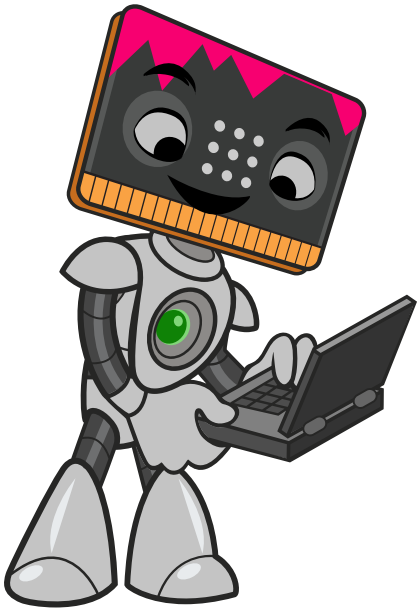
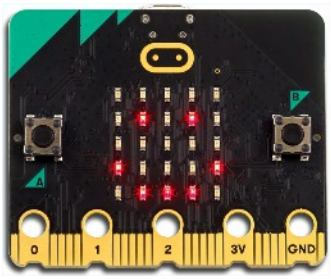
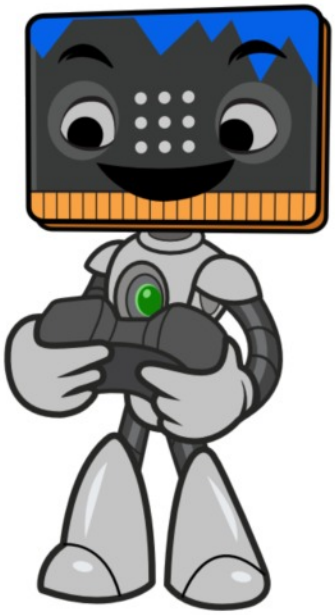
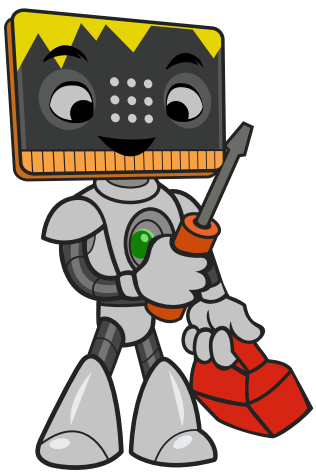
Programming the BBC micro:bit in Plain English

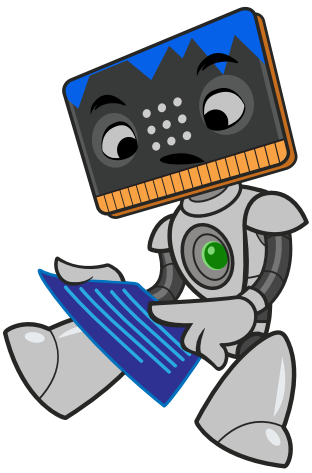






The Mr Bit Approach Explained





Programming with Mr Bit Coding Editor

Introduction

Insight Mr Bit is the coding software for creating programs for the BBC micro:bit, specially designed for young people new to computer programming. These notes will help you understand some basic ideas about computer programs, and will explain how the *Mr Bit* Plain English method solves many problems that you face when you create a program.

What is a computer?

We tend to think of computers as devices with a screen and 'qwerty' keyboard like a *desktop* or a *laptop* computer. There is also, the *tablet* computer where the screen and keyboard are combined in one unit. In fact, a much greater variety of computers are actually hidden in devices such as washing machines, mobile phones, cash machines and photocopiers etc. For all these machines, the common principle by which they work, as with all computers, may be summarised by this simple framework:

Input – Process – Output

input

- *Input*: For a desktop or laptop the keyboard is obvious as the input device, but in general an input could be a humble switch or button or a sensor for measuring, say, light or temperature. Anything that provides an electrical signal can be a potential *input*.

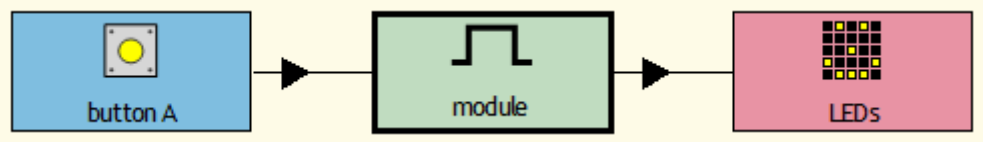
output

- *Output*: The conventional flat screen is only one of many devices that may be controlled by a computer. Virtually any equipment that works when an electric voltage is applied can be used as a computer *output*; lights, motors, LEDs, buzzers, loudspeakers etc. the list is endless.

process

- *Process*: This is at the heart of the special function of a computer. At its grandest, it is the application of 'artificial intelligence'; at its simplest, it makes on/off decisions that control devices. The nature of the process depends on a 'program' - a set of instructions designed to perform a useful task. An instruction usually begins with a *decision* about input conditions followed by an *action* giving or controlling an output. A sequence of instructions performed in a certain order is often referred to as an *algorithm*. When this is coded in a form that a computer can understand, it becomes a *program*.

Mr Bit uses the *Input-Process-Output* model as the framework for building control systems shown graphically as a set of connected boxes. You create a program which is contained in the green control module which is where the processing gets done. (Inputs appear blue and outputs appear pink.)



The 'coding' of a Mr Bit program consists of instructions in plain English sentences, as will be explained below. One of the advantages of this method is that the program script appears almost identical to the instructions in an algorithm.

The BBC micro:bit

The micro:bit is a computer, albeit a tiny one! As a piece of hardware, its circuit board contains all the elements you expect to find in a computer:

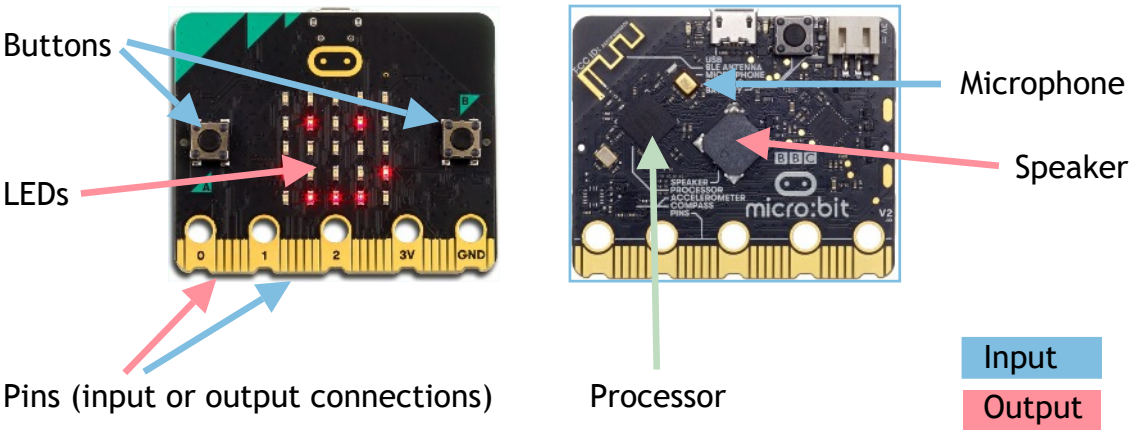
Memory for storing program instructions and data

Processor for performing program instructions

Inputs for supplying information signals (buttons, sensors, pins)

Outputs for observing the resulting action of a program instruction (LEDs, pins)

The micro:bit is able to take in information from inputs, 'process' the information and control outputs accordingly.



The ability of the micro:bit to perform tasks depends upon the *software* code stored in its memory. It is called software because it is not fixed but may be altered according to what the task is. However, a section of the software consists of a standard set of commands and operations called *firmware*, because it stays in memory longer than the code for a particular task. That remaining software contains the set of instructions known as the *program*.

The processor, often referred to as a *microprocessor*, on account of its tiny size, is where program instructions are performed. Such instructions involve detecting information from inputs, measuring signals, evaluating them, making decisions, sometimes performing calculations, and then sending information and signals to outputs. *Mr Bit* makes these instructions as understandable as possible by expressing them in plain English sentences. This method is tailor-made for control applications where the main activity is switching output devices on and off in response to the behaviour of input sensors.

Mr Bit Plain English coding for controlling the micro:bit

The sentences for Mr Bit program instructions have a common format. Essentially they describe how an output device is to be switched on and off. The sentence usually has a three-part structure beginning with a test of inputs to define switching ON ("When....") and ending with a second test to define when the output is to be switched OFF ("...until...").

For example:

When the temperature is above 25, switch on the fan until the temperature is below 25.

Some instructions specify a process rather than an on/off switching action;

e.g. counting or calculating or timing, so a more general expression of an instruction is:

"WHEN a certain input condition is fulfilled, DO something, UNTIL another condition is fulfilled."

Sometimes the WHEN condition may be omitted so that the action begins straight away.

For example:

Switch on the red LED until the button is pressed.

The Mr Bit Plain English language approach was developed especially to achieve efficiency with control applications which typically involve pressing buttons, detecting sensors and switching on LEDs and other devices. Since the micro:bit is designed to perform all these actions with ease, the Mr Bit language perfectly matches the programming possibilities of the micro:bit. However, the method also supports wider programming application, particularly in the management and manipulation of variables: every switch, button or sensor is automatically defined as a *variable*, and a variety of operations with variables are possible; for example, logic, comparison, counting and calculation.

variable

Similarly every output device may be identified as a variable, and as such, its state can also be used as an input signal to a control module.

algorithm

program script

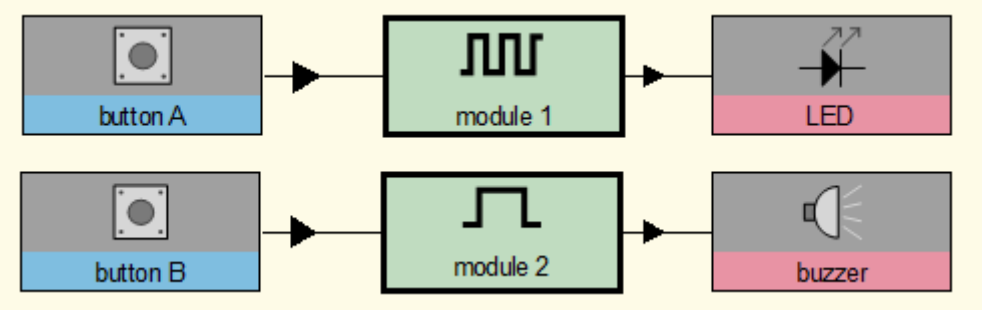
Many programming tasks require a series of instructions in a sequence or in parallel. Before you start coding you need to break down a problem into single steps which are simple enough to be executed by a computer. A set of instructions for performing these steps is often called an *algorithm*. Mr Bit program instructions, or *program script*, attempt to be as similar as possible to the algorithm. In this way you can avoid getting confused with the special words and format used in conventional computer languages; instead you can devote your thinking to getting the algorithm to work.

A context for the Mr Bit approach to coding

There are many ways of introducing young learners to the principles and concepts of computer programs. A common starting point is simple problem-solving activities with *Bee-Bots* and turtle graphics or scripting animated graphics using *Scratch*. Such activities require logical thinking, breaking a problem down into simple steps and then applying them to devise a solution.

The *BBC micro:bit* takes this thinking to the next level, opening the possibility of sensing the environment and controlling physical devices. The coding process demands similar problem-solving skills, but now with a simple computer language format. This is where the plain English approach of *Mr Bit* forms a valuable bridge to more formal approaches. By imitating the instruction sequences expressed in text format in algorithms, it becomes a small step to realise textual code that can be interpreted by the micro:bit's processor, whilst retaining meaningful action.

Clearly, Mr Bit program scripts appear quite different from more conventional coding systems, but equivalence in function is easily found. Take the following example in which one micro:bit button controls an LED and the other controls a buzzer:

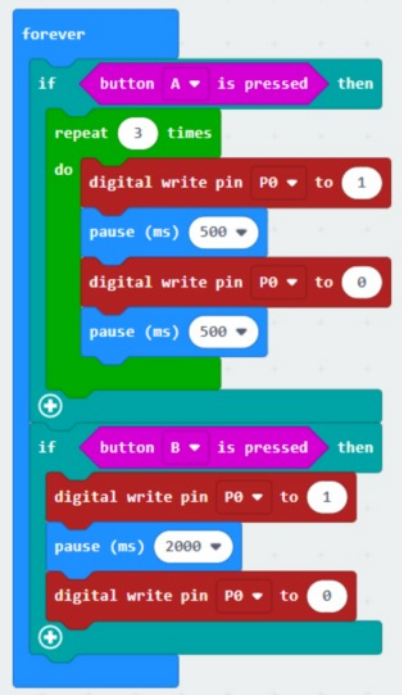


The instructions for the two program modules are thus:

Module 1: When button A gets pressed, flash the LED 3 times.

Module 2: When button B gets pressed, switch on the buzzer for 2 seconds.

This is how the same results are achieved in *MakeCode* and *Micropython*:




```
from microbit import *

# pin0: LED
# pin1: buzzer

while True:
    if button_a.is_pressed():
        count = 0
        while count<4:
            pin0.write_digital(1)
            sleep(500)
            pin0.write_digital(0)
            sleep(500)
            count = count +1

    elif button_b.is_pressed():
        pin1.write_digital(1)
        sleep(2000)
        pin1.write_digital(0)
```

Points to note:

'forever'

Insight Mr Bit (IMB) program instructions repeat 'forever' by default. *MakeCode* (MC) shows this as 'forever', *Micropython* (MP) shows 'while true'.

input conditions

A program instruction with WHEN and UNTIL conditions behaves in a similar way to two IF statements which seek to test the state of the button inputs.

'repeat' loops

IMB has a dedicated pulse control which simplifies the specification for the flashing LED. MC has a 'repeat..do' block to do this. MP defines the variable 'count' which is incremented each time the 'while count<4' loop repeats.

output pins

IMB automatically allocates the output pins for the LED and buzzer when they are linked to the program module. MC and MP switch the LED and buzzer on and off using a 'digital write' command to output pins P0 and P1 respectively.

pauses

MC uses the 'pause' block and MP uses the 'sleep' command to specify the on and off times for the outputs.

Summary of advantages

- In the example, IMB controls the outputs efficiently with just two sentences for its program instructions, compared with 12 blocks with MC and 16 lines of code with MP. And of course, the instructions are immediately understandable.
- Every input sensor and output device is automatically allocated a variable which can store its value or information.
- In addition to simple on/off controls, the range of IMB program modules includes subroutines for time measurements, counting events and calculations.
- The IMB coding editor offers a drag-and-link graphical *Design* mode which automatically builds instruction sentences from the user's choices of inputs, outputs and signal conditions.
- The IMB *Run* mode simulates the program on the computer screen enabling immediate testing of the design.
- The IMB *Check* mode provides physical reporting of the values given by the on-board sensors and testing of the connections to the input/output pins.
- The IMB *Control* mode shows on the computer screen all the activity of the program while it is running inside the micro:bit; ideal for de-bugging the program.
- The IMB music editor makes it easy to create tunes from music scores uniquely demonstrating how music stave notation is converted into scientific code representation.
- The Insight-MrBit.com website contains a wealth of additional ideas, demonstration videos and free downloads.

Programming Basics

Variables

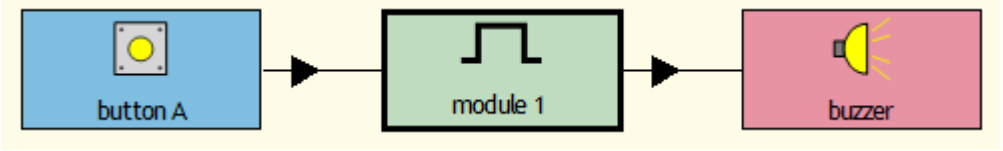
digital

analogue

In testing the state or value of inputs, every input is automatically assigned a variable bearing the name of the input. (Think of a variable as a *store* of information that sits in the computer memory. Its value will change as a program runs, but it can always be accessed by its name.) If the sensor is a type of switch or button, the variable takes a *digital* value of 1 or 0 (equivalent to on/off or true/false). For *analogue* sensors, such as those for temperature or light level, the variable is a number representing the value of the property being measured.

Example of *digital* variables:

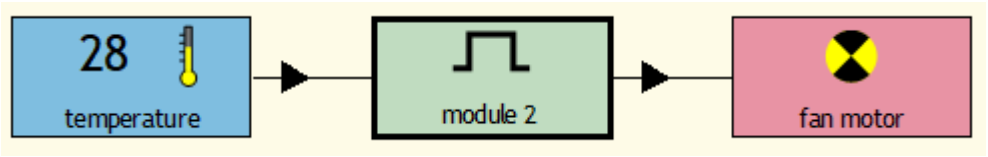
When button A is pressed, sound the buzzer alarm until button A is released.



The variable *button A* stores the on/off signal from that button. The program causes the variable *buzzer alarm* to switch between a high and low state, activating the buzzer device accordingly.

Example of *analogue* variables:

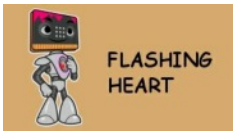
When it is warmer than 25, switch on the fan motor at 60% speed until it is cooler than 20.



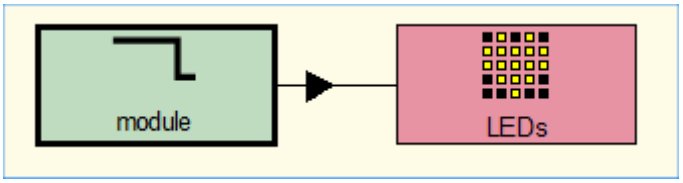
The variable *temperature* is tested for its magnitude which controls the state of the motor. In this case the variable *speed* is fixed at a value of 60% of maximum.

time

Time, as a variable, is frequently used as a WHEN or UNTIL condition. In the following example, the UNTIL condition for the first instruction is a time of 0.5 seconds; the UNTIL condition for the second instruction (pause) is also a time. The result shows a flashing heart image on the LED display.



Show the LED image (heart) for 0.5 seconds.
Wait for 0.4 seconds.



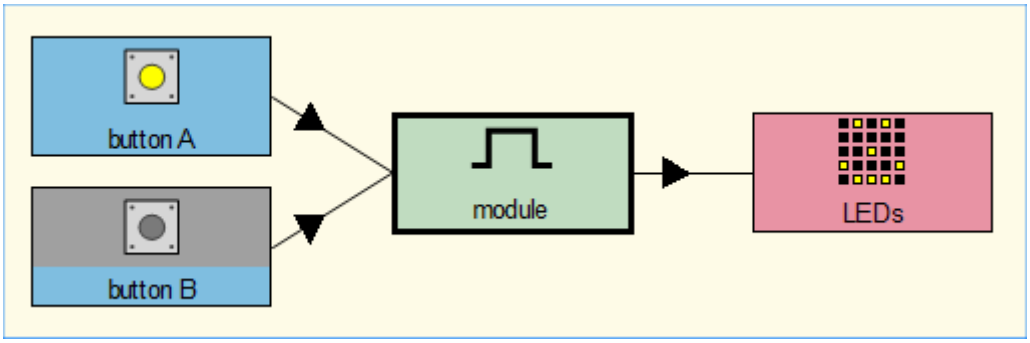
There are other types of variable and many uses which are discussed later.

Loops

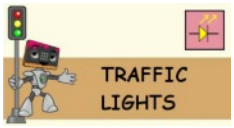
A single program instruction is automatically repeated. (This is a default property of a Mr Bit program instruction. It has occasional limitations, but the vast majority of control systems require a 'forever' duty cycle.)

In this example, as soon as button B is pressed, the instruction waits for button A to be pressed again:

When button A is pressed, show the LED image (smiley), until button B is pressed.

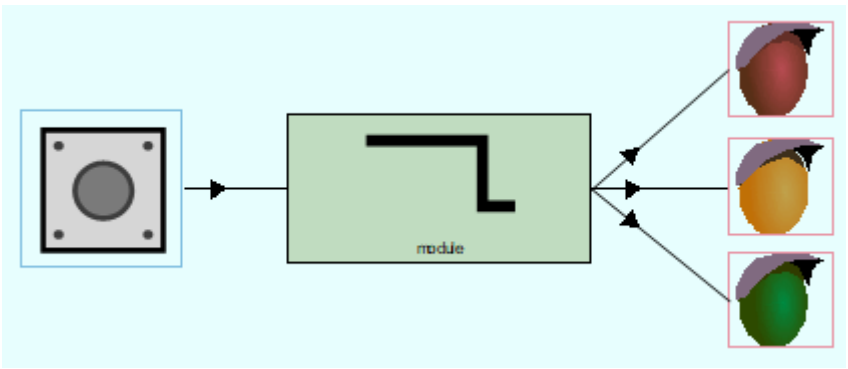


For a series of program instructions in a sequence, the same loop rules applies; as soon as the last instruction is completed, control returns to the first instruction.



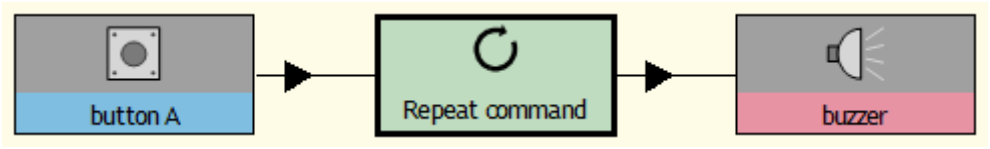
In this example for controlling traffic lights, the Green LED is switched on after the red and yellow LEDs have been on for 2 seconds:

Switch on the Green LED until button A gets pressed.
Switch on the Yellow LED for 2 seconds.
Switch on the Red LED for 6 seconds.
Switch on the Red LED and Yellow LED for 2 seconds.



When a limited number of repeats is required, or repetition until a certain condition is fulfilled, the **Repeat** instruction may be used, as in this example:

When button A is pressed, repeat until button A is free:
Pulse the buzzer twice.
Wait for 1 second.



Testing digital inputs

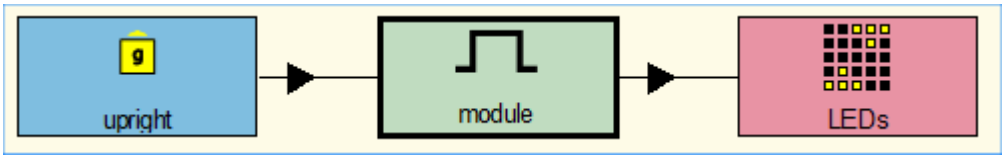
WHEN-UNTIL

IF-THEN-ELSE

The WHEN and UNTIL phrases define conditions which have to be fulfilled for switching the output on and off respectively. In other languages these tests are achieved with IF – THEN – ELSE statements. The distinctive feature of the WHEN – UNTIL structure in Mr Bit is that it combines a switching-on test with a switching-off test in a single instruction. WHEN behaves like IF, but additionally waits for the IF condition to be fulfilled before proceeding to the action. Similarly, UNTIL behaves like IF, but also waits for the IF condition to be fulfilled.

In this example, the LED animation is switched on while the micro:bit is upright. The Mr Bit instruction states the conditions for the action:

When the micro:bit is upright, show the LED animation (snowflakes) until the micro:bit is not upright.



(This example uses the position of the micro:bit as an input signal which comes from the built-in accelerometer sensor.)

state condition

true or false

A WHEN or UNTIL condition can test the *state* of an input signal or a *change* in the input signal (an *event*). The above example shows each condition testing the *state* of the micro:bit. The result may be described as being *true* or *false:* If the micro:bit is in an upright state, the WHEN condition is *true* but the UNTIL condition is *false*; if the micro:bit is not upright, the WHEN condition is *false* and the UNTIL condition is *true*.

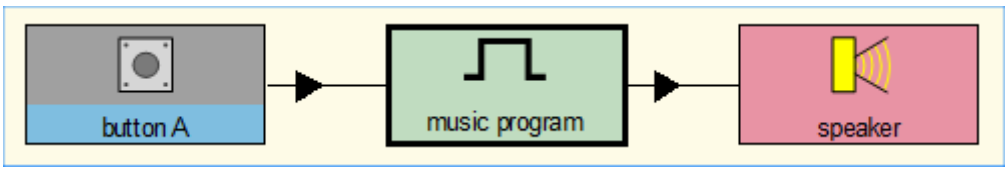
When, as in this example, the UNTIL condition is the opposite of the WHEN condition, the program instruction may be further simplified to:

While the micro:bit is upright, show the LED animation (snowflakes).

Thus in an instruction beginning with WHILE, the action continues until the WHEN condition becomes false.

event condition

In contrast to conditions that test the *state* of an input, an *event* condition tests for a *change* in an input; it waits for a change to happen. For a button the event may be the action of being pressed or released, as in the next example:

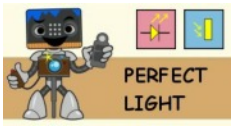


When button A gets pressed, play "MUSIC" until button A gets pressed again.

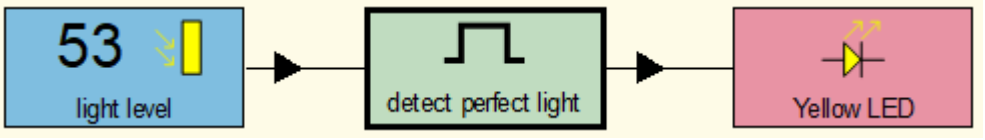
Here, both WHEN and UNTIL test for the *event* of button A being pressed. Note that the instruction language for testing an event is different from testing the state of the button: "gets pressed" instead of "is pressed".

Testing Analogue Inputs

When an analogue sensor is used as an input, WHEN and UNTIL conditions are defined by comparing the sensor value with a specified number. The evaluation takes the form ‘is greater than..’, ‘is equal to...’, ‘is less than...’ etc. as in this example:



When it is lighter than 50, switch on the Yellow LED until it is darker than 50.



As we have seen previously, the WHEN and UNTIL conditions can be simplified to:

While it is lighter than 50, switch on the Yellow LED.

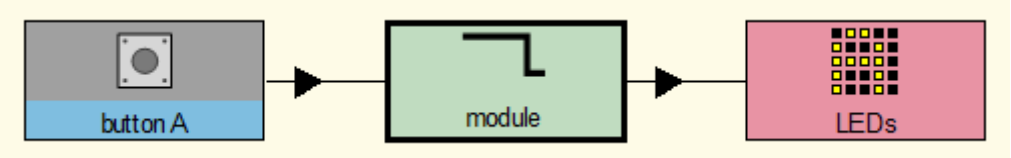
Using AND or OR logic an analogue signal can be tested to detect when it falls within or outside a specified range, as in this example:

While it is lighter than 50 and darker than 60, switch on the yellow LED.

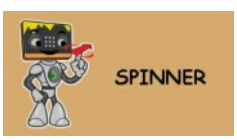
Sequence of instructions

When a series of instructions is created, they are performed in sequence order. (In Mr Bit, all the instructions are together in one control module.)

Show the LED message "TO SLEEP, PRESS A" until button A gets pressed.
Show the LED message "GOODNIGHT" for 5 seconds.
Show the LED message "SLEEP WELL" at 5/10 brightness for 6 seconds.
Show the LED message "....ZZZZ" at 1/10 brightness for 6 seconds.
Wait until button A gets pressed.



Other examples:

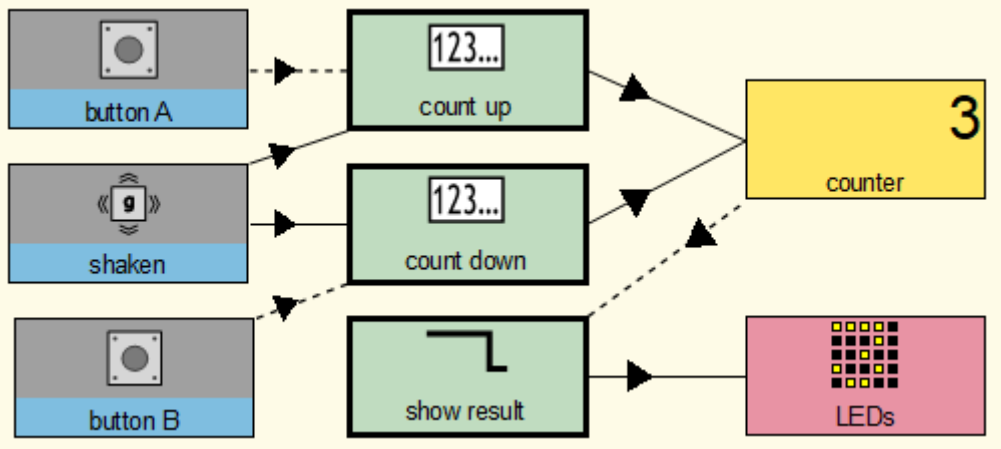


Parallel instructions

When you create more than one control module, each with their own program instruction, they behave independently of each other.

In this example, both buttons control counting independently of each other, and the shaking gesture resets counting to zero at any time. There are three control modules each with its own program instruction:

count up: Count how many times button A gets pressed until the micro:bit is shaken.
count down: Count (down) how many times button B gets pressed until the micro:bit is shaken.
show result: Show the LED message (counter) until exit.



Combining inputs

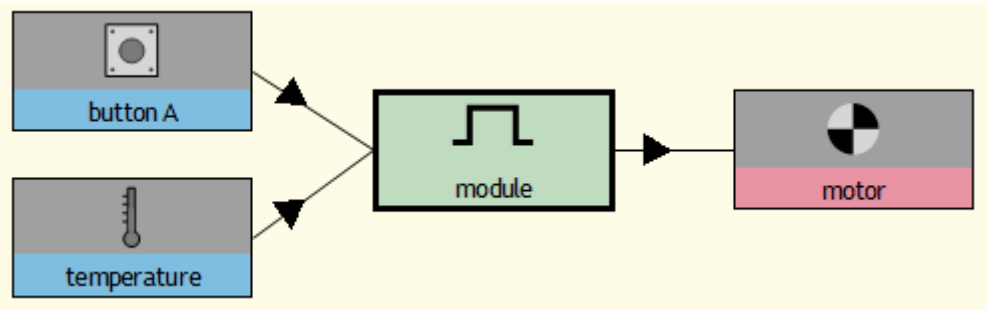
logic

A second input is allowed when defining a WHEN or UNTIL condition. You need to chose whether to use AND logic or OR logic for combining the input conditions.


Greenhouse project

In this example the motor will only switch on when *both* inputs are high (button pressed AND temperature high), but either input can switch it off (button pressed OR temperature low). The program instruction makes it quite clear:

When button A gets pressed and it is hot, switch on the motor until button A gets pressed or it is cold.



Numerical variables without sensors

Three types of module have an automatic numerical variable output:

Timer module - measures the time between WHEN and UNTIL events

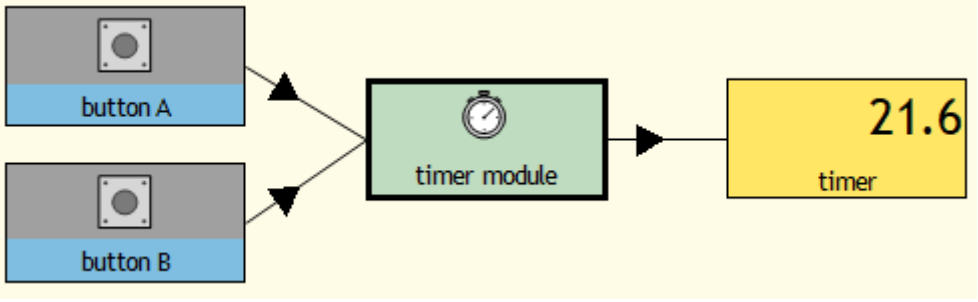
Counter module – counts events when a signal changes

Calculation module – performs a simple calculation upon two or more numerical variables

In each case the output variable value appears in a yellow coded box viewer.

Timing events:

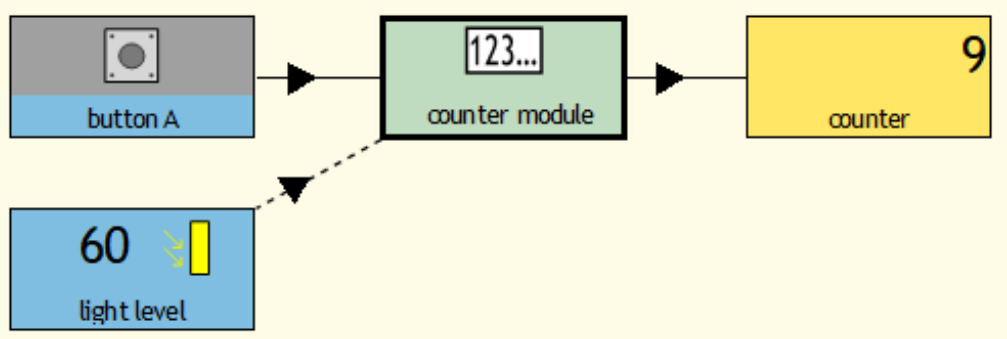
When button A gets pressed, measure the time until button B gets pressed.



Counting events:

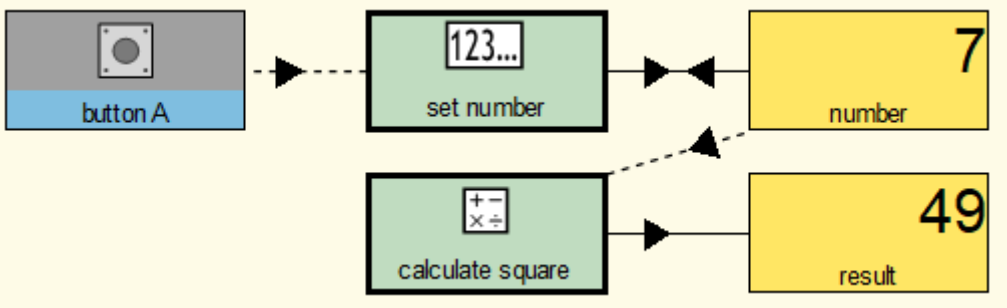
When button A gets pressed, count how many times it gets darker until exit.

In this case, the counting event is detected each time the light level falls below a user-defined threshold.



Calculating a value:

<u>set number</u>: Count how many times button A gets pressed until the number is greater than 12.
<u>calculate square</u>: Calculate result = number x number until exit.



Custom variables

A variable is essentially a store of information. For the variables described so far, they have stored the value of an input signal or the state of an output; *analogue* variables have a numerical value whereas *digital* variables indicate a state as on/off, high/low, or true/false.

The Assign module is used to create and assign values to a variable of your own making. There are three main types:

String variable

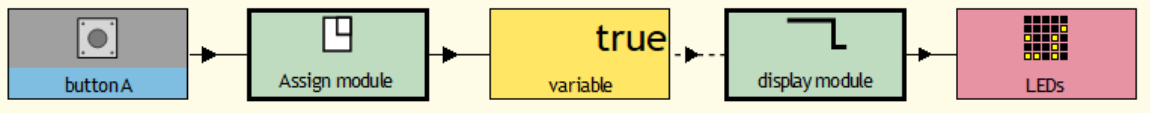
String variable – contains text (a string of letters making words or sentences)

Numerical variable

Numerical variable – contains an analogue number, it can be assigned a 'value' or can be changed by a specified amount. One option assigns a *random number* to the variable.

Boolean variable

Boolean variable – contains a digital state 'true' or 'false'



The variable may be displayed on the micro:bit LEDs using an additional module.

Assigning a string *variable named* "**weather**":

When it is warmer than 30, set weather to "HEAT WAVE".
When it is cooler than 10, set weather to "REALLY CHILLY".

Readings from a temperature sensor are used to define the string (text) stored by the variable "weather".

Assigning a numerical *variable named* "**column**":

Start: Set column to 0.
Move right: When button A gets pressed, add 1 to column.
Move left: When button B gets pressed, subtract 1 from column.

In the Minesweeper game, the aim is to scan the LEDs display to find a hidden mine. The variable "**column**" stores the horizontal position during scanning.

Assigning a random number *to the variable named* "**height**":

New meteor: After 5 seconds, set height to a random number from 1 to 5.

In the Meteorite game, you try to avoid a meteorite flying across the LEDs display. The variable "height" stores its vertical position as a random number.

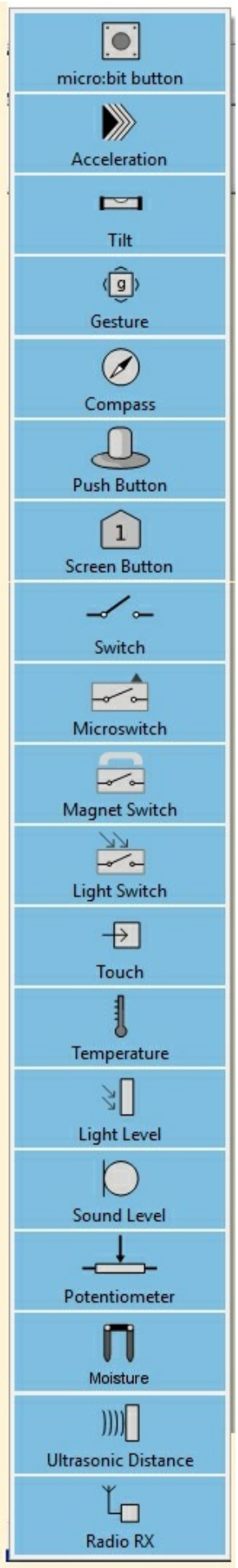
Assigning a Boolean *variable named* "**hit**":

Ball hits wall: When the x-ball is equal to 5, set hit to false.
Ball is hit: When x-bat is equal to x-ball, set hit to true.

In the Ping Pong game, the ball and bat are shown as lit single LEDs in the display. The variable "**hit**" is set to *true* when these coincide.

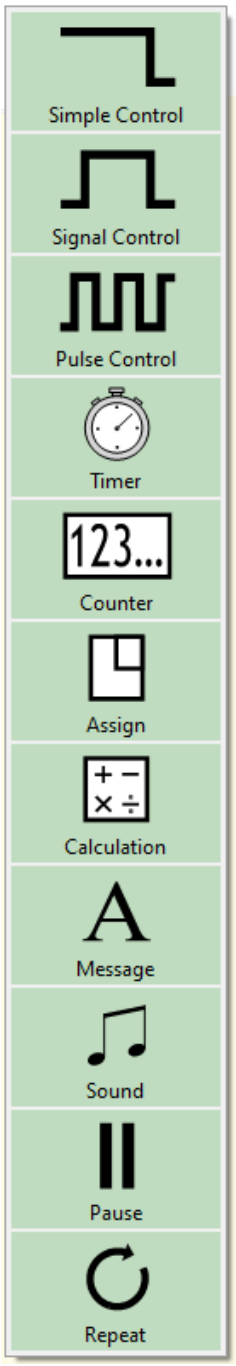
Appendix Overview of sensors, devices and program modules

Input sensors



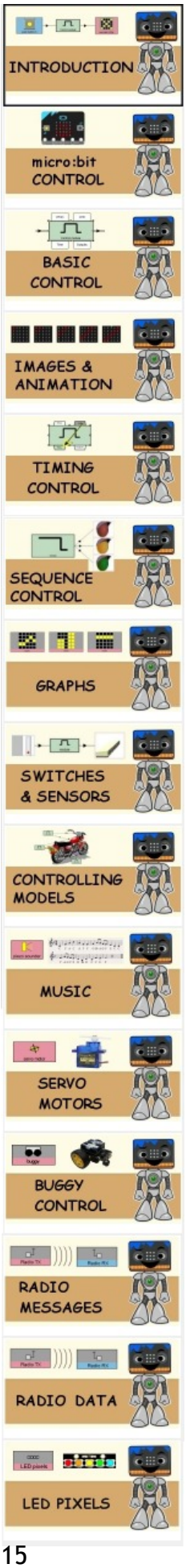
Output devices

Program modules

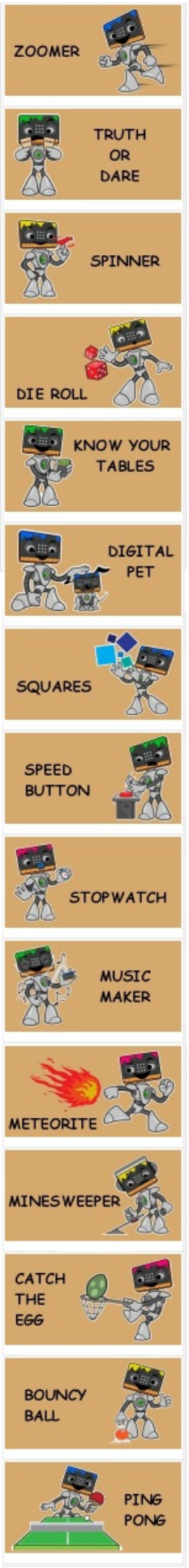


Appendix Insight Mr Bit Resources at a glance

Tutorials



Exercises



Experiments with sensors

Projects

Programming with Mr Bit

Version 4.0, Nov.2020

Author: Laurence Rogers

Published by Insight Resources

www.insight-mrbit.com