scissors table

7.7 Task 4: Play the game

Description: Once you have filled the table, you need to decide how to program these rules in your code. Your program will:

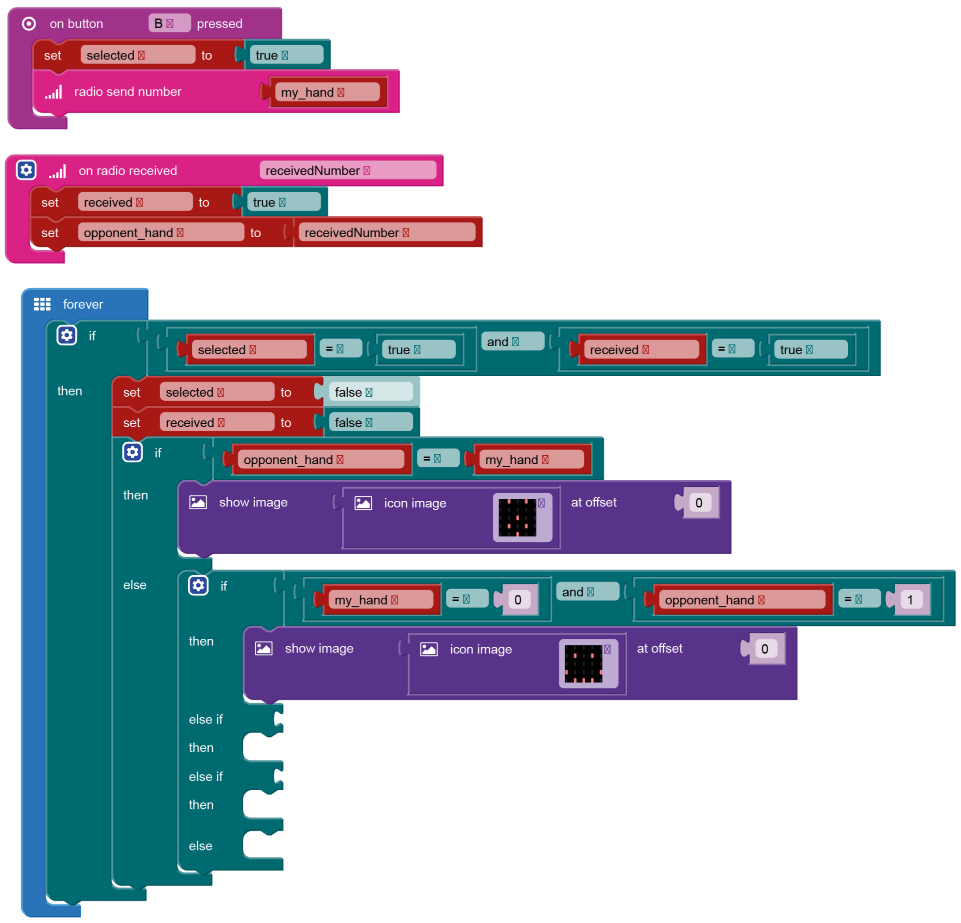
1. play the game based on Rock-Paper-Scissors rules (see Figure 7.1)

2. display a *happy* face if you won, a *sad* face if you lost. And if it's a draw, show a *surprised* face.

Instruction: Figure 7.3 shows a template for programming the table using the *if* block in the JavaScript Blocks editor *Logic* menu. Note that this is just a template and it is there to give you an idea of the structure of your program. For instance, your *on radio received* block will have to be different to do unicast communication (see Chapter 5).

You will notice in the template that we used two variables: *selected* and *received*. selected is set to *True* when you make the selection for your hand by pressing button B. *received* is set to *True* when you receive your opponent's hand. In the *forever* block, the game is only played when both *selected* and *received* are *True*. Once you enter the block to play the game, these variables are reset to *False* for the next round.

After you program the game, play it with your teammate! Who wins more often?



```
on button B pressed
    set selected to true
    radio send number my_hand

on radio received receivedNumber
    set received to true
    set opponent_hand to receivedNumber

forever
    if selected = true and received = true
    then
        set selected to false
        set received to false
        if opponent_hand = my_hand
        then
            show image icon image at offset 0
        else
            if my_hand = 0 and opponent_hand = 1
            then
                show image icon image at offset 0
            else if
            then
            else if
            then
            else
```

Figure 7.3: Rock paper scissors: A template for programming the rules

Exercise 1.
How might you expand your program to play rock/paper/scissors/lizard/spock? To learn more about this extension check the website: http://www.samkass.com/theories/RPSSL.html

7.8 Problems

Problem 7.1 How do you test a tie in your program?

Problem 7.2 How does the Table 7.2 change, if paper=2, rock=0, and scissors=1? Redraw your table.

Problem 7.3 To play with a different player, what do you need to change in your program? Remember you are using unicast to send your hand.

Problem 7.4 What happens if you send your hand to the other player before they pick theirs? Will there be a problem? Could they cheat?!

7.9 Resources

- **Flash game: Rock-Paper-Scissors: You vs. the Computer - http://www.nytimes.com/interactive/science/rock-paper-scissors.html**

8. HANDLING ERRORS: RETRANSMISSIONS

8. Introduction

In the previous chapters, you probably noticed that wireless communication isn't always reliable. In other words, not every message you send may be received by the other side. In this chapter, you will learn how to increase the chance your messages are received. So, what would you do if a message gets lost? This chapter will cover one simple but effective method: retransmissions.

In summary, you will learn about:

- **Wireless communication errors**
- ***Retransmissions* as way to improve reliability**

8.2 What you'll need

- **2 micro:bits**
- **1 teammate**

8.3 Background

In wireless communications, an error can occur for several reasons. For example, there may be physical obstructions, like walls, doors, and even people. The wireless signals lose power as they go through these obstructions and sometimes bounce off them! The more obstructions there are between a sender and a receiver, the more chance there is of an error. Also, if the sender and receiver are too far away from each other, they may not always be able to communicate. Imagine there are many obstacles between two people, they may not always hear what the other is saying!

Another reason for a wireless error may be *radio interference*. This is because wireless communication is broadcast (remember Chapter 2). This means that there may be many broadcasters, and their transmissions may collide at the receivers. These broadcasters *interfere* with each other.

Definition 1. — Interference.
In wireless communications, interference is any other signal that disrupts a signal as it travels to its destination.

Imagine, in a classroom, when everybody is talking at the same time. You will miss half of the things your friend says. Other people's signals interfere with your friend's signal on its way to you. In networking, this is a *packet loss*.

Definition 2 — Packet loss.
Packet loss happens when one or more data packets traveling in a computer network do not reach their destination. Packet loss is measured as the ratio of packets lost and the packets sent (see Equation 8.1).

$$\text{Packet loss} = \frac{\text{Packet lost}}{\text{Packets sent}} \qquad (8.1)$$

Also, if there is too much interference, you may receive messages incorrectly! Going back to the classroom example, this is like you hearing "Bat" when your friend is shouting "Cat". In networking, this is a packet error. Packet errors are measured as *packet error rates*.

Definition 3. — Packet error rate.
Packet error rate is the ratio of packets that have been received with one or more errors and the packets sent.

$$\text{Packet error rate} = \frac{\text{Packets with errors}}{\text{Packets sent}} \qquad (8.2)$$

In this chapter, we will cover one simple method to handle these errors, *retransmissions*, where the sender automatically retransmits messages multiple times to increase the chance of reception. In

Definition 4. — Retransmissions.
Retransmissions mean sending messages many times.

Figure 8.1, let's assume the sender knows that the communications medium loses half of its packets. In other words, the packet loss is 0.5 (or 50%). The sender micro:bit decides to send each packet twice, to increase the chance that its messages get through. The first packet is the transmission, and the second packet is the retransmission. So, the number of retransmissions is 1.

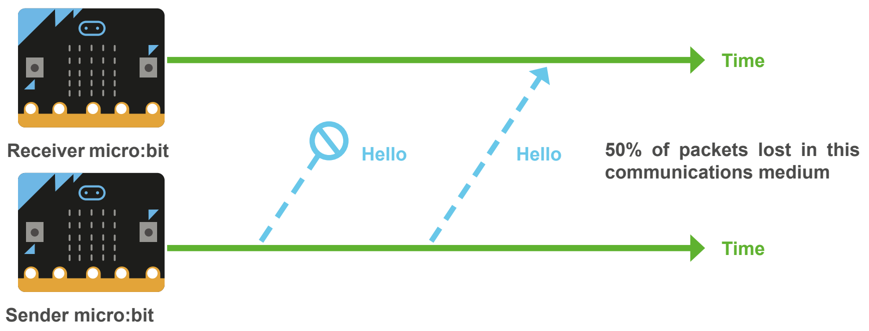


Figure 8.1: Retransmissions may increase message success. In the example, the sender sends each message twice by default. So, even if the first "Hello" failed, the second "Hello" was received by the receiver!

It is common to use retransmissions combined with another method. For instance, senders may choose to retransmit only when they are sure there has been an error. We will explore this option inthe next chapter, Chapter 9.

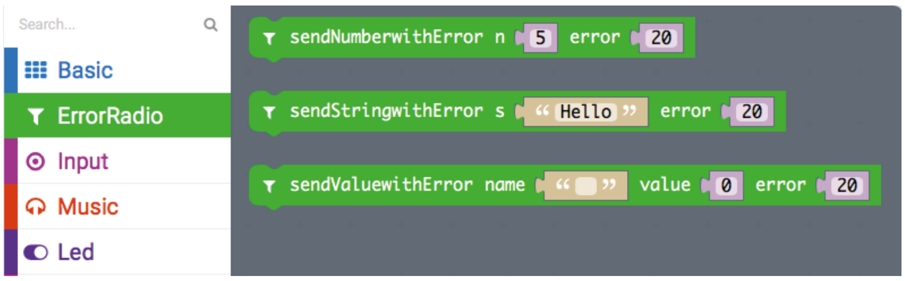
8.4 Programming: Retransmissions

This activity is best done with a teammate. You will start with creating packet errors in Task 1, and then test different packet error rates in Task 2. In Task 3, you will program the retransmissions solution to handle these errors. In this task, you will also run a series of experiments to measure how well retransmissions work.

8.5 Task 1: Create packet errors

Description: In wireless communication, packets may fail randomly. This may make testing your code for this chapter difficult. To test errors, you will use a custom block in the JavaScript Blocks editor to send messages with deliberate errors.

The ErrorRadio functions are like the Radio functions but have an extra error parameter. This parameter is set to 20 by default, which means, on average, 20 packets fail out of 100. So, the packet error rate is 0:2 or 20%.



```
sendNumberwithError n 5 error 20
sendStringwithError s "Hello" error 20
sendValuewithError name "" value 0 error 20
```

Figure 8.2: Error radio custom blocks

Instruction: To use the custom blocks in the JavaScript Blocks editor, import the ErrorRadio.hex file at https://microbit.nominetresearch.uk/networking-book/ErrorRadio.hex into your JavaScript Blocks editor. With your teammate, decide who will have the sender micro:bit, and who will have the receiver micro:bit. Follow the approach in Chapter 5 to put sender and receiver addresses in your packets.

You may copy and change one of the programs you have written for Chapter 5 to use the *ErrorRadio* blocks. Write a simple sender program that sends a number with an error. Download it to the sender micro:bit. Write a simple receiver program that receives a number and displays it on the screen. Download this program to the receiver micro:bit.

Change the packet error rate using these error values in your program: 0, 50 and 100. Test packet errors by observing the receiver display.

8.6 Task 2: Send a sequence of messages

Description: In this section, you will send a sequence of messages to the receiver micro:bit.

Instruction: Extend your program in Task 1, to send this sequence:

Start 1 2 3 4 5 6 7 8 9 10 End

You can send the *Start* and *End* using the normal Radio blocks to send them without an error. But remember that your micro:bit's radio may drop messages. So, there may be errors even in sending *Start* and *End*. No radio is perfect!

Extend the receiver program to count the number of messages it receives in this sequence. Run experiments setting the *error* parameter to 25, 50, and 75. Calculate the packet loss using Equation 8.1. Repeat each experiment three times. Fill the Table 8.1 with the result of your experiments. For example, when *error* is set to 25, and you received:

Start 1 5 6 7 8 9 10 End

this means you received 7 packets, and lost 3. Your packet loss is 0.3. The first row of the table is filled based on this example. Add in the values from your own experiment. Based on your experiment results, discuss with your teammate how the experiment results change as you change the value of *error*.

Error value	Experiment no.	Packets received	Packet loss
25	(example)	7	0,3
25	1		
25	2		
25	3		
50	1		
50	2		
50	3		
75	1		
75	2		
75	3		

Table 8.1: Experiment results

8.7 Task 3: Retransmit by default

Description: In this task, you will program automatic retransmissions at the sender side.

Instructions: Change your sender code from Task 2 to send each number in your sequence more than once. To try out your code, set *error* to 75. For example, by setting the number of retransmissions to 1, you will send the following sequence:

Start 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 End

This means the sender sent 20 packets in total, of which 10 are retransmissions.

Change your receiver code to count the unique numbers received. Count also the duplicates. Calculate the packet loss. For example, let's assume your receiver received, for the case of 1 retransmission:

Start 1 1 2 3 5 5 6 8 9 9 10 End

This means the receiver received 8 unique numbers (1, 2, 3, 5, 6, 8, 9 and 10) and 3 duplicates (1, 5 and 9). Note that the packet loss is 9 packets out of 20 (0.45). But with retransmissions, the receiver only lost 2 numbers out of 10 (it did not receive 4 and 7). Let's call this improved packet loss *information loss*. So, the information loss with retransmissions is 0.2. The first row of table 8.2 is filled in based on this example.

Run each experiment three times each, for different retransmission values and fill in the rest of the table.

Retransmission	Experiment no.	Unique packets received	Duplicates	Packet loss	Information loss
1	(example)	7	3	0.45	0.2
1	1				
1	2				
1	3				
3	1				
3	2				
3	3				
5	1				
5	2				
5	3				

Table 8.2: Experiment results

8.8 Extended activity

Exercise 1.
Based on your experiments, discuss with your teammate how the increase in retransmissions helps. In your discussion, answer the following questions:
- **How does the information loss reduce as you increase the number of retransmissions?**
- **Does the method guarantee all messages are received at least once?**
- **How would you improve the retransmissions method?**

Exercise 2.
Imagine you are going to survey packet loss at different locations inside a room using two micro:bits. Write a receiver and a sender program to measure packet loss. What do you observe? How does the packet loss change at different locations?

8.9 Problems

Problem 8.1 What is interference? Why does it happen?

Problem 8.2 If the sender sent 20 messages, and 11 messages were lost on the way to the destination, what is the packet loss?

Problem 8.3 If the packet error rate is 0.2 and the sender sent 40 packets, how many packets had errors?

Problem 8.4 Assume you do not know how many numbers there will be in a message sequence, but you know the numbers will start from 1, and will increment by 1. For example, the sent message sequence may be:

Start 1 2 3 4 5 6 7 8 9 10 11 12 End

What happens if you lose *Start* or *End* messages? Which is worse: the loss of a *Start* or an *End* message? If the only message you receive is a 4, what can you say about the number of messages lost?

Problem 8.5 Assume you do not know how many numbers there will be in the message sequence.

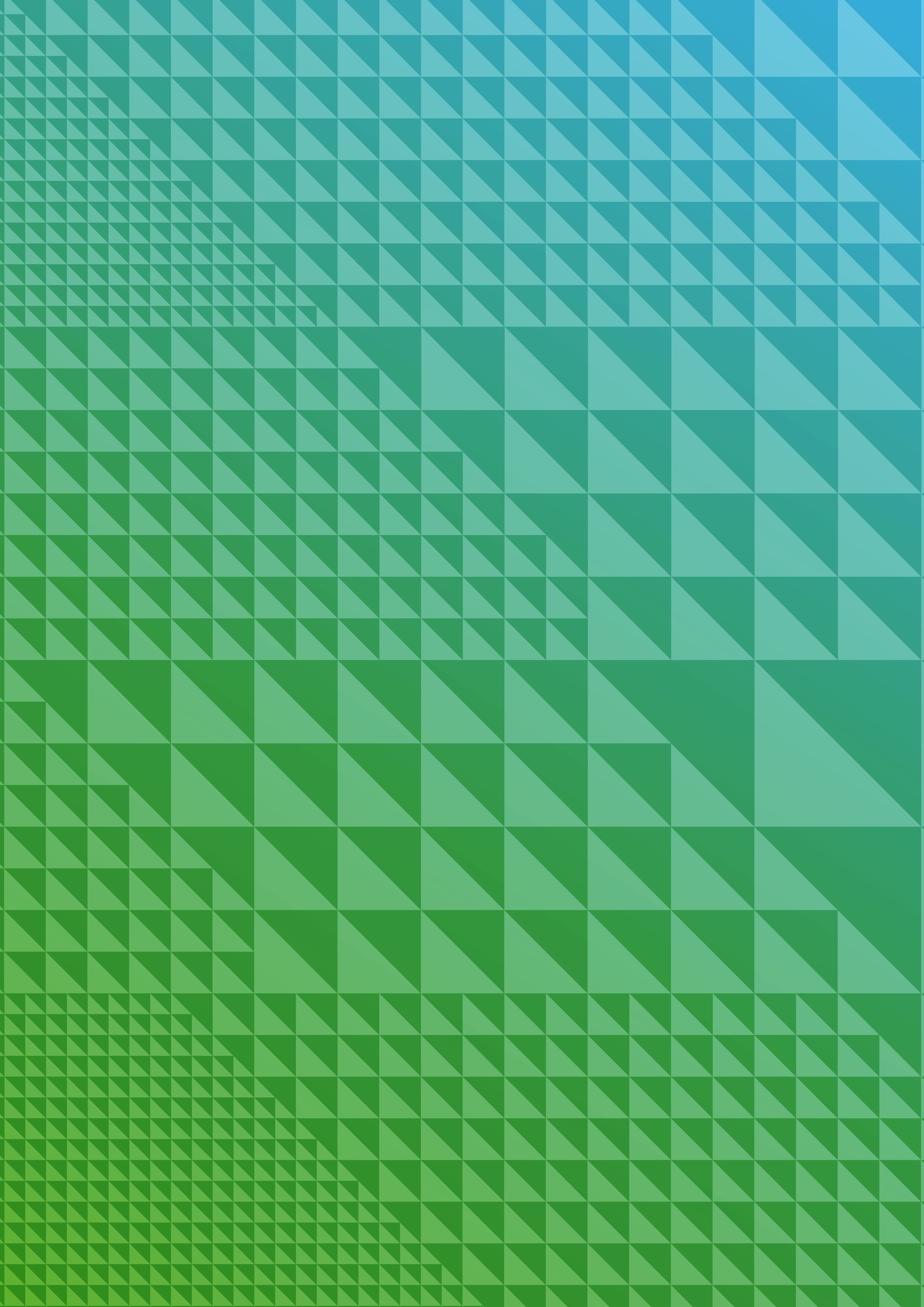
And they do not follow any order. For example, the sent message sequence may be:

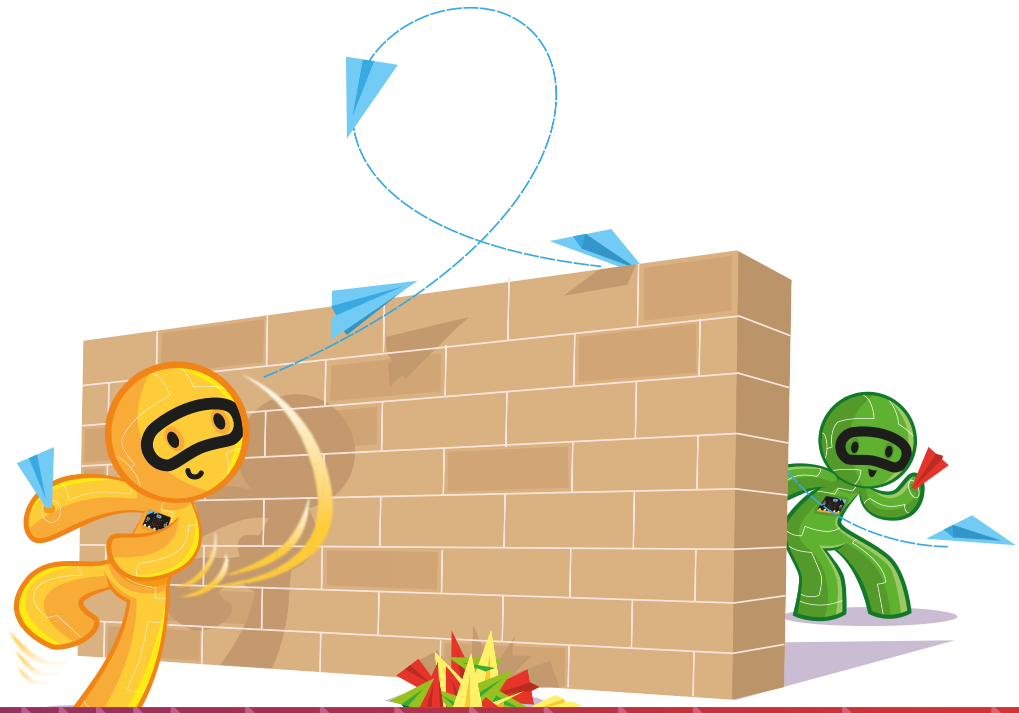
Start 3 5 10 2 End

What happens if you lose *Start* or *End* messages in the sequence? Which is worse: the loss of a *Start* or *End* message? If the only message you receive is a 5, what can you say about the number of messages lost?

8.10 Resources

- **Video: The Internet: Packets, Routing, and Reliability - https://youtu.be/AYdF7b3nMto**





9. HANDLING ERRORS: ACKNOWLEDGEMENTS

9. Introduction

In the previous chapter, you used retransmissions to deal with wireless transmission errors. In this chapter, you will improve on this by using acknowledgements. Doing this activity, you will learn several key methods and protocols for error control in networking.

In summary, you will learn:

- **The concept of *acknowledgements***
- **The concept of *Automatic Repeat Request (ARQ)***
- **The *Stop-and-Wait protocol***

9.2 What you'll need

- **2 micro:bits**
- **1 teammate**

9.3 Background

In the previous chapter, a message was transmitted multiple times even if the receiver already received an earlier copy. This is wasteful! You could have been transmitting new information instead of repeating yourself. This is also wasteful for the receiver, which needs to keep discarding the duplicates.

Definition 1 — Acknowledgement (ACK).
Acknowledgements are small messages that the receiver sends back, to tell the sender that it received a message. The sender then knows that it doesn't need to retransmit, and is ready to send the next message.

To avoid this, we will introduce a new concept called *acknowledgements*. If the sender does not receive an acknowledgment, it should retransmit its message.

But how long should the sender wait for an acknowledgement? This is specified by a *timeout*

Definition 2. — Timeout.
A timeout is the amount of time allowed to pass before the sender gives up waiting for an acknowledgement.

In other words, if the sender does not receive an acknowledgement within a timeout period, it will decide the packet must have got lost.

Acknowledgements are used in an error control method called *Automatic Repeat Request (ARQ)*.

Definition 3 — Automatic Repeat Request (ARQ).
Automatic Repeat Request is an error control method. It uses acknowledgements and timeouts to retransmit packets. Retransmissions may continue until the sender receives an acknowledgment, or a maximum number is reached.

ARQ is used both in the Internet and mobile networks.

In its simplest form, an Automatic Repeat Request uses the *Stop-and-Wait ARQ protocol.*

Definition 4 — Stop-and-Wait ARQ Protocol.
In the Stop-and-Wait ARQ protocol, the sender:
1. sends a packet
2. waits for the acknowledgement (ACK) but gives up after the timeout period
3. if timeout, goes to step 1
4. if ACK, gets a new packet, goes to step 1.

In *Stop-and-Wait protocol*, the sender cannot send a new packet until it receives the acknowledgement for the previous one.

So, how does the Stop-and-Wait protocol handle packet losses? In the following, we go through a few examples.

Figure 9.1 shows an example of a successful transmission. The sender sends "Hello" and the receiver responds with an ACK. The sender receives the ACK before the timeout ends, so it knows that the packet was received OK. Now, the sender can start sending another message.

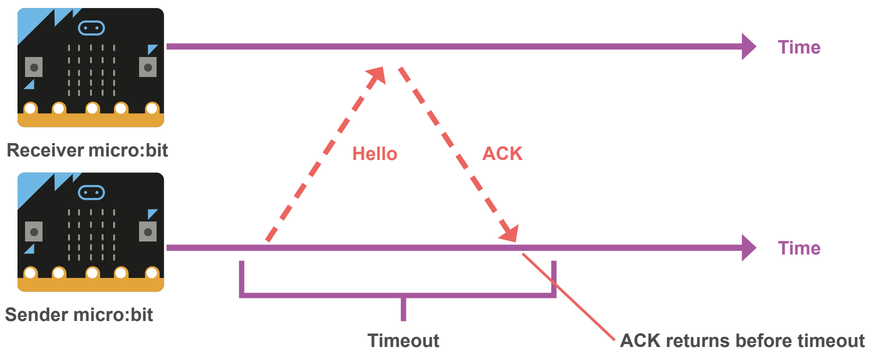


Figure 9.1: Stop-and-Wait ARQ protocol: The receiver sends an ACK back to the sender, so the sender knows that the "Hello" message arrived OK.

Now, let's look at some error cases. Figure 9.2 shows that the first message from the sender is lost.

So, the receiver does not send an ACK. When the timeout ends, the sender has not received an ACK. So, it retransmits the message. The second attempt is successful, and the sender receives an ACK on time (before a timeout).

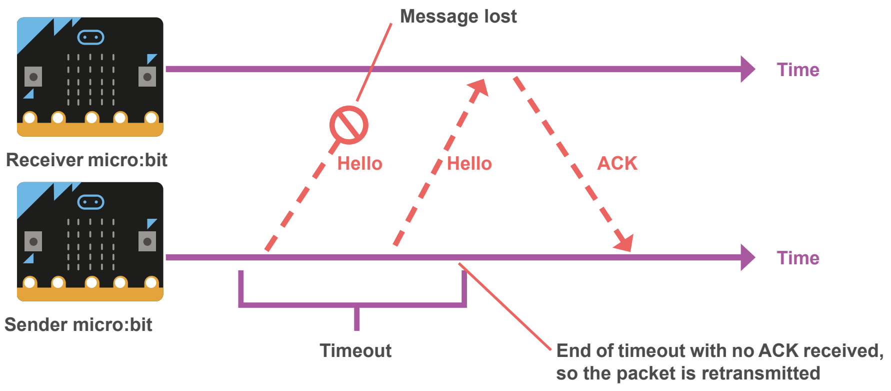


Figure 9.2: Stop-and-Wait ARQ protocol: The message gets lost, so the sender retransmits it.

Figure 9.3 shows an example where the message from the sender is received, but the ACK from the receiver is lost. Again, when the timeout ends, the sender has not received an ACK. So, it retransmits its message. The receiver receives the duplicate message and again, sends an ACK. This time the ACK succeeds and things can go as normal.

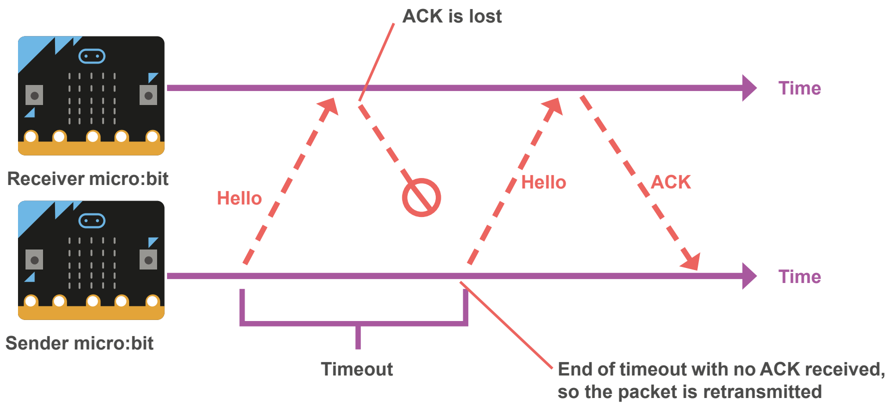


Figure 9.3: Stop-and-Wait ARQ protocol: The message was received, but the ACK gets lost, so the sender retransmits the message.

These examples show that the Stop-and-Wait ARQ protocol handles data packet and ACK losses quite well. But does it always work? Figure 9.4 shows a problem that can happen when messages or ACKs are delayed. In other words, the timeouts end before ACKs can be received.

In the example, when the sender sends the first “Hello”, the receiver receives this message and sends an ACK back. But the sender times out before it receives this ACK. So, it retransmits the second “Hello”. Then, it receives the delayed ACK message. But which packet does this ACK refer to? The first “Hello”, or the second? This confuses the receiver as well! Is the second “Hello” a new packet, or a duplicate? To solve this confusion, the protocol needs to use *sequence numbers.*

Definition 5 — Sequence number
A sequence number is a number chosen by the sender, and included in the packet header. When the receiver sends an ACK, it includes the next sequence number to tell the sender that it received the first packet and is ready for the next one.

For example, when the sender sends “Hello, 0”, this is a “Hello” message with a sequence number 0. On receiving this packet, the receiver will send “ACK, 1”, which says “I received packet 0, send me packet 1 next”.

We won't use sequence numbers in this lesson's tasks, but you could try adding them as an extended activity.

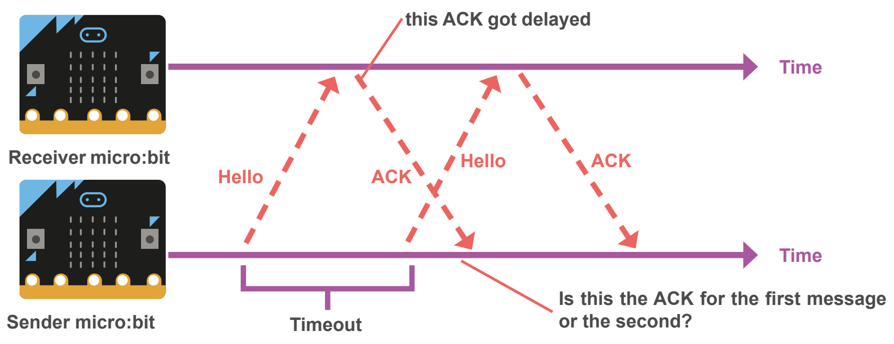


Figure 9.4: Stop-and-Wait ARQ protocol: What happens if a message gets delayed? It's not clear which ACK refers to which message.

9.4 Programming: Stop and Wait!

To program the Stop-and-Wait ARQ protocol, you will work with a teammate. Like in Chapter 8, you will use the custom *ErrorRadio* blocks to send messages with errors. The communication is unicast, so you will still use source and destination addresses in your messages like you did in Chapter 5. Do not forget that your receivers need to check if the received messages are addressed to them!

9.5 Task 1: Design your data and ACK packets

Description: Before you can send and receive any packets, first you will decide what your data and ACK packets should look like.

Instruction: Discuss what is the minimum information you should have in your packets. Create two string variables for data and ACK packets, using the Text blocks in the JavaScript Blocks editor.

9.6 Task 2: Timeout and retransmission

Description: To program the Stop-and-Wait, you need a timeout mechanism. After each transmission, you need to wait for the ACK or timeout. You'll need to decide how long the timeout should be.

Instruction: To do this task, you may either start from scratch or change your code from Chapter 8 for the sender micro:bit. At the sender side, program how to wait for the ACK. In the Basic menu, the pause function will be useful for the timeout mechanism. If your pause ends before you receive an acknowledgement, then you should retransmit the packet. If you receive the ACK before the pause ends, you should remember this information when the pause ends and use it to send your next message.

To test the program, you need to also program the receiver. The receiver sends an ACK packet for each data packet it receives.

9.7 Task 3: Testing the reliability of Stop-and-Wait

Description: In this task, you will experiment with the Stop-and-Wait protocol you programmed. For this, you will add a counter on the sender side to count the number of retransmissions. On the receiver side, you need a counter to understand the effect that acknowledgements have on retransmissions.

Instruction: Decide on a timeout/pause time. Send five numbers to your teammate's micro:bit using the Stop-and-Wait protocol. Run the protocol with different error values (25 and 75), repeating each experiment three times.

In the table, retransmissions are the number of times a packet needed to be resent. Duplicates are the number of times the receiver received unnecessary retransmissions. So let's assume that the sender sent the following:

1_1_1_2_2_3_4_4_4_4_5_5

The retransmissions are underlined: there were 7 retransmissions. And the receiver received the following:

1_2_2_3_4_5_5

The duplicates are underlined: 2 duplicates were received. The first row of Table 9.1 is filled as an example. Use your experiment results to fill in the rest. By comparing retransmissions to duplicates, discuss how good the protocol is at handling errors.

Error value	Experiment no.	Retransmissions	Duplicates
25	(example)	7	2
25	1		
25	2		
25	3		
75	1		
75	2		
75	3		

Table 9.1: Experiment results

9.8 Extended activity

Exercise 1.
Discuss how acknowledgements work better than using only retransmissions. Do you see any problems with using acknowledgements?

Exercise 2.
Discuss how the duration of the timeout period affects your protocol. For instance, what happens if your timeout is too short or too long? What happens if acknowledgements are delayed.

Exercise 3.
Research the "Alternating Bit Protocol", which uses 1-bit sequence numbers to help with problems discussed in Figure 9.4.

9.9 Problems

Problem 9.1 What does ARQ mean?
Problem 9.2 In the Stop-and-Wait ARQ protocol, if 10 packets are sent, how many acknowledgements are needed?

9.10 Resources

- **Video: The Internet: Packets, Routing, and Reliability - https://youtu.be/AYdF7b3nMto**

10. GAME 3: BATTLESHIP OVER RADIO

10. Introduction

In this activity, you will program the micro:bit version of a famous classic game called Battleship. Battleship has been played since World War 1 with pencil and paper[1]. A plastic board game was released in 1967, and now there are many electronic versions and apps[2].

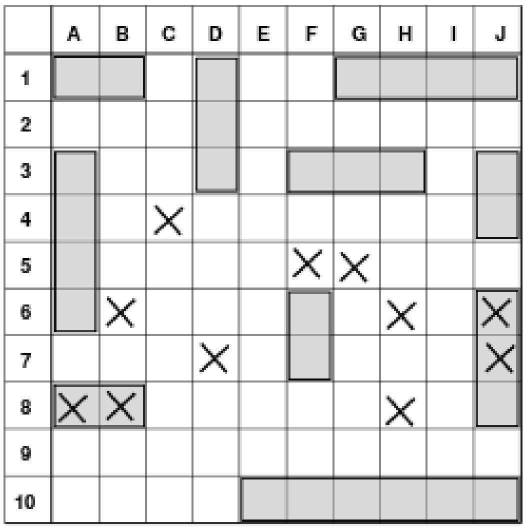


Figure 10.1: Battleship board.

Let's look at how this game works, using the example board in Figure 10.1. In this example, each player uses their own 10x10 board, and each player's fleet has 10 ships of different sizes (the grey rectangles) placed on the board: 4 ships of size 2, 3 ships of size 3, 2 ships of size 4, and 1 ship of size 6. arrangement of ships is of course hidden from the opponent. Once both players have placed their ships on their boards, they start guessing the position of (shooting at) their opponent's ships. In the example board, the crosses mark the shots of the opponent. Notice that some of these crosses did not hit any ships, and some of them did. The opponent has sunk the ship on squares 8A-8B. The ship on the squares 6J-7J-8J was hit twice, and another shot on 8J will sink it. The players also each keep a second board to mark the shots they have already tried.

[1] https://en.wikipedia.org/wiki/Battleship_(game)
[2] For examples, see: https://battleship-game.org and http://www.mathplayground.com/battleship.html

They record each hit and miss, so that they can decide which shot to fire next.

To program Battleship into your micro:bits, you will use your networking knowledge. This game requires unicast and bidirectional communication, which you learned in Chapters 5 and 6. If you program the variant in Exercise 10.1, you will use your learning from Chapters 8 and 9. In summary, you will practice:

- **The concept of *unicast communication*, *two-way communication* and *retransmissions***
- **Sending and receiving messages**
- **Button inputs**
- **Display and its coordinates**
- **Variables and random numbers**
- **Arrays**
- **Loops**

10.2 What you'll need

- **2 micro:bits**
- **1 teammate**

10.3 How the game works

Let's start by going over the different things we need to program Battleship. In the section above, you saw an example of the game, with a 10x10 board.

Using the micro:bit display as a Battleship board: Since a micro:bit only has a 5x5 display, your battleship board needs to be smaller! This does not allow for many ships or big ones. So, your fleet will be 5 ships, each with a size just 1.

When you fire a shot, you will need to know if it was a hit or a miss. So, we need to reserve the top row (as in Figure 10.2) to display hits and misses. If your opponent's micro:bit says you had a hit, your micro:bit will light the leftmost LED on the top row. If it was, unfortunately, a miss, your micro:bit will light the rightmost LED.

Since your micro:bit has a limited display, you won't be able to show your tries and misses in the display. Maybe that's a memory challenge that can be added to the game, or you can keep track of these with paper like the children who played the game in earlier times?

Firing shots: To fire shots, you will use the buttons: First you will select a row and a column number to choose a target, and then press both buttons together to fire the shot. Note that when LED coordinates are given as "(x,y)", x is the column number and y is the row number, and the numbers start at zero. For more information, see https://www.microbit.co.uk/device/screen. Button A will be used to select the column number and button B will be used to select the row number. So to fire a shot to (2,3), you will need to press button A twice, and press button B three times, and then press both buttons A and B together. To check your understanding, discuss with your teammate how you can send a shot to (0,4).

When you press both buttons to fire a shot, your program will send a message to your opponent's micro:bit. So for example, if you want to fire a shot at (4,4), you will send the coordinates (4,4).

When your opponent's micro:bit receives a shot, it will check whether it is a hit or a miss: It will send a message back with its radio saying either it is a "Hit" or a "Miss".

When you receive a "Hit", your micro:bit will light up the LED on the left corner of the top row.

When you receive a "Miss", your micro:bit will light up the LED on the right corner of the top row.

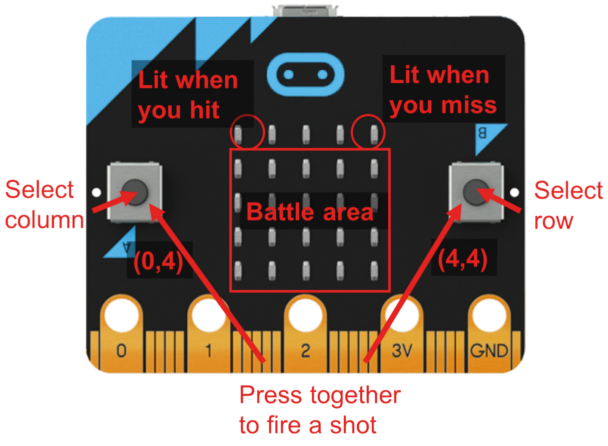


Figure 10.2: Battleship in micro:bit.

10.4 A sample game

Let's see how things will look like on your micro:bits. At the beginning, you will have all your battleships placed in the lower 4 rows, as in Figure 10.3. So, both players have 5 ships placed in the battle area.

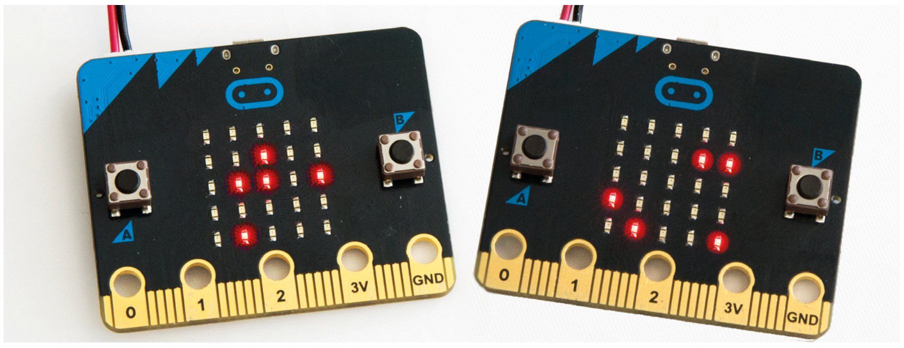


Figure 10.3: Battleship game: Initial stage with randomly placed ships.

The attacker (on the left) presses button A three times, and button B once. Pressing both buttons at the same time fires a shot and sends a message over the radio for the position (3,1). There is a ship on this location, and so this is a hit! So, in Figure 10.4, the leftmost LED in the top row of the attacker's micro:bit lights up. And in the opponent's display, the LED in the position (3,1) gets turned off, because this ship was sunk.

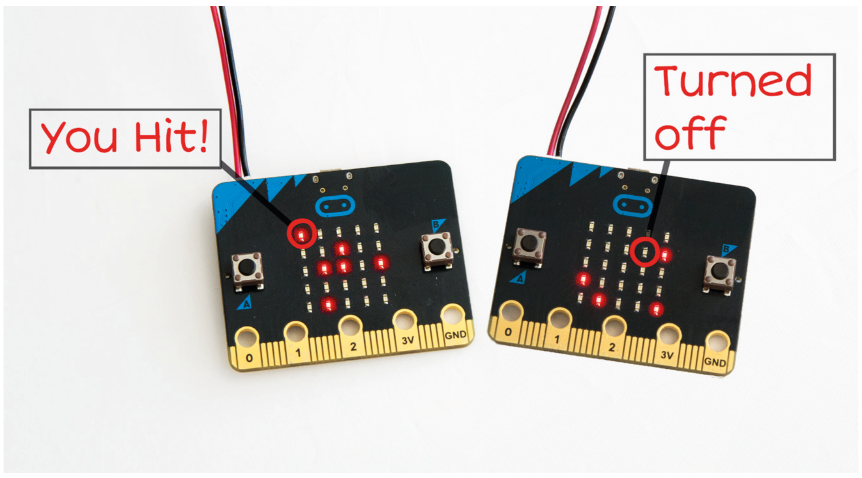


Figure 10.4: Battleship game: Success! You hit a ship!

Let's also look at a miss situation (see Figure 10.5). In this case, nothing should change on the opponent's board. But in the attacker's display, in the top row, the rightmost LED lights up to show a miss.

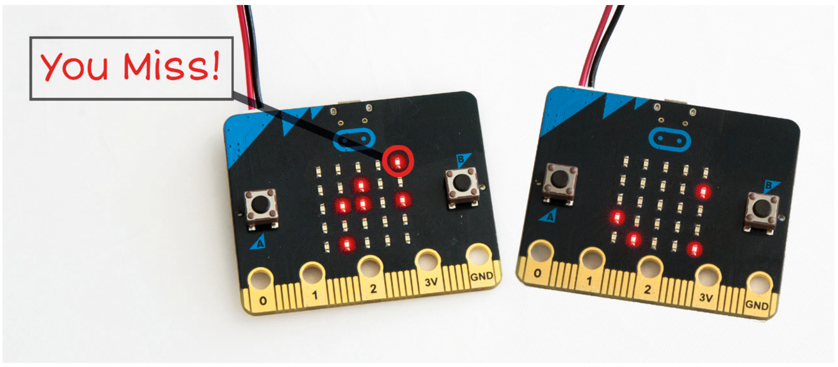


Figure 10.5: Battleship game: An unfortunate miss!

10.5 Programming: Battleship

Battleship is a two-person game. Both players can run identical programs, or you can each program your own version, as long as you agree on the details of the radio messages. When writing a more complex program like this, you will find it easier if you split it up into parts, and test each part as you write it. (This is a valuable skill as you learn more about programming!)

To help with this, we have split the program into four tasks: once you have completed the final task, you'll be able to play the game with your teammate. If you find any errors (bugs) in your program, work with your teammate to fix them until the game plays as described in Section 10.2.

10.6 Task 1: Setting up the game

Description: This part needs to take place before the game starts. You will place 5 ships on your board. Think about randomly placing 5 points in the battle area, which is a 4 x 5 matrix. Answer the following questions:

- **How will you represent the battle area in your program as a data structure?**
- **How will you select random coordinates (*column_number*) for 5 ships, where column_number is 0..5 and row_number is 1..5?**
- **How will you represent the information that there is a ship at each of these coordinates?**

You will also set up your radio and packet configuration to send unicast messages.

Instruction: Create the necessary data structures and variables that represent the ships in the battlearea. Set up your radio and packets for unicast communication.

Test whether your program displays 5 ships randomly on the lower 4 rows of the display, like in Figure 10.3.

10.7 Task 2: Firing a shot

Description: When button A is pressed, it defines the *column_number* for a shot. So you need to count how many times this button was pressed to get the *column_number*. When button B is pressed, it defines the *row_number* for the same shot. Again, count the number of times to get the row_number.

Important: If you don't press either button A or button B, the *column_number=0* and row*_number=0.* A shot with *row_number = 0* is a wasted shot because there cannot be any ships on that top row! Also make sure that if either button is pressed more than four times, it should start counting again from 0. In other words, the button counters should increment with each button press like this: 0, 1, 2, 3, 4, 0, 1, 2, 3, 4 etc.

Pressing both buttons together is firing the shot, so your program should send *column_number* and *row_number* over the radio to your opponent. Decide how to send this message in a packet, and agree on this with your teammate if you're writing separate programs.

Instruction: Program the button presses for A, B, and A+B. The program section for buttons A+B will send a radio message.

To test the correctness of your code, add a little test code so that when you fire, as well as sending a radio message, it also lights up the LED at (*column_number, row_number*). Use this to check that aiming is working correctly. This is just test code, so remove it once you're confident that it works.

10.8 Task 3: Receiving a shot

Description: When you receive a shot over the radio from your opponent, you will check whether you have a ship on the (*column_number*, *row_number*) of the shot. If you have a ship there, then itwas hit and sunk: You will send back a "Hit" message to your opponent, and remove the ship from the display. If your opponent misses, you will send a "Miss" message.

Instruction: Depending on how the packet was formatted, decode (*column_number*, *row_number*) from the received packet. If you have a ship on (*column_number*, *row_number*), it is a hit: Turn off the LED at that position. If you have a separate data structure as a variable to represent your ships, update that too. Send your opponent a "Hit" message. If it is a miss, send a "Miss" message to your opponent.

10.9 Task 4: Receiving the shot result: "Hit" or "Miss"

Description: Turn on LEDs in the top row depending on the result. If it is a "Hit", check if you reached 5 hits. Then you won: Display a smile!

Instruction: If you receive a "Hit", light the left LED of the top row (the LED in (0,0) position).

Update the count of your hits, and if you reached 5, display a smile! If the result was a "Miss", light the right LED of the top row (the LED in (4,0) position).

Test your program(s) with your opponent. To start with, it'll be easier if you can see each other's screens. You might find it helpful to put in some test code, like you did for the previous task. For example, you could print out "hit" or "miss" when you receive and decode a shot. You might even find it helpful to print out the coordinates of the shot when you receive the packet.

10.10 Extended Activity

Battleship game has many variations. See the Wikipedia site in Resources to read about the variations.

Exercise 1.
One variation of the game allows players to keep secret that a ship has been sunk. So, their opponent has to take further shots to confirm that the whole area is clear. This is a bit like having a packet loss! Remember how you dealt with packet losses in Chapters 8 and 9. How would you apply those concepts to this case? Discuss possible solutions with your friend. Then, program and test your new solution.

Exercise 2.
Imagine a variant when it takes 3 hits to sink a ship instead of 1 hit. How would your program change? Do you need to make changes on the sender side or the receiver side? How similar is this to using default retransmissions in Chapter 8?

10.11 Problems

Problem 10.1 Figure 10.6 shows randomly placed ships in a battle area. Which coordinates do you need to send to hit all the ships?

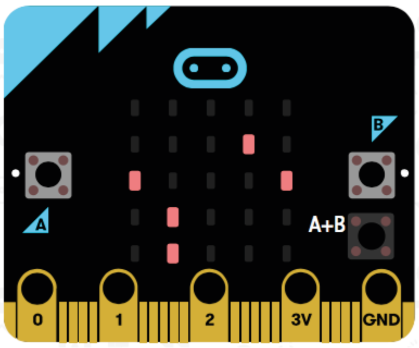


Figure 10.6: Battleship game: A random battle area

Problem 10.2 Figure 10.7 shows randomly placed ships in the battle areas of two micro:bits.

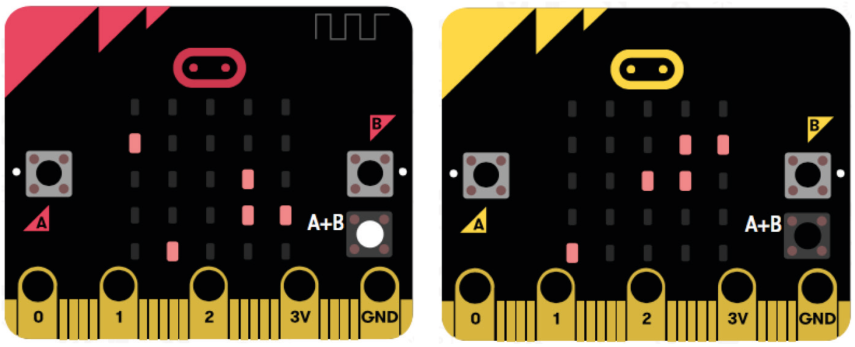


Figure 10.7: Battleship game: Two players

Table 10.1 lists all the shots that are fired from micro:bit 1 (left/red micro:bit) and micro:bit 2 (right/yellow micro:bit). Who wins?

Rounds	Micro:bit 1	Micro:bit 2	Result
1	(3,1)	(2,1)	
2	(0,3)	(0,1)	
3	(1,1)	(3,2)	
4	(4,1)	(3,3)	
5	(0,3)	(4,3)	
6	(2,2)	(0,3)	
7	(3,2)	(1,4)	

Table 10.1: Shots in each round

10.13 Resources

- **Battleship in Wikipedia - https://en.wikipedia.org/wiki/Battleship_(game)**
- **Online Battleship game 1 - https://battleship-game.org**
- **Online Battleship game 2 http://www.mathplayground.com/battleship.html**

INDEX

A

address 32

broadcast address 18

IP address 33

multicast address 23, 24

allow-list 35

B

bidirectional communication 38

binary 11, 12

bit 11, 12

byte 12

broadcast 11, 18

broadcast address 18

C

communications medium 11

custom block 49

E

electromagnetic radiation 18

electromagnetic spectrum 18

frequency 18

hertz 18

radio waves 18

speed of light 18

wavelength 18

G

group communication. 11, 23, 25

multicast address 21, 22, 25

H

header 29

I

interference 50

internet protocol 33

M

multicast 23

N

network 12

P

packet 33

ping 11, 39, 40, 41

protocol 33

R

reliability. . . . .

acknowledgement (ACK). 11, 55

Automatic Repeat Request (ARQ). .55, 56

retransmission 12, 50, 51

sequence number 57

Stop-and-Wait protocol 56, 58

round-trip-time 39

running time 40

S

signal 11, 12

T

timeout 56, 57

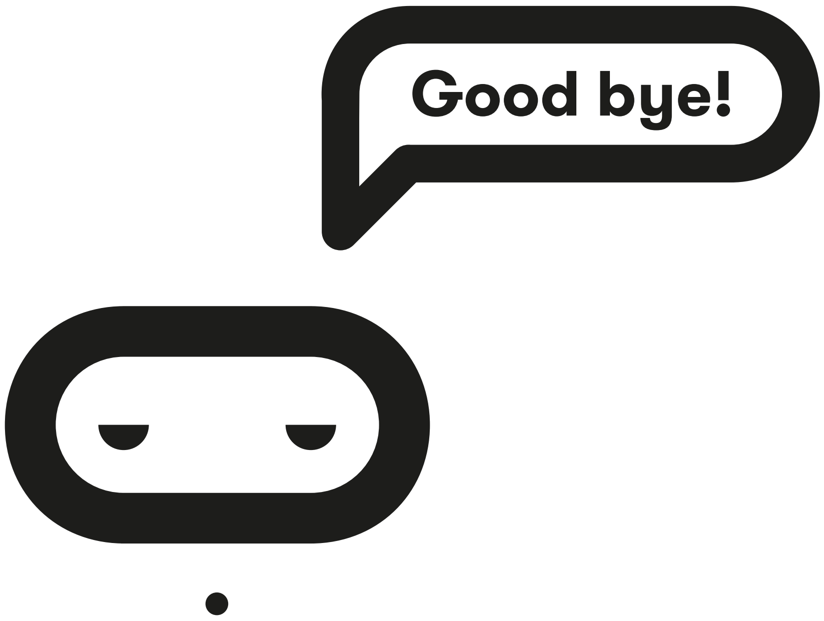
U

unicast 11, 32, 44

W

wireless 17

wifi. 17


Good bye!