Handling inputs

Programming for beginners with the BBC micro:bit

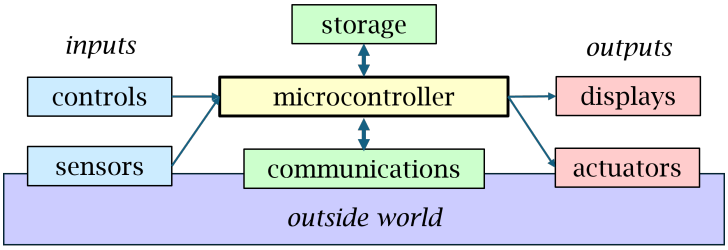
John Davies

2024 July 16

1 Introduction

Introduction

Our first programs simply ran by themselves. This is unrealistic: every program for an embedded system has inputs and outputs of various types.



How does a program react to the changing state of its inputs?

We shall look at two ways of handling inputs in programs.

I have divided inputs into two types.

- **Controls** are operated by the user, such as buttons.
- **Sensors** tell the system about its environment, such as its temperature.

These are less distinct to the microcontroller. For example, a common type of control is called a *touch sensor*. The micro:bit has one of these, the golden logo near the top centre of the front.

Likewise, I divided outputs into two types.

- **Displays** inform the user, such as LEDs, more sophisticated visual displays and sound.
- **Actuators** do something to the environment. The micro:bit doesn't have any of these itself but the microcontroller in a microwave oven would control the turntable and the power to the magnetron.

The micro:bit can communicate with other systems over USB or Bluetooth as well as by other methods that I shall not describe. By *storage* I mean something outside the microcontroller itself and the micro:bit does not come with this, but it could be added.

2 Traditional approach

The traditional approach: if–then–else decisions

A program should change its behaviour when an input changes. This is traditionally done with a construction called an **if–then–else statement** (or block).

Pseudocode:

if some condition is true **then** do this **else** do that

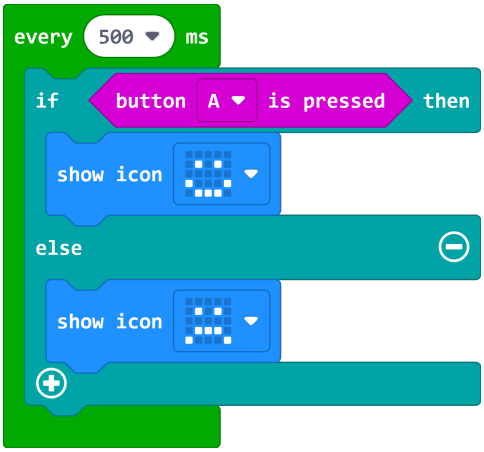
For example:

if button A is pressed **then** display a happy face **else** display a sad face

This translates directly into blocks.

Pseudocode is not real code but something closer to everyday language.

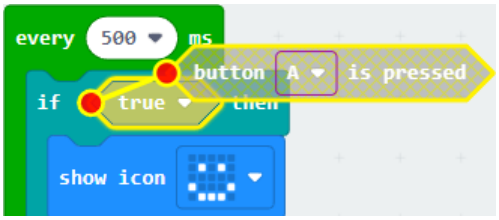
Blocks for button and faces



Construct this as before with a couple of new features.

Construct the program

1. The every block is similar to forever but we can control how often it runs. It is in the Loops section of the toolbox.
2. The if–then–else block is in Logic and you know where to find show icon.
3. The purple lozenge for button A is pressed is in Input. (It is not the same as on button A pressed, which we shall use later.) Drop a copy on the workspace.
4. Drag button A is pressed over the true lozenge in the if–then–else block to replace it. Both lozenges show red dots on their left points, which are joined by a yellow line as they approach.



Let go of button A is pressed and it snaps into place.

There is also an if–then block, without the else. It is also possible to extend the tests using the ⊕ and ⊖ signs: if … then … else if … then … and so on. We shall use this later.

Test the program

Before testing your program, change the time in the every block to its minimum, 100 ms, so that the loop runs as rapidly as possible (10 times per second). First try the simulator on the screen.

1. The display shows a sad face initially.
2. Hold down button A. You need to hold it, not just press quickly and release. The display changes to a happy face after a short delay.
3. Release the button again and the display reverts to the sad face, again after a short delay.
4. Experiment by pressing the button for different lengths of time. Does the face always change?

The delays arise because the loop runs only every 100 ms and this is when the button is tested. If you press and release the button within 100 ms it may be missed altogether. You can make this worse by increasing the time interval.

Speed up the program

Is 100 ms a short enough time to avoid irritation? You can make the loop faster by replacing the every block by forever. Do it this way to avoid having to reconstruct the program.

1. Drag the if–then–else block and its contents out of the every block and park it on the workspace.
2. Delete the every block and replace it with a forever block from the toolbox.
3. Drag the if–then–else block into the new loop to reconstruct the program.

You might expect the forever loop to run continuously but in fact it is similar to an every loop with a delay of 20 ms. This allows time for other parts of your program to run. Is it fast enough?

This style of programming, where you test the button explicitly, becomes cumbersome with more inputs. We'll take a more modern approach next.

I find it surprising how even a short delay makes a system feel unresponsive. The delay in the forever loop is explained in MakeCode blocks have a built in pause.

If you have written programs in the past, perhaps to do some sort of calculation, you probably find the concept of loops and if–then–else constructions familiar. You might be concerned

about the forever loop, though, because an 'infinite loop' that repeats without limit is a disaster in a traditional sort of program: it means that the computer has become stuck somewhere and cannot complete its task. Simple programs only required input from the user at times defined in the program.

A program to run on a computer embedded in hardware is quite different. It has to keep running for as long as power is applied, hence the need for an infinite loop like forever. Input may arrive at any time, depending on the whim of the user, so the program has to check continually. The structure of the program depends less on the calculations to be performed than on waiting for inputs and interpreting them.

I wrote programs for the Apple Macintosh in the late 1980s, where the main loop had to check for all possible inputs in the same way that we have just checked a single button. It was indeed cumbersome and *object-oriented* languages such as C++ were developed to improve such programming.

3 Event-driven programming

Event-driven programming

This is a style of programming that enables us to concentrate on the important aspects – how to handle events – and leave the details of detecting the event (including the infinite loop) to the MakeCode editor.

- An **event** is a change in some input, which may be provided by a user (buttons) or come from the hardware (timer, communication).
- A **handler** is the part of a program that we write to respond to the event.

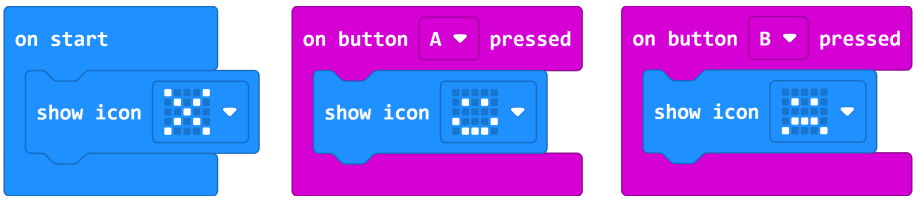
That's the jargon – the practice is much simpler!

Blocks whose name starts with 'on' act as the framework for event handlers, such as `on button A pressed`.

Buttons and faces

Here is a program to display a:

- cross at the start, before any button has been pressed
- happy face when button A is pressed
- sad face when button B is pressed



Try it out. Exactly what action triggers the handler and causes the corresponding face to be displayed? It is probably not what you would expect.

Exactly what triggers `on button A pressed` is explained in the reference manual for On Button Pressed:

- For button A or B: This handler works when the button is pushed down and released within 1 second.
- For A and B together: This handler works when A and B are both pushed down, then one of them is released within 1.5 seconds of pushing down the second button.

So the event is not triggered by *pressing* the button but by *releasing* it, provided that you do so quickly enough.

You might have noticed that we cannot make a program with the same function as that with the if–then–else loop because MakeCode provides only an on button A pressed block, not an on button A released. There is a closely related block, on pin P0 pressed, which detects a signal on the 'pin' at the bottom of the micro:bit – actually an area of gold contact labelled 0 with a hole. (The micro:bit has three such pins, labelled 0, 1 and 2.) This has a complement on pin P0 released.

If you look on the page in the reference manual for inputs you will find other signals that are not associated with handlers at all: temperature and compass heading, for example. These vary continuously rather than suddenly so it would not be useful to associate them with a single event.

How does the program know that a button has been pressed, given that we no longer have an explicit loop that continually checks its state? In reality there has to be such a loop, called the *event loop*, which checks the state of all active inputs and launches the corresponding handler when required. This loop is run regularly by a piece of the program called the *scheduler*, which is provided by MakeCode. The scheduler is described in The micro:bit – a reactive system, which explains as follows.

> The scheduler uses a timer built into the micro:bit hardware to interrupt execution every 6 milliseconds and poll the inputs, which is more than fast enough to catch the quickest press of a button.

'Poll' means to check an input regularly.

Extend the program

Try some of the other event handlers. You can simply add them to the program and display different emojis (or do something else).

- on button A+B pressed – both buttons simultaneously rather than individually.
- on shake – as you would expect.
- on logo pressed – the golden micro:bit logo near the top centre of the front is a touch sensor and has other options.
- on loud sound – yell at your micro:bit! The level of sound can be adjusted.

The Pet hamster project on the MakeCode home page shows an application of event handlers.

To go further we need to add **variables** to our program.

I found that the logo did not behave as I expected. The page in the reference manual for on logo event does not describe the events explicitly.