Microsoft MakeCode

Embedded Programming for Education, in Blocks and TypeScript

Thomas Ball
Microsoft Research
Redmond, WA, United States
tball@microsoft.com

Abhijith Chatra
Microsoft
Redmond, WA, United States
abchatra@microsoft.com

Peli de Halleux
Microsoft Research
Redmond, WA, United States
jhalleux@microsoft.com

Steve Hodges
Microsoft Research
Cambridge, United Kingdom
sehodges@microsoft.com

Michał Moskal
Microsoft Research
Redmond, WA, United States
mimoskal@microsoft.com

Jacqueline Russell
Microsoft
Redmond, WA, United States
jacqueline.russell@microsoft.com

Abstract

Microsoft MakeCode (https://www.makecode.com) is a platform and accompanying web app for simplifying the programming of microcontroller-based devices in the classroom. For each device, MakeCode provides a customized end-to-end experience in the web browser consisting of code editors, device simulator, debugger, compiler to machine code, and linker to a pre-compiled C++ runtime, as well as a documentation and tutorial system. We present an overview of MakeCode and detail the major design decisions behind the platform.

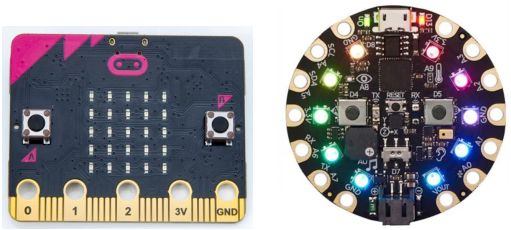
CCS Concepts • **Social and professional topics → Computing education**; **K-12 education**; • **Computer systems organization** → *Embedded software.*

Keywords CS education, microcontrollers, TypeScript, JavaScript, Blockly

ACM Reference Format:
Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michał Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: Embedded Programming for Education, in Blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN SPLASH-E Symposium (SPLASH-E '19), October 25, 2019, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3358711.3361630

SPLASH-E '19, October 25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6989-3/19/10…$15.00
https://doi.org/10.1145/3358711.3361630



(a) (b)

Figure 1. Two Cortex-M0 microcontroller-based educational devices: (a) the BBC micro:bit; (b) Adafruit's Circuit Playground Express

1 Introduction

Microsoft MakeCode (https://www.makecode.com) is a platform and accompanying web app for simplifying the programming of microcontroller-based devices in the classroom. [5] For each device, MakeCode provides a customized end-to-end experience in the web browser consisting of code editors, device simulator, debugger, compiler to machine code, and linker to a pre-compiled C++ runtime, as well as a documentation and tutorial system.

The BBC micro:bit (see Figure 1(a)) is one example of a *physical computing* device that MakeCode supports [1]. The micro:bit has several user-programmable buttons, a 5x5 LED display, various sensors, Bluetooth Low Energy (BLE) radio technology, and an edge connector for expandability. The micro:bit was introduced in the UK in 2015 to all year-7 students (age 10-11) in support of the government's mandate to provide CS education for all grades. Since the fall of 2016, the micro:bit is a global effort spearheaded by the micro:bit Education Foundation (https://microbit.org), with several million units distributed worldwide to date. Figure 1(b) shows another device supported by MakeCode: Adafruit's Circuit Playground Express (CPX).

Figure 2 shows the MakeCode web app for the micro:bit (available at https://makecode.microbit.org), which consists of five main sections: (A) menu bar with toggle between

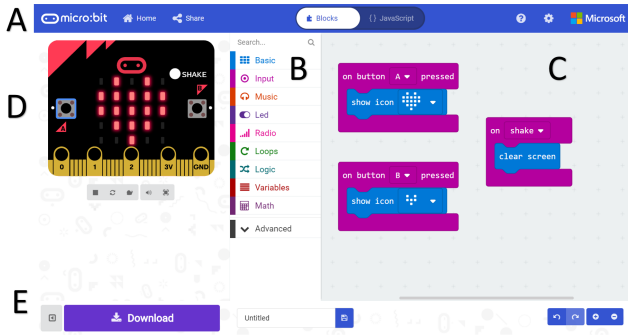


Figure 2. MakeCode web app for the BBC micro:bit

Block and TypeScript editors; (B) a toolbox of API categories; (C) a programming editor (Blockly); (D) a micro:bit simulator that allows testing of the user program without a device; (E) a download button that invokes the compiler to produce a machine-code executable. The simple program shown in the editor consists of three event handlers that display different icons on the LED screen when buttons A or B are pressed and clears the screen when the accelerometer on the micro:bit detects shaking.

MakeCode is implemented in TypeScript (https://www.typescriptlang.org) and provides a subset of TypeScript (referred to as Static TypeScript or STS, for short [2]) for end-user programming, and an even smaller language subset available via Blockly. Automated translation between Blockly and STS allows the user to switch editing paradigms (though not every STS construct, such as classes, is editable in Blockly). STS is compiled into an intermediate representation (IR) supported by two backends: the first converts the IR into a continuation-passing form of JavaScript that allows simulation of non-preemptive multi-threading in a single-threaded JavaScript runtime; the second compiles the IR to ARM assembler and then to machine code. There are two ways to install the compiled program onto the micro:bit: first, the executable can be copied to the micro:bit when it is plugged into the computer over USB (where it masquerades as a file system); second, the executable can be flashed over the air via Bluetooth.

Recently, we have added MakeCode Arcade for programming retro-styled video games (https://arcade.makecode.com). For the first time, instead of creating an editor for existing hardware, we have published a hardware specification and seen a number of companies building devices compatible with Arcade (see Figure 3). Arcade allows users to express themselves at a variety of levels, from customizing sprites using the built-in pixel editor, all to way to building custom AI for controlling opponents in a game. Arcade is made possible by the high performance of the generated machine code, resulting in smooth game play on such constrained devices.

Figure 3. Various devices compatible with MakeCode Arcade.

The entire MakeCode platform is open source under MIT license.[1] More information about MakeCode can be found at https://makecode.com/blog.

2 Design Choices

This section describes some of the major design choices behind MakeCode.

2.1 Web App and UF2

Any requirement to install applications or device drivers on school computers is a significant barrier to adoption as most such computers are locked down. Furthermore, early trials in UK schools with the BBC micro:bit showed that relying on an internet service for compilation led to unacceptable failures due to spotty network connectivity. Finally, the diversity of school computers, which now includes a substantial number of Chromebooks means that the only software in common across platforms is the modern web browser. For all these reasons, we chose to implement the entire MakeCode experience as web app. Once the web app has loaded into browser, it remains resident and operational (even if the browser is closed and reopened, or network connectivity is lost).

This leaves the question of how to get the binary executable file from the computer to the device. For this, we designed a new file format called UF2 (https://github.com/Microsoft/uf2) for flashing microcontrollers. UF2 leverages the USB MSC (mass storage class) used by removable USB

[1] See https://github.com/microsoft/pxt

(a)
```
//% block="show|string %text"
//% async
//% blockId=device_print_message
//% parts="ledmatrix"
//% text.shadowOptions.toString=true interval.defl=150
function showString(text: string, interval?: int32): void;
```

(b)
```
input.onGesture(Gesture.Shake, function () {
    basic.showString("Hello!",200)
})
```

(c)

Figure 4. (a) Device APIs (in this case, `showString`) are defined in TypeScript and comment metadata defines the mapping to Blockly; (b) example Static TypeScript user code with value for optional second parameter to `showString` (c) example Blockly user code (second parameter not available).

flash drives, which is supported by all modern operating systems.[2] This lowest-common denominator approach has been successful but requires the user to perform a copy operation after each compilation; with the advent of WebUSB, MakeCode can perform the copy operation automatically once the user has permitted the browser to access USB. Unfortunately, not all web browsers support WebUSB yet.

2.2 Static TypeScript and Blockly

The choice of TypeScript was motivated by our experience with using static types to provide better intellisense and code hints in the TouchDevelop project [3]. All MakeCode device APIs are strongly typed, which means that users can for the most part code without types (essentially in JavaScript; the editor is even labelled JavaScript for better recognition among students and teachers). TypeScript's type inference engine determines the type of variables. As shown in Figure 4(a), MakeCode uses metadata in the comments associated with a device API (`showString` in this case) to make that API available via Blockly (see Figure 4(c)): the types of the API are used to automatically determine the kind of block that will be generated. The return type for function `showString` is `void`, which means that a Blockly "statement" block is used rather than an "expression" block.

Going further, Static TypeScript (STS) eliminates most of the "bad parts" of JavaScript; following StrongScript [6], STS uses nominal typing for statically declared classes and

[2] UF2 is used on CPX and Arcade boards. BBC micro:bit uses ARM DAPLink software which also acts as MSC device, but uses a file format not optimized for this purpose and requires a separate interface chip.

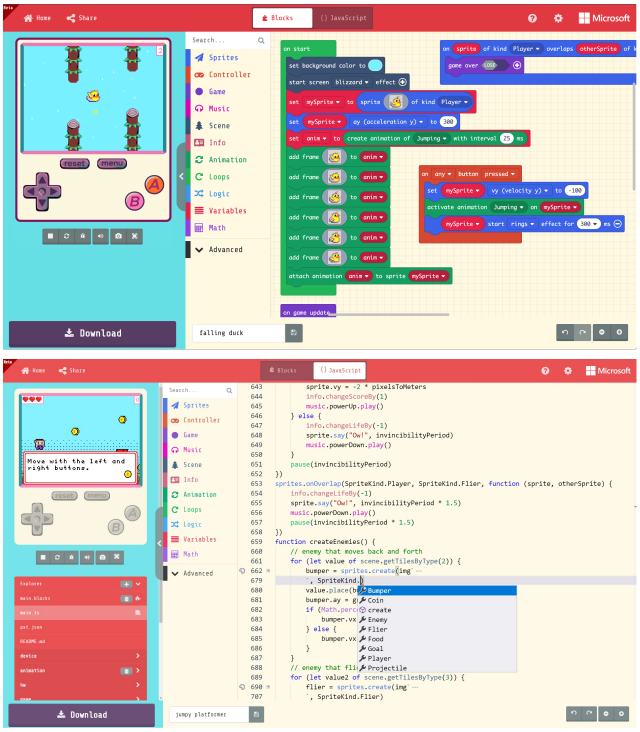


Figure 5. Sample Arcade games in Blocks and Static TypeScript.

supports efficient compilation of classes using classic techniques for v-tables. Thus, STS can be used to teach classic object-oriented programming.

2.3 APIs

The design of APIs is a critical part of the MakeCode experience. APIs are segmented into categories (using TypeScript namespaces), which are reflected into the toolbox categories shown in the IDE (see Figure 2(B)). APIs are layered as follows: high-level event handlers allow a program to respond to a device event (such as shaking, which is enabled by the accelerometer sensor); polling APIs allow access to sensor values (the X, Y, Z accelerometer values); while low-level APIs such as I2C provide direct access to hardware components.

Furthermore, via the API metadata, the block associated with an API can have fewer parameters than available in TypeScript, further simplifying the API. For example, Figure 4 shows that the API `showString` has two parameters, where the second one is optional. The attribute `block` defined by the first comment line defines that only the first parameter (`text`) will appear in the block. The optional parameter is giving a default value in a later comment.

2.4 Concurrency Model

The `showString` function for the micro:bit (shown in Figure 4) scrolls a message across the micro:bit screen, which can take many seconds, depending on the length of the string. During this time, the user may want to react to another event (say a button press) and perform an action.

MakeCode supports this scenario through implicit non-preemptive multi-threading. A function call can pause for a given time, or wait for a specific event, which causes other threads to execute. As in Scratch, only specific functions (namely pause and wait-for-event) can yield to other threads, making it simpler to reason about integrity of data and to avoid races.

When executing in the browser this is achieved with a custom compiler backend, which generates continuation-passing-style JavaScript from the TypeScript source code. Functions are transformed into big switch statements inside of a while loop that facilitates control flow. Local variables are placed in explicitly allocated objects representing stack frames. This allows for suspension of execution of any function at calls to pause and wait-for-event. When the current thread is suspended, other user threads are allowed to run.

When compiled in debug mode, potential suspension points are additionally inserted before each statement to allow for step-through and pausing on breakpoints. At all suspension points, as well as on back-jumps the runtime may pause to allow the browser to process events (in particular, user pressing "stop the simulator" button). Neither of these pauses, however, allows other user threads to run.

All of these are similar to generic mechanisms implemented in Stopify [4].

The C++ runtime has explicit support for events and threads (called fibers), so the native compiler backend does not do anything special in this area.

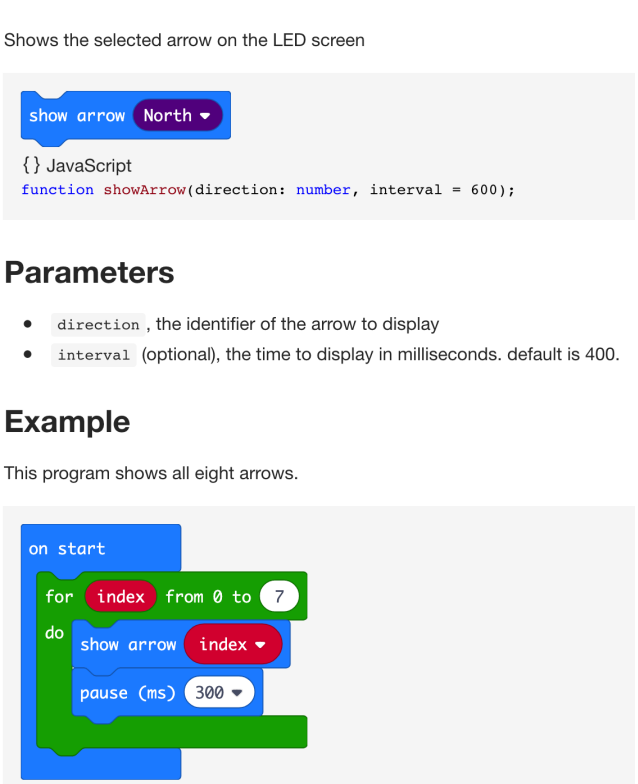
2.5 Documentation

The MakeCode documentation system uses markdown to make it easy to create documents.[3] It is: **sustainable**: all Blockly images are rendered on the fly (from their Static TypeScript textual equivalent) to avoid having to maintain hundreds of screenshots (see Figure 6); **localizable**: the documentation is synchronized with the Crowdin platform and localized on the fly by the MakeCode cloud backend; **reusable**: documentation pages can be shared across different MakeCode editors and specialized per device; **integrated**: step-by-step tutorials or sidebar-hosted documents allow surfacing information within the editor without having to switch to a different browser tab.

2.6 Sharing

MakeCode users can share their programs in a variety of ways. First, the binary executable that MakeCode creates contains the source text of the user program. If the executable is dragged into the MakeCode web app, the app will extract the source text into the editor. Second, MakeCode can store the user's encrypted program in the cloud and generate a URL (that contains the decryption key) to share with other users.

[3] https://makecode.com/writing-docs




```
# Show Arrow
Shows the selected arrow on the LED screen
```sig
basic.showArrow(ArrowNames.North)
```

Parameters
* ``direction``, the identifier of the arrow to display
* ``interval`` (optional), the time to display in
  milliseconds. default is 400.

Example
This program shows all eight arrows.
```blocks
for (let index = 0; index <= 7; index++) {
    basic.showArrow(index)
    basic.pause(300)
}
```
```

Figure 6. A sample documentation page as rendered, and its source.

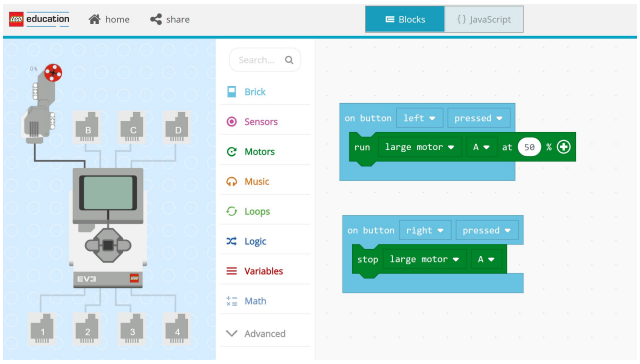


Figure 7. MakeCode editor for LEGO Mindstorms EV3 (https://makecode.mindstorms.com).

2.7 Extensibility

MakeCode supports the concept of an extension, a collection of STS files that also can list other extensions as dependencies. Third parties use extensions to extend the MakeCode editors, mainly to support hardware peripherals.[4] Due to the efficiency of the STS compiled code and the availability of low-level STS APIs for accessing hardware via digital/analog pins (GPIO, PWM and servos) and serial protocols (I2C and SPI), most extensions for hardware peripherals can be implemented solely in STS, without recourse to the use of C++ or assembly.

3 Other MakeCode Editors

In addition to the editors for the micro:bit, CPX and Arcade devices, MakeCode has a number of other editors. We call out two here for their distinctive features.

First, there is an editor for LEGO Mindstorms EV3, a robot construction system that features a programmable "brick" to which can be attached a variety of motors and sensors. A novel aspect of the MakeCode editor for EV3 is that the simulator is automatically configured based on the program the user has written. As shown in Figure 7, the program instructs the EV3 brick to run/stop a large motor (attached to the brick's port A). MakeCode automatically configures the simulator to show a large motor attached to port A. That is, MakeCode ensures that the wiring of motors/sensors to the brick is in sync with the program that the user has written.

Second, there is an editor for Minecraft, a game that allows players to build with blocks in a 3D procedurally generated world.[5] This is our first editor not associated with a physical device. In this case, the user's program communicates to Minecraft via a web socket and sends commands that can query and update the state of the Minecraft virtual world. The editor and Minecraft run in separate windows. Minecraft Education Edition incorporates the code editor inside the Minecraft application itself.

4 Experience with MakeCode

Since its release in 2016, Microsoft MakeCode has been used by educators around the world as an introductory platform for physical computing. MakeCode is most often used at the middle school level, where learners often have already been exposed to block-based programming via tools like Scratch, but have little to no experience with text-based programming.

MakeCode serves as a bridge to take a student from the familiar drag-and-drop coding paradigm into the less familiar territory of typing code. With the ability to transition back into blocks, as well as code snippets, intellisense, and error-detection, MakeCode gives these new learners 'water wings' as they venture into the deep end. More research is required to evaluate if such a multi-mode editor approach is useful. [7]

Below are a few quotes from teachers and parents:

- "You guys have nailed the block to text transition, which is probably one of the most fundamental difficulties in teaching kids to code" – P. G. (Code Club Teacher)
- "The concept of allowing our youth to discover their creativity and passion is crucial to our future as a country and I don't say that lightly. These tools, MakeCode and MakeCode Arcade, in my opinion, can open doors to creativity for our kids." – G. S. (Middle School Teacher)
- "Quick thank you note - My kids woke up at the crack of dawn this morning, and the first thing they wanted to do was pull out the micro:bit and get to work. My kids have had a lot of exposure to block-based programming, MaKey MaKey, microprocessors galore... Yet, this is the most motivated I have seen them with these micro:bits!" – M. H., Mother (Maker Faire attendee)

For more examples of how educators are using MakeCode in the classroom, see the videos at https://www.microsoft.com/en-us/makecode/about.

Acknowledgements

We would like to thank current and former members of the MakeCode product team: Sam El-Husseini, Caitlin Hennessy, Guillaume Jenkins, Shannon Kao, Richard Knoll, and Daryl Zuniga. We also express our gratitude to James Devine and Joe Finney at Lancaster University, the authors of CODAL used as a layer of our C++ runtime.

References

[1] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli de Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale.

[4] For example for micro:bit, see https://makecode.microbit.org/extensions
[5] https://makecode.minecraft.org

2020. The BBC micro:bit – from the UK to the World. *Commun. ACM (to appear)* (2020).

[2] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: Static Compilation of Dynamic Languages for Embedded Systems. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19)*.

[3] Thomas Ball, Jonathan Protzenko, Judith Bishop, Michał Moskal, Jonathan de Halleux, Michael Braun, Steve Hodges, and Clare Riley. 2016. Microsoft Touch Develop and the BBC Micro:Bit. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. 637–640.

[4] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. 2018. Putting in all the stops: execution control for JavaScript. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. 30–45.

[5] James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '18)*. 19–30.

[6] G. Richards, F. Z. Nardelli, and J. Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*. 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

[7] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *TOCE* 18, 1 (2017), 1–25.