# Introduction

If you’re reading this, it means I am dead, and the mission must continue.

Just kidding! It means Andrew McDonald has hired you and your team to continue the micro:bit hide-and-seek project. This project was a lot of fun to work on. Unfortunately, there were many things left incomplete… some pretty major things. It will be your job to fix these, do what we were incapable of, and then add your own improvements.

My name’s Morgan Bakelmun. I was effectively the team lead of Team Orion, and I handled most of the coding and logic of the micro:bit’s themselves. Andrew McDonald had to sign an agreement not to contact us once the semester ended (you have likely signed a similar agreement). I’ve decided to amend this a little bit; you can email me for help in the early stages. Taking over a project that’s unfamiliar to you is obviously a lot to take in. I’m going to try and go over the important parts here, but if you need an extra hand, email me at [morganbakelmun@hotmail.com](mailto:morganbakelmun@hotmail.com) . I’d love to see this project get a proper completion. Send me an email and I’ll get back to you as soon as I feel like it. Unless I really did die.

Alright, now on to the real stuff.

# Tips for this project

1. **Time management.** 90% of this semester is time management (math skills is the other 15%). You need to get really good at it. If you’re not, I recommend you learn. Fast. In this class alone you’re expected to dedicate 16 hours a week. It is effectively a part-time job. Seriously, set some time aside every week to be micro:bit time. You’re in for a rough semester.
2. **Manage your group members.** Easily the biggest issue our team had was with the inequality of contributions. We were a team of 6, and three members did over 90% of all the work. If someone isn’t contributing their fair share, kick their ass. Tell your professor they aren’t contributing. At the end of every sprint, you must write a PM Dashboard, which includes a breakdown of everyone’s contribution. Their contribution directly affects their mark. Be incredibly honest here, and don’t be afraid to give someone 0 if they did nothing. They’ll just have to do better next sprint.
3. **Play with the micro:bits.** You should all just do stupid shit with the micro:bits in sprint 1. Make some simple projects, so you learn how to code with them. Here are some starter ideas:
   1. When you press the “A” button, it shows a smiley face. “B” shows a sad face.
   2. A program that lets you scroll through several greeting messages (“Hello”, “How are you?”, etc.) using the A and B buttons. Adjust the speed so they scroll faster.
   3. Make a program using the A and B buttons to decrement and increment a number. When you press both buttons, it sends that number to a different micro:bit. Then, the receiving micro:bit displays it.
4. **Play our example game.** Read the example below to learn how to play. Load up each micro:bit with one of the code files from the “example” folder. Keep in mind that each micro:bit can only hold one hex file at a time. Play this game so you understand what it’s supposed to look like from a user perspective.
5. **Check out** [**MakeCode**](https://makecode.microbit.org/)**.**  I used this simple drag-and-drop interface to code much of the micro:bit code. It’s very easy to understand as it’s just simple coding language without actually writing code. The problem? Getting it onto the application. Turns out its conversion into python code only works on the MakeCode site or the Mu editor. So, it was no good for our purpose when we wanted the application to convert the Python code into hex code. We should have been using micropython; a python coding language specifically for micro:bit. Unfortunately, we discovered our error way too late in the semester. Fortunately, you can import the three hex files (microbit-beacon\_main.hex, microbit-seeker\_main.hex, and microbit-retriever\_base.hex) into the MakeCode editor and take over where we left off.

# Overview of the project

The readme goes into detail about how stuff goes down, but I’ll repeat the steps here.

1. The teacher/professor will open index.html. They will enter the number of seeking micro:bits (hereafter called “seekers”) and the number of hidden micro:bits (hereafter called “beacons”). When they hit enter, a new form will pop up.
2. On this second form, the teacher will enter a number of questions equal to the number of beacons and include their answers.
3. When they hit submit, a bunch of files will be downloaded onto their computer. The teacher will then load them onto their micro:bits.
4. The teacher will hide the beacons around the school/classroom, leave a slip of paper with the question near them, and give out hints on where to find the beacons. They will then give a seeker to each student/group of students.
5. Let the hunt begin! When the student gets close enough to a beacon (approximately 1 meter), the seeker will light up. They press both buttons to request the question from the beacon.
6. The seeker will say “Q#” where # is the question number. They will then use the A and B buttons to scroll through the options. When they think they have the right answer, they press both buttons to lock it in.
7. The seeker and beacon will show a checkmark if the answer is correct or an “X” if incorrect.
8. The student moves on to find the next beacon. Repeat from step 5 and continue until all beacons have been answered.
9. At anytime, the student can shake the seeker to see their score.

Other features of the application:

1. A color inverse mode for accessibility. It effectively also works as the dark-mode toggle. Let’s be real, everyone prefers dark-mode. But in a realer sense, dark-mode and color-inverse mode should be different toggles.
2. A way to find lost beacons using a special “retriever” micro:bit.

So that’s what it looks like from a user perspective. Here are some of the important parts of the back-end to understand.

1. The front-end application is a very simple web-page; HTML, CSS, and JavaScript. The first form takes in the number of seekers and beacons. The number of beacons is what is used to populate the next form.
2. For this second form, the questions typed in aren’t actually used for anything. We had plans to make a save function, load function, and print but never got around to it. All that really matters is the type of question (True/False, multiple choice with 3 or 4 options) and what the answer is (T, F || A, B, C, D). These will be inputted into the beacons.
3. When the submit button is hit, each seeker will be given a unique ID starting at 1. Each beacon will also be given an ID starting at 1. The beacons will also be given a “question\_type” and an “answer”. What happens is that two files, BeaconMain.py and SeekerMain.py, are copied and edited with their required values. These python files are then converted into hex files, which are sent to the user’s download folder. Note that we didn’t get that part to work, so the python files are downloaded instead. This is one of the **major issues** that you will need to tackle.
4. We wanted to include a “save questions” button and a “print questions" button but ran out of project time.
5. Now is a good time to review the internals of how the search for beacons works.
   1. When first turned on, the seekers will say “S1” and the beacons “B1”, where the number is their unique ID. This is very useful when testing; it lets you know what type of micro:bit you have, and you know it turned on and is working.
   2. The seeker contains an array of “options” that will be the choices for possible answers when they start answering a question. They also have a Boolean called “isSeeking”. This value starts as true, as they are seeking out beacons. Various actions will turn isSeeking to false, like answering a question or shaking to see the score. While isSeeking is false, the LEDs will not light up when they get close to a beacon.
   3. All seekers and beacons start on channel 0.
   4. The beacons are constantly sending out a signal of “Beacon1” where the number is their ID. This is what the seeker picks up when they are close. We added a “signalLimit” variable that ensures the seekers only pick up the signal within 1 meter or so of a beacon. This is why the readme recommends that beacons be at least 3 meters apart. The seeker will light up when it picks up on this signal. The more LEDs showing, the stronger the signal.
   5. Most importantly, we added a bunch of checks to make sure only the correct radio signals will be received. Seekers will never react to a signal sent by another seeker and beacons will never react to the signals from other beacons.
6. The internals of answering a question are very important. We nicknamed the system “TCP”, as it shared some similarities to the Transmission Control Protocol that you learned about in your networking class (even though it’s not really TCP). It works by sending a lot of “keys”, which are strings with a value. For this example, let's assume that the seeker has an ID of 1 and the beacon an ID of 2, where the question is a True/False where the answer is “True”.
   1. When both buttons are pressed on the seeker, while it’s in range of a beacon, it sends out “ask”=1 (1 is the ID). This is the “Syn” part of TCP’s three-way handshake.
   2. Upon receiving “ask”, the beacon first checks that the signal is of acceptable strength. Then, it makes sure that this seeker hasn’t already answered this question. It verifies that its ID is not in the array of known seekers. We wanted it to send a message to the seeker so the seeker would display a message like “Already answered” but couldn’t implement it.
   3. If the signal is the right strength and the seeker has not already answered this question, the beacon sends “S1T/F”=2. This is the “Syn-Ack”. Then the beacon switches from channel 0 to channel 2 (its own ID).
   4. The seeker picks up on the signal “S1T/F”=2. It verifies that “1” is it’s ID number. It then switches over to channel 2 as well. This is the equivalent of “Ack”, but not really, as no signal is sent. Now, both the seeker and beacon are on their own private channel. Seeker sets “isSeeking” to false so the LEDs don’t light up for no reason.
   5. The seeker parses through and finds “T/F” in the signal, so it knows this is a true or false question. It replaces the array “options” with “T” and “F”. It then displays the item at position 0 (which is “T”).
   6. The user can cycle through the items in the “options” array using the A and B buttons. When they press both buttons, the item from that position in the array is sent with their ID. So either “T”=1 or “F”=1. This is the “Fin” signal.
   7. The beacon receives this and compares it to see if it is the correct answer. If it is it sends “1” and shows a checkmark. Otherwise, it sends “0” and shows an “X”. It adds the seeker’s ID to its array of “found\_seekers”. It then switches back to channel 0 (“Fin-Ack”).
   8. The seeker receives the number. If it’s 1 it shows a checkmark and increases its “score” by 1. If its 0 it shows an “X” and does not change its score. In either case, it switches back to channel 0.
   9. If the seeker tries to answer again, nothing happens. The beacon will find its ID in the array of “found\_seekers” and ignore the “Syn” request.

So that’s the technical aspect of the interaction between the seekers and beacons. Now, to explain the retriever, which is a bit easier.

1. The retriever turns on and sends out the signal “Homing”. The retriever sends this signal at max strength. We’re not sure how far the max strength goes. We do know that it can reach from one end of C building to the other, so it’s far.
2. When any beacon receives “Homing”, they boost their power from the lowest setting to the highest. They then start flashing and sending out a repeating “Here” signal. The flashing is to make them more visible.
3. The retriever receives the “Here” signal and measures its strength. This determines how many LEDs light up on its screen. A weak signal means less while a stronger signal means lots of lit LEDs. So as you get closer to the lost beacon, more LEDs will light up. This allows you to play a game of hot-and-cold to find the beacon.

*Note: You’re probably wondering why we didn’t just have the beacons make a noise. That’s because we wanted to make this game completely possible with V1 micro:bits, which don’t have sound.*

# What we didn’t do

* The application does not properly edit the Python files and does not convert those files to hex code.
* Save/load questions not implemented.
* Print out for questions not implemented.
* Option for the user to include a “hint” for each beacon, and have those printed out too, or somehow displayed on the seeker.
* Retrievers should also be able to find lost seekers. It turns out that users frequently leave them behind.
* A way to automatically load micro:bits. The micro:bits are notorious for not having the hex files uploaded to them properly. In the command line, you can use robocopy to send the files, which is really reliable. See “debugging.txt” for details.
* We learned that users don’t understand what “press both buttons” means. They either hold both buttons or repeatedly press both buttons (this causes them to ask for the question, answer it with whatever the first option is, and then get locked out). From a developer perspective, I have no idea how to stop people from doing this. This data is from our interactions with users during our RE/ACTION showcase. Other members found that using more concise language such as “Tap both buttons together once” or doing a live demonstration were some of the more effective ways to get the message across.
* The seeker should show a message when it tries to answer the same question more than once.
* V2 micro:bits can use sound. Implementing sound features is optional.
* Pressing the A or B button on a seeker while not answering a question should do nothing.
* While a seeker is answering a question, other seekers cannot “ask” that beacon for its question. This can be a problem if the person walks away before answering. There should be some sort of time-out counter.
* Entering an invalid character in the seeker part of the form still generates the question form instead of giving an error.

# How the game actually works

So as you can see, the application side of things is incomplete. This means that there isn’t a way to create this game from the application, but you can still make your own game. Here are the steps.

1. Make your own question sheet in Word. Make sure all questions are either true/false or multiple choice with two, three, or four possible answers. Make as many questions as you have beacons. Keep in mind that you need at least 1 micro:bit to be the seeker. Print this sheet out when you’re done.
2. Go to <https://makecode.microbit.org/> and make an account. Or you can use your Algonquin account. On the main page you’ll see an “import” button. Click it.

A grey and black text

Description automatically generated

1. Import “microbit-beacon\_main.hex” from the microbit/hex\_files directory of the project. You should see the MakeCode editor appear. Find the “on start” function:

A screenshot of a computer

Description automatically generated

1. There are 3 things you need to edit in each beacon.
   1. “beacon\_id” should be the question number. Each beacon will have a unique ID.
   2. “question\_type” needs to be “T/F” for a true/false question or “M2”, “M3”, or “M4” for a multiple choice question. The number denotes how many options there will be.
   3. “answer” is the letter for the correct answer. It will only be “T”, “F”, “A”, “B”, “C”, or “D”.
   4. After changing these 3 variables click Download in the lower left corner. Repeat steps a through d for each question on your sheet.
2. Import “microbit-seeker\_main”. For the seekers, the only thing you need to change is seekerId. Each seeker needs a unique ID.
3. All of the required files should be in your downloads folder. You can now plug in a micro:bit and copy/paste one of the hex files. Then eject and unplug it. Repeat until all seekers and beacons are loaded up.
4. In the event that you get an uploading error, a sad face will show on the micro:bit, followed by an error code. Likely 541. Go into debugging.txt and try the solution in there.
5. Your game is now set up!