Advanced features and techniques

Programming for beginners with the BBC micro:bit

John Davies

2024 July 17

1 Introduction

Introduction

Our final session is for you to construct your own programs on the micro:bit.

Here is some more information that may help you to move forward and to understand how your program is executed by the computer in the micro:bit.

- How to use the radio.
- How to use functions and include comments in your programs.
- Text-based programming languages and MicroPython
- What goes on behind the scenes of your program?

2 Radio

The micro:bit radio

The radio in the micro:bit can send packets (short messages) to another micro:bit.

An obvious question is: how do you know which micro:bit(s) will receive the message? This controlled by setting a **group** in the radio, which you should do at the start. All radios in the same group will receive a message but others will not.

This is something like tuning the frequency on an old-fashioned radio. The group can go from 0 to 255 and must be the same for the transmitter and receiver.

Sending and receiving

The simplest blocks send or receive a number or a string (short text):

radio send number	—	on radio received *receivedNumber*
radio send string	—	on radio received *receivedString*

Here is how they are used to send a simple text message.

You need to program two micro:bits to test this, set to the same group. The send block on one activates the receive block on the other.

You could put only the send block on one micro:bit and the receive block on the other, in which case a message can be sent only one way. But it's easier to download the same program, with both blocks, onto the two micro:bits so that you can send in either direction. The on radio received block puts the message that it receives into the variable receivedString, which you can copy into the show string block.

Another pair of functions communicates both a name and value. Further details are on the reference page for the radio.

Ideas to try

The Make it: code it web page has numerous suggestions for exercising the radio.

- The teleporting duck appears to send an image from one micro:bit to another when the board is shaken.

  I enhanced the program so that the duck appears to scroll off one screen and onto the other, but this is far more complicated than the radio!

- The indoor–outdoor thermometer might be more useful.

  One micro:bit is put outside and sends its temperature at regular intervals to the inside micro:bit. This can display either the inside or outside temperature.

- Different types of remote burglar alarm.

- Perhaps we could all try the Fireflies program

3 Functions

Functions

A cookery book does not repeat the instructions for common requirements such as pastry every time they are needed. Instead, the recipe for apple pie says something like 'Make 200 g of short crust pastry (page 218)'.

The same is done with a **function** in computer programs:

- The instructions on page 218 are called the **definition** of the function (short crust pastry here).
- The recipe for apple pie **calls** the function and passes the **parameter** 200 g, which specifies how much pastry to make (although this is not always needed).

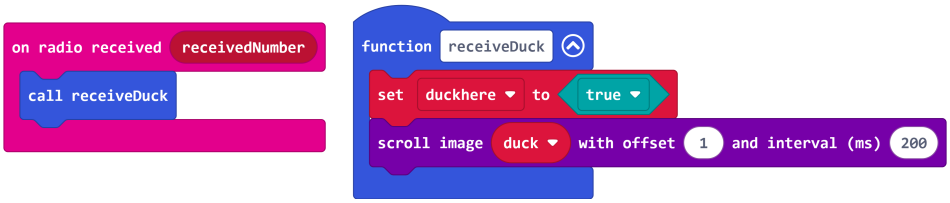
Functions are widely used in computer programs for two main reasons:

- to avoid repetition (as above)
- to make the program easier to write and understand by putting each distinct activity into a function

A function in MakeCode is essentially a block that you design yourself.

Defining and calling a function

Here is part of a teleporting duck program.



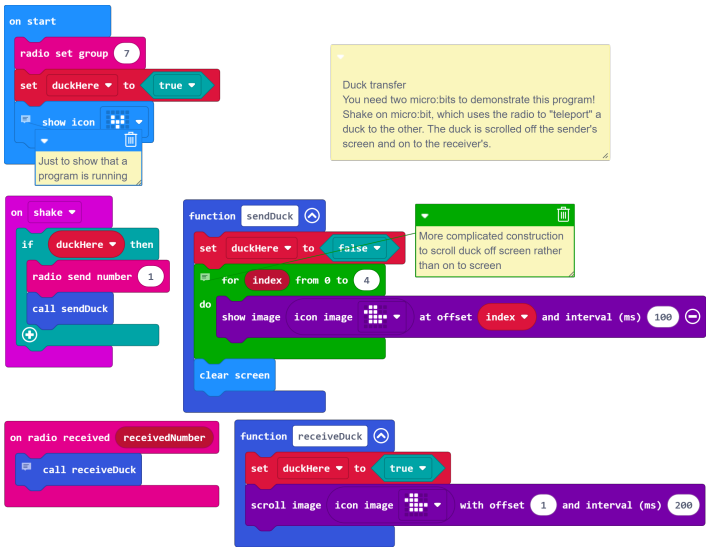
The handler for on radio received calls the function receiveDuck, which does two things.

1. The variable duckhere is set to true, to show that this micro:bit now has a duck.
2. A duck is scrolled on to the display.

The function is barely worthwhile here but it illustrates the usage.

No parameters are passed to the function in this example. It is also possible for functions to return values to the block that called them but I have not done this either.

A function is more useful for sendDuck because more complicated code is required to scroll the duck off the screen. The full program is complicated and introduces several features that I have not yet explained.



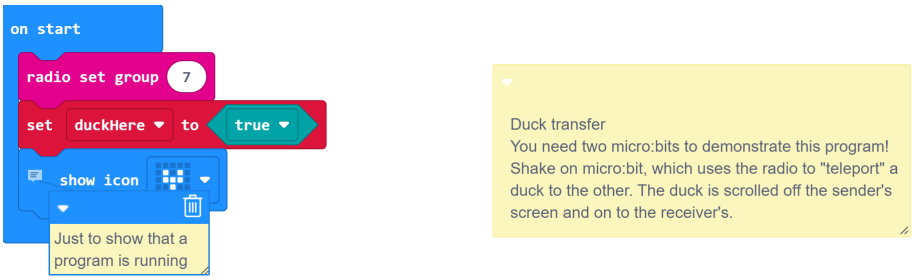
- The variable duckhere is a *boolean* variable, which means that its value can be either true or false. This is convenient for going into if–then statements.
- The on shake handler sends a duck only if we already have one. We have to send some data in the radio signal so I have sent the value 1. This could be the number of ducks if we wanted to be able to send more.
- The function sendDuck sets duckhere to false because we have sent the duck away. It then scrolls a duck off the screen, which requires several blocks because the scroll function is intended to scroll an image on to the screen rather than off.
- The on radio received handler ignores the value that it receives and just calls the function receiveDuck. This is the opposite of sendDuck: it sets duckhere to true, to show that we now have the duck, and scrolls an image on to the screen, which the scroll image block does automatically.

Comments

A well written program should include **comments** to explain features that are not obvious from the code itself. They are intended for humans that study the program and are ignored by the computer. To quote Commenting on your code,

> Adding comments to your code helps explain to other people reading your code what you have written in code and why you have written your code in a certain way.

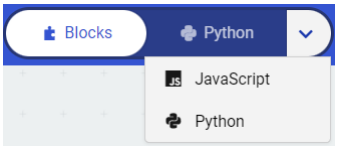
Add comments to MakeCode by right-clicking either the background or a block.



4 Text-based programming languages

Text-based programming languages

If you have programmed in the past, you probably used a text-based language such as FORTRAN, BASIC, Pascal or C. The blocks in MakeCode are wonderful for constructing simple programs but may become frustrating if you want to write something more ambitious. Many other options are available and here are a few.



MakeCode has a selector at the top of the screen that allows you to switch to one of two text-based languages:

- **JavaScript** – the language used to support interactive web pages
- **Python** – a version of a widely used programming language

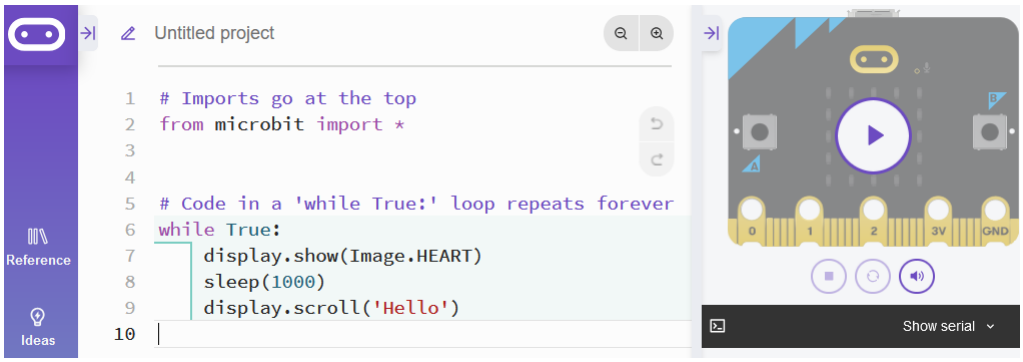
I recommend Python *but not this version.*

Both of these languages are designed to map on to the blocks used in MakeCode so that the user can switch seamlessly between them. Unfortunately this feature constrains the way in which the languages are used, which is why I do not recommend them.

MicroPython

Python is a widely used programming language for desktop computers while **MicroPython** is a version intended for microcontrollers like that on the micro:bit.

You can program in MicroPython using a web-based editor, which includes a simulator, but *this is completely separate from MakeCode.*



Ignore this section if the screenshot of the editor brings back nightmares! MicroPython offers more control than MakeCode and the functions that are equivalent to many blocks provide access to parameters that are not available in the blocks. For example, the brightness of the LEDs can be controlled rather than simply turning them on or off. The user guide is a good introduction with links to the reference manual.

An obvious question is why the micro:bit has two versions of Python. This is answered in the article MakeCode Python and MicroPython. To summarise, the fundamental issue is that Python is a general purpose language and does not include instructions to access the features of the micro:bit. These are added through specific software called an *application programming interface* (API) and the two versions of Python have implemented the API in different ways. The MakeCode version is designed to map on to blocks while that in MicroPython is more conventional, which is why I recommend it.

Many other languages are available but I have tried only these.

5 Behind the scenes

What lies between your code and the processor?

The processor in the nRF52833 is an ARM Cortex-M4F. This includes many advanced features but its basic instructions fall into three categories. They:

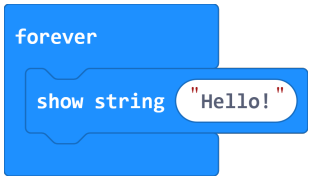
1. **move data** between the registers (fast memories) within the CPU and the main memory. This is like the RCL and STO keys on a pocket calculator.

2. **perform simple calculations** on data in the registers. Some operations are familiar, such as addition, while others are related to Boolean algebra (AND, OR etc).

3. **control the flow** of the program. These instructions support loops, if–then–else statements and so on.

How do our programs relate to these basic instructions?

Hello! program



The forever block could translate directly into a basic instruction but what about show string? This block must:

1. convert the string Hello! into a pattern of dots to display on the LEDs, which requires a table to show the pattern for each character

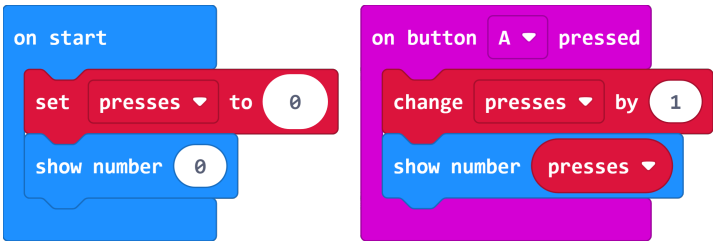
2. put the first character on the 5×5 display

3. wait 150 ms then shift the pattern by one dot so that the text scrolls

4. check whether it has reached the end of the string, so the block has finished

A lot of software lies between show string and the CPU!

Event-driven program with button



Software in the micro:bit checks the state of each button regularly (every 6 ms) and calls your **handler**, the code within on button A pressed, when an event is detected.

The **scheduler** is a vital piece of software that runs on the micro:bit 'behind the scenes' to ensure that all **tasks** receive their fair share of the CPU.

Let's see how it manages the button and the display in this simple program.

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

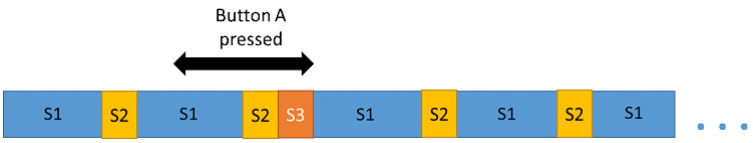
- For button A or B: This handler works when the button is pushed down and released within 1 second.

- 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.

The scheduler in action

This example is taken from The micro:bit – a reactive system. The diagram shows the tasks being run as a function of time across the screen.



S1 task that scrolls the number of presses across the display

S2 task that checks the state of button A, called every 6 ms

S3 handler for button A, called when the button is pressed

The scheduler is a central part of any operating system.

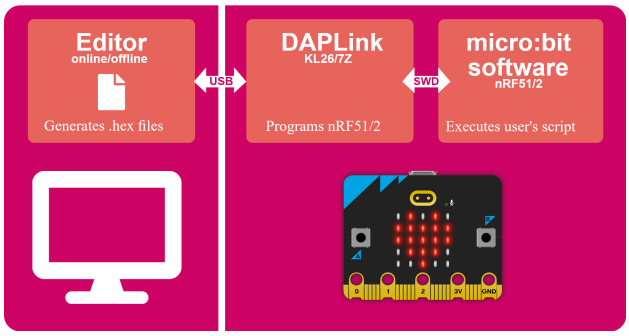
Even a 'simple' system like the micro:bit is supported by a large library of software, without which it would not be possible to construct programs in the simple way that we have done.

This example is inaccurate because the scrolling function S1 runs every 150 ms, which is less often than the polling function S2, every 6 ms. It is possible to react to inputs in a completely different way, using a feature of the computer called an *interrupt*. This is used by the scheduler itself so that it can 'take over' the processor to run tasks at specific times.

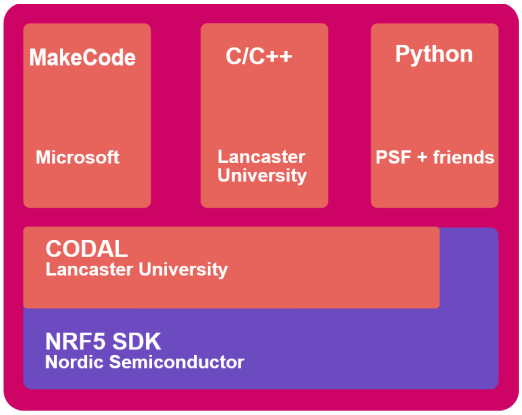
Another example of the scheduler in action is provided by the forever block, which runs its contents every 20 ms. It doesn't simply restart as soon as it gets to the end of its enclosed blocks because that would leave no time for other tasks to run, such as those that check the state of the buttons.

The ARM Cortex-M4F is capable of running a type of operating system called a *real time operating system* or RTOS. As well as a scheduler, this typically passes events to tasks, allows communication between tasks, manages memory and files, and allocates shared resources, such as communication interfaces. The term *real time* means that the system must respond to requests within a specified time so that urgent events are handled before any danger might arise. A small RTOS needs only a few kilobytes of code, a far cry from the operating system on a desktop computer.

The scheduler is only part of the software needed to run your programs on the micro:bit. Several pages on the micro:bit developer community web site show the relation between the various components. This diagram from The micro:bit software ecosystem shows how a program gets from your editor to the micro:bit.



Once on the micro:bit, your program is supported by software called the runtime or *Component Oriented Device Abstraction Layer* (CODAL).



All these layers of software serve two main purposes.

- They greatly simplify the user's programs and permit a clearer structure.
- They hide the details of the hardware from the programmer. This means that the same program could run on different versions of the micro:bit even if the processor were changed. (This has already happened with v1 and v2.)

That's enough!