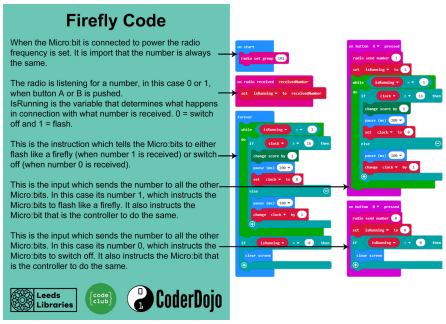
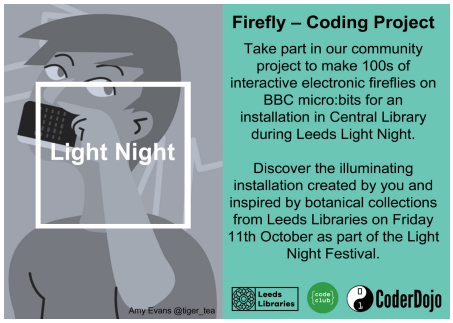
Lightning ‘Bugs’

A micro:bit makecode project (mentor guide)

In the real world, no software is ever perfect - especially when it’s first written. Maintaining most software involves an ‘endless cycle’ of discovering and fixing bugs (artistic license for a pun goes here).

This workshop project is designed to follow on from the Firefly coding workshop, where hopefully several micro:bits were programmed to create an awesome light display for a local public event.





Young coders are told that 2 problems have been found with the Firefly code - one that would allow ‘Mischievous Malory’ to take control of the artistic light display and another that makes devices unresponsive when Alice keeps pressing button A.

Young coders are challenged to complete each of the following stages. How they achieve this can be left up to them.

Perhaps in ‘competitive’ teams, their **4 challenges** are:

1. Use a Firefly coded micro:bit to demonstrate the Alice problem (including verbal ethical disclosure).

2. Program a micro:bit with Malory’s code, then demonstrate control of a Firefly display.

3. Rewrite the Firefly code to fix the Alice problem, reprogram the display devices, then demonstrate that it’s been fixed (coders figure out how to prove this).

4. Rewrite the Firefly code to fix the Malory problem, reprogram the display devices, then demonstrate that it’s been fixed (coders figure out how to prove this).

Explaining the Alice problem:

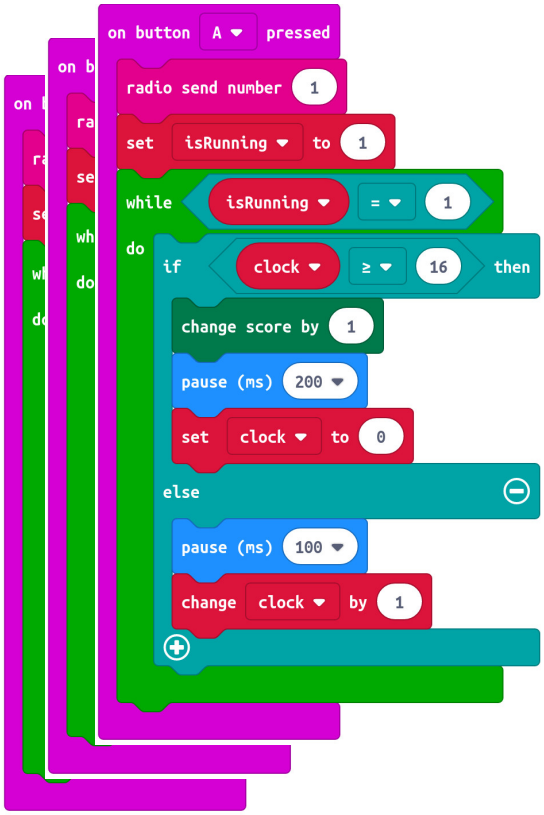
Looking at the Firefly code, the *event* of a button A press causes the code in the block to be executed.

The potential problem in the block is the *while-loop* that will carry on repeating until the *isRunning* variable in not equal to 1.

But what happens when button A is pressed **again**, while *isRunning* is still set to 1?

Rather than not letting anything else happen until the button A press code has finished, or stopping the button A press code that's already running, the micro:bit handles this situation by allowing the unfinished code to continue executing, but only after it's been sent to the *background*.

An identical block of code from the second button A press then starts executing in the *foreground*.



If button A is pressed again and again, with each while loop still going, we get a set of these copies all executing in the background. Luckily, there's a limit to the amount of memory that is reserved to keep track of all these copies whirling away in the background.

Corresponding background processes are also being executed on all other communicating devices.

Only when an event (button B press) sets the isRunning variable to 0 (ie not 1), will the currently executing foreground button A block be allowed to end. It's place is then promptly taken by the executing block that was last pushed into the background.

So Alice basically has to cancel out each of the background button A press code using button B.

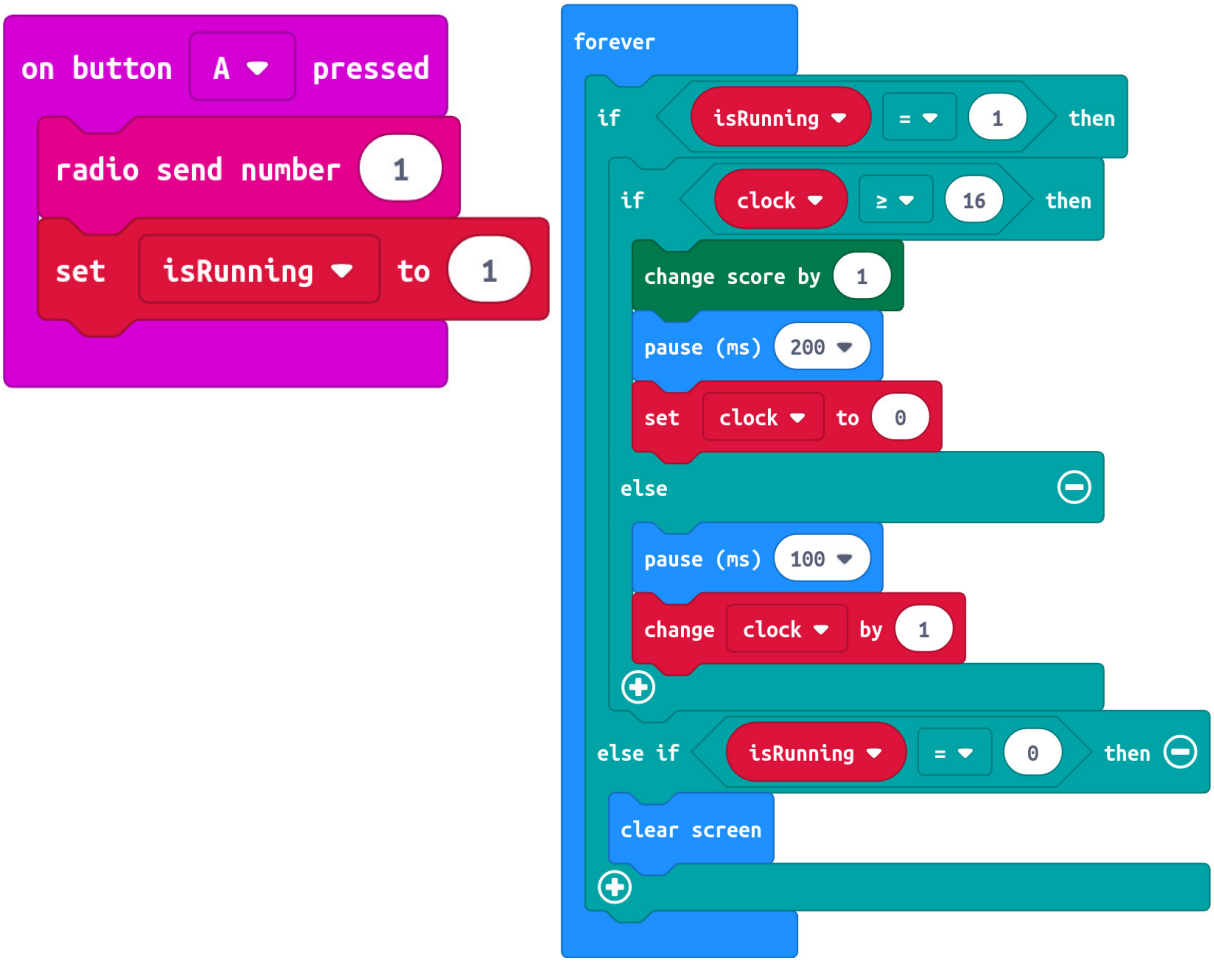
Resolving the Alice problem:

Short version:

The only cause of the Alice problem is the button A while-loop, and its' termination condition - needing a button B press).

The LED lighting code *within* the while-loop in button A is not even needed! All button A (and B) need to do is to change the *isRunning* variable and the forever-loop block will act accordingly.

In fact, the while-loop condition in the forever block is also unnecessary since when contained within the forever block, the *if control block* will light the LEDs for an infinite length of time.

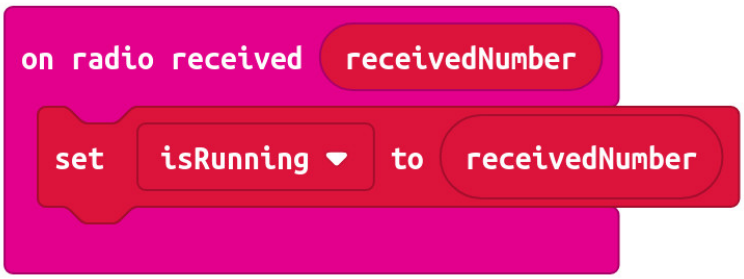


[firefly_v2 makecode image]

Explaining the Malory problem:

The Firefly code tells the micro:bit radio to listen for, then accept ANY number it receives, then immediately start doing stuff with it - all without first checking that it's one of the only two numbers (zero and one) the program *knows* what to do with.

When a number is received via radio, the program immediately sets the *receivedNumber* variable to the value of the number and then sets our *isRunning* variable to the value of *receivedNumber*.

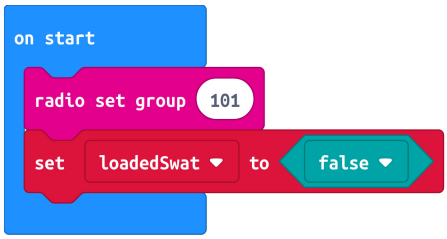


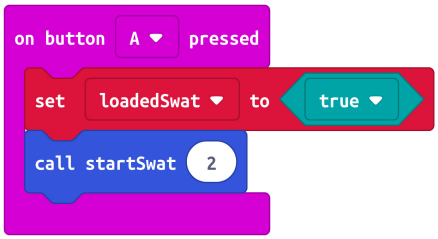
So, when Malory writes her program to send a number *other* than zero or one, it's like she's hired someone to go into Halton library to return a 'bag of fish and chips'! During the time it's being politely explained that the library "could not have issued this bag of fish and chips", no genuinely issued items can be returned.

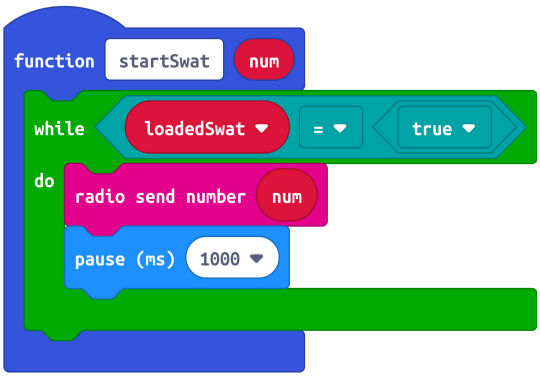
Malory smiles as she then thinks - "what if I get everyone in Leeds to buy bags of chips, and try to return them to the Halton library desk?" (All politely queued of course). "I reckon Halton library won't be able to do *anything else* until I issue the instruction for them to stop".

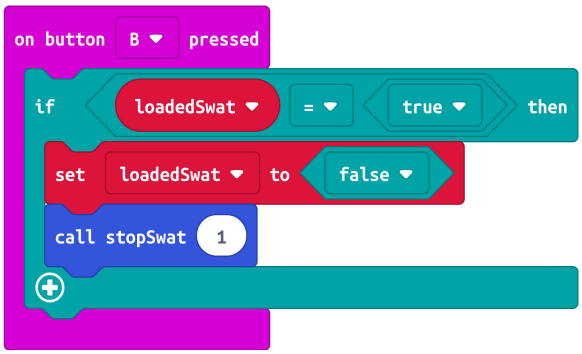
Malory's program:

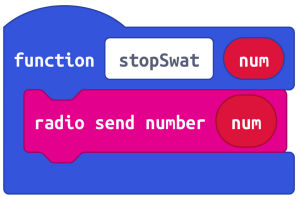
Malory designed her program to use button A to start a Denial of Service (DoS) attack on the Firefly display, and button B to stop the attack (whenever she feels like it).







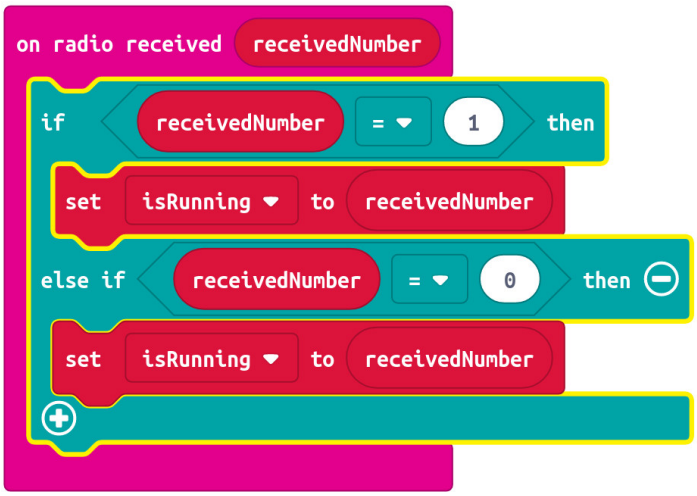




Resolving the Malory problem:

The program just needs to check what's coming in by radio and immediately throw away (or just ignore) any number that it's not expecting.

Librarians could recruit a volunteer to stand at the library entrance and check what the person at the front of the queue is trying to return. If it's anything other than the expected items, that person is ejected from the queue, so cannot enter the library to tie up valuable resources. (Note: As these queued people have agreed to being controlled by Malory, it's technically anti-social behaviour!)

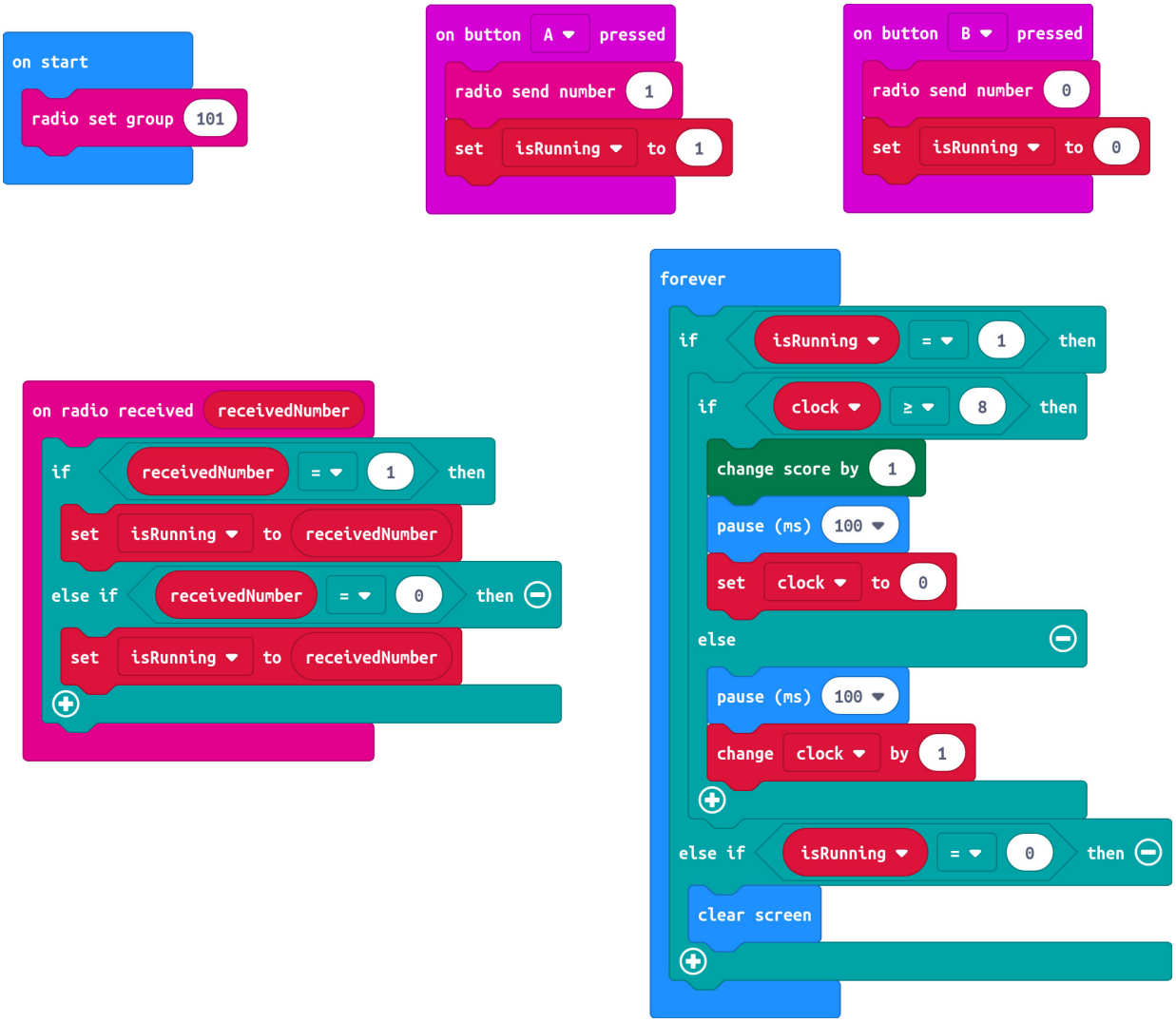


[Firefly_v3 - just the radio block]

Other Changes to the program:

Patched version of the Firefly program ready to be released as an update.

Button B event simplified and since there's now only one block of LED lighting code running, *clock* period decreased from 16 to 8:



Upgraded version to be released and marketed as lightningBug.

Puts the LED lighting code into functions and uses more meaningful variable names.

Decreases *timer* period to 7.