MakeCode and CODAL: Intuitive and Efficient Embedded Systems Programming for Education

James Devine
Lancaster University, UK
j.devine@lancaster.ac.uk

Joe Finney
Lancaster University, UK
j.finney@lancaster.ac.uk

Peli de Halleux
Microsoft, USA
jhalleux@microsoft.com

Michał Moskal
Microsoft, USA
michal.moskal@microsoft.com

Thomas Ball
Microsoft, USA
tball@microsoft.com

Steve Hodges
Microsoft, UK
steve.hodges@microsoft.com

Abstract

Across the globe, it is now commonplace for educators to engage in the making (design and development) of embedded systems in the classroom to motivate and excite their students. This new domain brings its own set of unique requirements. Historically, embedded systems development requires knowledge of low-level programming languages, local installation of compilation toolchains, device drivers, and applications. For students and educators, these requirements can introduce insurmountable barriers.

We present the motivation, requirements, implementation, and evaluation of a new programming platform that enables novice users to create software for embedded systems. The platform has two major components: 1) Microsoft MakeCode (www.makecode.com), a web app that encapsulates an entire beginner IDE for microcontrollers; and 2) CODAL, an efficient component-oriented C++ runtime for microcontrollers. We show how MakeCode and CODAL provide an accessible, cross-platform, installation-free programming experience for the BBC micro:bit and other embedded devices.

CCS Concepts • **Software and its engineering → Embedded software**; *Runtime environments*; *Integrated and visual development environments*;

Keywords education, classroom, embedded systems

ACM Reference Format:
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 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3211332.3211335

LCTES'18, June 19–20, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5803-3/18/06.
https://doi.org/10.1145/3211332.3211335

Figure 1. Example projects undertaken within education: Rishworth School sent a micro:bit to space [19] (left); The micro:bit placed in a custom-built rocket car for telemetry [20, 21] (right).

1 Introduction

Recent years have witnessed expansive growth in the popularity and ubiquity of embedded systems. This growth can be primarily attributed to the emergence of new application domains ranging from wearables, to home automation, industrial automation, and smart grids – a phenomenon broadly referred to as the Internet of Things (IoT). As the IoT continues to grow, it has become more pervasive – far beyond the realm of domain experts and into the everyday lives of the public. This has led to growth in non-expert developers actively creating software for embedded systems. Small to medium sized businesses now create new products through *rapid prototyping* of embedded devices. Hobbyist *makers* create novel technical projects to inspire themselves and society. And now, more than ever before, *educators* are making extensive use of *physical computing* devices as a direct means to teach and inspire a generation of students – and to prepare them for a society where IoT will be the norm. These new developers all share a common characteristic: **they are not professional software developers** [9, 10, 16].

To address this trend, in 2015 a consortium of industry and academic partners came together to develop the BBC micro:bit – an embedded device designed specifically for education. One million of these devices were delivered to UK school children in 2016. The micro:bit is a highly capable IoT device containing a 32-bit ARM Cortex-M0 processor, integrated light level, temperature, acceleration and magnetic sensors, touch sensitive inputs, and USB and 2.4GHz radio communications.

Figure 1 highlights two example educational projects based on the BBC micro:bit. The first is a data gathering experiment, where multiple sensor data was recorded to non-volatile memory as the device was launched into near space (32.5km altitude), along with an externally interfaced camera and GPS unit. The second highlights a live data telemetry application, where acceleration data was streamed in real-time via Bluetooth Low Energy to enable the profiling of chemical-rocket powered model vehicles. These projects are highly sophisticated and were undertaken by high school educators and their students.

We introduce an open-source[1] platform for embedded devices such as the micro:bit that enables the development of embedded applications by non-expert programmers. This platform consists of two major components: 1) Microsoft MakeCode (www.makecode.com), a web app providing a beginner IDE for embedded systems; and 2) CODAL (Component-Oriented Device Abstraction Layer), an efficient C++ runtime with high-level programming abstractions.

In the remainder of the introduction, we present the major design challenges in bringing embedded systems into education, briefly describe the architecture of our solution, and give an overview of the paper and our results.

1.1 Design Challenges

Enabling novice programmers to successfully develop embedded applications is a non-trivial task. Throughout our research we have identified a number of design challenges that we address through MakeCode and CODAL.

High Level Languages: Programming languages for microcontroller units (MCUs) have not kept pace with advances in hardware. Despite active research in the field, the C/C++ languages remain the standard for embedded systems: they provide a familiar imperative programming model, with compilers that produce highly efficient code, and enable low level access to hardware features when necessary. The Arduino project (www.arduino.cc), started in 2003, and ARM's Mbed platform (www.mbed.org) both rely heavily on a C/C++ programming model [6, 24]. However, the limitations of using C/C++ as an application programming language for inexperienced developers are well understood [8]. *To address this, higher level languages such as JavaScript, Python, and even visual programming languages are required.*

Zero Installation Architecture: The development environment for existing embedded systems typically requires the installation of code editors, custom device drivers, compiler toolchains, and even additional programming hardware (such as a JLink programmer). For many, particularly in the field of education, this presents a high adoption barrier as in many schools, custom hardware and software is simply not permitted by policy and/or access to the necessary technical support is not present. *An effective solution must therefore provide a fully transparent, platform agnostic, zero installation experience to developing embedded software.*

[1] MakeCode is open-source at https://github.com/microsoft/pxt; CODAL is open-source at https://github.com/lancaster-university/codal.

Table 1. Example microcontroller devices in relationship to a typical PC. Device abbreviations: Uno (Arduino Uno), micro:bit (BBC micro:bit), CPX (Adafruit Circuit Playground Express), PC (Personal Computer).

Device	RAM	Flash	Word Size	CPU Speed	CPU
Uno	2 kB	32 kB	8	16MHz	AVR
micro:bit	16 kB	256 kB	32	16MHz	ARMv6-M
CPX	32 kB	256 kB	32	48MHz	ARMv6-M
PC	16 GB	1 TB	64	3GHz	x86-64

Optimization for Code Efficiency: The projects of Figure 1 are enabled by small, highly resource-limited programmable MCUs, which may have as little as a few kilobytes of RAM and FLASH memory. Table 1 compares the core capabilities of the class of MCU-based devices typically used in the education domain to a typical PC. Note that these devices have a *proportionally* large amount of processing power, relative to their storage. Consider the BBC micro:bit vs. a typical PC: the micro:bit has about 100 times less CPU power, but 10^6 times less RAM, and 10^6 times less storage. *A language/runtime should therefore not only seek to provide high code density and spatial efficiency, but actively trade off temporal for spatial efficiency where possible.*

Asynchronous and Concurrent Programming: It is already well understood that novice programmers benefit from event-based programming paradigms [17, 18, 26]. This is increasingly relevant for embedded systems due to the typically asynchronous nature of their hardware. MCUs still follow Moore's law, but this additional capacity is not typically invested in speeding up processors. Instead, more independent peripherals (such as Bluetooth/WiFi radio modules, audio inputs/output processors, etc.) are integrated onto the same package as the CPU as a *system-on-chip*. Such peripherals often operate independently of the main CPU. *An effective language/runtime should directly support an asynchronous interaction model designed to cooperate with the independent nature of peripherals while remaining highly intuitive to the programmer.*

Intuitive and Extensible APIs: Intuitive APIs and programming models are required to support novice users, yet it is equally important that these APIs remain complete enough to realise the ambitious projects that may be undertaken as students advance: simplification via the reduction of functionality is not a valid approach. *An effective solution must provide APIs that are consistent, easy to understand/use, and progressive, to address the growing capabilities of the programmer.*

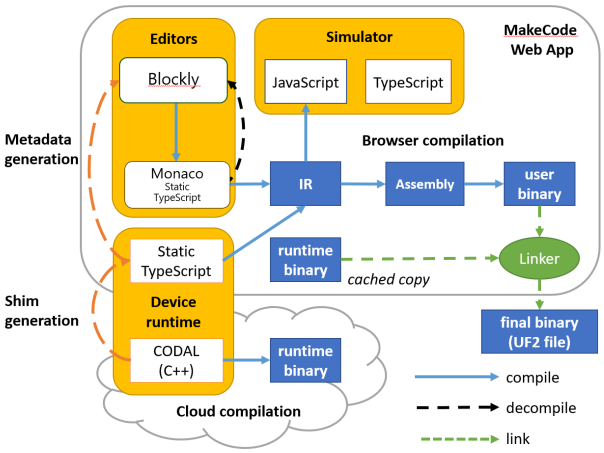


Figure 2. The MakeCode and CODAL Architecture

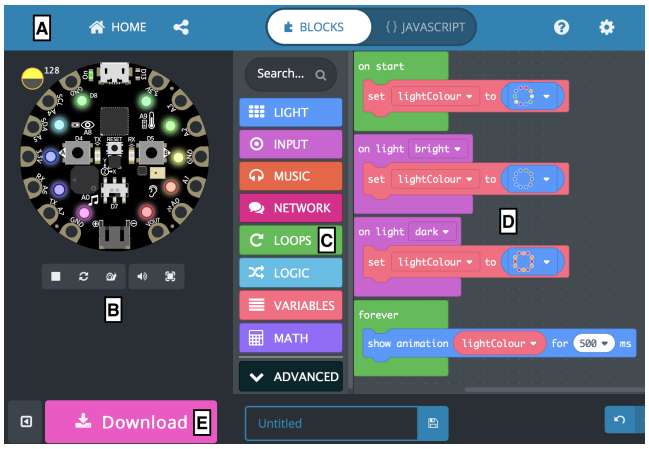


Figure 3. Screen snapshot of the MakeCode web app.

1.2 Architecture

Figure 2 illustrates the architecture of our platform. The MakeCode web app is the primary entry point for the end-user. MakeCode supports the simplified programming of MCUs via editors for visual blocks and the textual TypeScript[2] language. CODAL is a component-oriented, event-driven, fiber-based C++ runtime environment that bridges the semantic gap between the higher-level languages (such as TypeScript) and the hardware (bottom-left of the Figure). Enabling the flashing of the microcontroller is UF2, a new file format and bootloader for the simplified transfer of binaries to the device over USB (bottom-right).

MakeCode can be accessed from any modern web browser and cached locally for *entirely offline use*. The MakeCode web app incorporates the open-source Blockly[3] and Monaco[4] editors (upper-left), an in-browser device simulator (upper-right) for testing programs before transferring them to the physical device, as well as *in-browser compilation* of TypeScript to machine code and linking against the pre-compiled CODAL C++ runtime (lower-left).

MakeCode devices appear as USB pen drives when plugged into a computer, thanks to UF2. After a user has finished developing a program, the compiled binary is "downloaded" locally to the user's computer (lower-right) and then transferred (flashed) to the MCU by a simple file copy operation. This works out-of-the-box on any OS with built-in support for USB pen drives (MacOS, Windows, Linux, ChromeOS).

1.3 Overview

The remainder of the paper describes the design, implementation, and evaluation of MakeCode (Section 2), the CODAL C++ runtime (Section 3), and the UF2 bootloader (Section 4). Section 5 shows that combined, these technologies enable simplified programming while maintaining a relatively high degree of temporal and spatial efficiency. We demonstrate up to 50x better performance than other state-of-the-art implementations, in some cases nearing the performance of native C++. MakeCode has been live since the fall of 2016, at first supporting just the micro:bit, but now supporting many more devices, most of which are based on CODAL. Section 6 discusses related work and Section 7 concludes the paper.

2 MakeCode

The key technical contribution of MakeCode is to provide users with a basic environment to write programs for MCUs, enabling a simple progression from visual block-based programming to text-based programming in Static TypeScript (STS), while leveraging C++ on the backend for efficient use of MCU resources. MakeCode uses the open-source Blockly and Monaco editors to allow the user to code with visual blocks or STS. The editing experience is parameterized by a fully-typed device runtime, which provides a set of categorized APIs to the end-user.

Figure 3 is a screenshot of the MakeCode web app for Adafruit's Circuit Playground Express (CPX) device[5]. The web app has five sections: (A) the menu bar allows switching between the two editors; (B) the simulator shows the CPX board and provides feedback on user code executed in the browser; (C) the toolbox provides access to device-specific APIs and programming elements; (D) the programming canvas is where editing takes place; (E) the download button invokes the compiler, producing a binary executable.

The web app encapsulates all the components needed to deliver a programming experience for MCUs, free of the need for a C++ compiler for the compilation of user code. The web app is written in TypeScript and incorporates the TypeScript compiler and language service as well. The app

[2] https://www.typescriptlang.org
[3] https://github.com/google/blockly
[4] https://github.com/microsoft/monaco-editor
[5] https://makecode.adafruit.com

is built from a MakeCode "target" which parameterizes the MakeCode framework for a particular device. The remaining subsections describe the essential components of the web app in Figure 2.

2.1 Static TypeScript

TypeScript is a typed superset of JavaScript designed to enable JavaScript developers to take advantage of code completion, static checking and refactoring made possible by types. As a starting point, every JavaScript program is a TypeScript program. Types can be added gradually to JavaScript programs, supported by type inference. While TypeScript provides classes and interfaces with syntax like Java/C#, their semantics are quite different as they are based on JavaScript. Classes are simply syntactic sugar for creating objects that have code associated with them, but these objects are indeed JavaScript objects with all their dynamic semantics intact.

Static TypeScript (STS) is closely related to StrongScript [23], which extends TypeScript with a type constructor for *concrete* types, allowing the programmer to choose between untyped, optionally-typed, and concretely typed code. This provides traditional type soundness guarantees, as in Java and C#. STS can be seen to be StrongScript where every variable and expression has a concrete type. As in StrongScript, classes are nominally typed, which permits a more efficient and traditional property lookup for class instances. Currently, STS goes further than StrongScript by outlawing downcasts.

2.2 Device Runtime and Shim Generation

A MakeCode target is defined, in part, by its device runtime; which is a combination of C++ and STS code, as shown in the lower-left of Figure 2. All the target's C++ files are precompiled (by a C++ compiler in the cloud) into a single binary, which is stored in the cloud as well as in the HTML5 application cache. Additional runtime components may be authored in STS, which allows the device runtime to be extended without the use of C++, and permits components of the device runtime to be shared by both the device and simulator runtimes. The ability to author the device runtime in both STS and C++ is a unique aspect of MakeCode's design.

Whether runtime components are authored in C++ or STS, all runtime APIs are exposed as fully-typed TypeScript definitions. A fully-typed runtime improves the end-user experience by making it easier to discover APIs; it also enables the type inference provided by the TypeScript compiler to infer types for (unannotated) user programs.

MakeCode supports a simple foreign function interface from STS to C++ based on namespaces, enumerations, functions, and basic type mappings. MakeCode uses top-level namespaces to organize sets of related functions. Preceding a C++ namespace, enumeration, or function with a comment starting with //% indicates that MakeCode should map the C++ construct to STS. Within the //% comment, attributes specify the visual appearance for that language construct, such as for the input namespace in C++ for the CPX:

```
1 //% color="#B4009E" weight=98 icon="\uf192"
2 namespace input { ...
```

Figure 3(C) shows the toolbox of API and language categories, where the category *INPUT* corresponding to the namespace *input* can be seen (second from the top).

Mapping of functions and enumerations between C++ and STS is straightforward and performed automatically by MakeCode. For example, the following C++ function *onLightConditionChanged* in the namespace *input* wraps the more complex C++ needed to update the sensor and register the (Action) handler with the underlying CODAL runtime:

```
1 //% block="on light %condition"
2 void onLightConditionChanged(LightCondition
      condition, Action handler) {
3     auto sensor = &getWLight()->sensor;
4     sensor->updateSample();
5     registerWithDal(sensor->id, (int)condition,
          handler);
6 }
```

MakeCode generates TypeScript declaration file (here called a shim file) to describe the TypeScript elements corresponding to C++ namespaces, enumerations and functions. Since the C++ function above is preceded by a //% comment, MakeCode adds the following TypeScript declaration to the shim file and copies over the attribute definitions in the comment. MakeCode also adds an attribute definition mapping the TypeScript shim to its C++ function:

```
1 //% block="on light %condition"
2 //% shim=input::onLightConditionChanged
3 function onLightConditionChanged(condition:
      LightCondition, handler: () => void): void;
```

Since the //% comment also contains a *block* attribute, MakeCode creates a block (named "on light"), which can be seen in the upper-left of Figure 3(D).

To support the foreign function interface, MakeCode defines a mapping between C++ and STS types. Boolean and void have straightforward mappings from C++ to STS (bool → boolean, void → void). As JavaScript only supports number, which is a C++ double, MakeCode uses TypeScript's support for type aliases to name the various C++ integer types commonly used for MCU programming (int32, uint32, int16, uint16, int8, uint8). This is particularly useful for saving space on 8-bit architectures such as the AVR. MakeCode also includes reference-counted C++ types for strings, lambdas (Action in C++, with up to three arguments and a return type) and collections, with mappings to STS.

MakeCode does not yet include a garbage collector, so advanced users who create cyclic data structures must be careful to break cycles to ensure complete deallocation.

2.3 Browser Compilation

When a user requests a download of the compiled binary, MakeCode first invokes the TypeScript language service to perform type inference and type checking on the user's program, the device runtime written in STS, and the TypeScript declarations corresponding to the C++ device runtime. It then checks that the combined TypeScript program is within the STS subset through additional syntactic and type checks over the typed AST. Assuming all the above checks pass, MakeCode then performs tree shaking of the AST to remove unused functions. The reduced AST is then compiled to an intermediate representation (IR) that makes explicit labelled control flow among a sequence of instructions with conditional and unconditional jumps, heap cells, field accesses, store operations, and reference counting.

There are three backends for code generation from the IR. The first backend generates JavaScript, for execution against the simulator runtime. A second backend, parameterized by processor type, generates assembly code. Currently supported processors include ARM's Cortex class (Thumb instructions) and Atmel's ATmega class (AVR instructions). A separate assembler, also parameterized by an instruction encoder/decoder, generates machine code and resolves runtime references, producing a final binary executable. A third backend generates bytecode instructions. MakeCode can encode the resulting binary in several formats, including Intel's HEX format [13] and the UF2 format, documented in Section 4.

The MakeCode compiler supports the STS language subset of TypeScript with two compilation strategies: untagged and tagged. Under the untagged strategy, a JavaScript number is interpreted as a C++ `int` by default and the type system is used to statically distinguish primitive values from boxed values. As a result, the untagged strategy is not fully faithful to JavaScript semantics: there is no support for floating point, and the `null` and `undefined` values are represented by the default integer value of zero. The untagged strategy is used for the micro:bit and Arduino Uno targets.

In the tagged strategy, numbers are either tagged 31-bit signed integers, or if they do not fit, boxed doubles. Special constants like `false`, `null` and `undefined` are distinguished by specific values. The tagged execution strategy has the capability to fully support JavaScript semantics and is used by all ATSAMD21 targets, including the CPX.

3 The CODAL Runtime

CODAL is a lightweight, object-oriented, componentized C++ runtime for microcontrollers designed to provide an efficient abstraction layer for higher level languages, such as JavaScript. CODAL has five key elements:

1. *a unified eventing subsystem* (common to all components) that provides a mechanism to map asynchronous hardware and software events to event handlers;

```
1  #include "CircuitPlayground.h"
2  CircuitPlayground cplay;
3
4  void onBright() { // user defined code }
5
6  int main() {
7      cplay.messageBus.listen(ID_LIGHT_SENSOR,
           LIGHT_THRESHOLD_HIGH, onBright);
8  }
```

Figure 4. Example of the CODAL MessageBus.

2. *a non-preemptive fiber scheduler* that enables concurrency while minimizing the need for resource locking primitives;
3. *a simple memory management system* based on reference counting to provide a basis for managed types;
4. *a set of drivers*, that abstract microcontroller hardware components into higher level software components, each represented by a C++ class;
5. *a parameterized object model* composed from these components that represents a physical device.

There are discussed in detail below.

3.1 Message Bus and Events

CODAL offers a simple yet powerful model for handling hardware or user defined events. Events are modeled as a tuple of two integer values - specifying an *id* (namespace) and a *value*. Typically, an id correlates to a specific software component, which may be as simple as a button or something more complex as a wireless network interface. The value relates to a specific event that is unique within the id namespace. All events pass through the CODAL MessageBus. Application developers can then *listen* to events on this bus, by defining a C/C++ function to be invoked when an event is raised. Events can be raised at any time simply by creating an *Event* C++ object, which then invokes the event handlers of any registered listeners.

Continuing the example of detecting the brightness of a room used in Section 2.2, a listen call to the MessageBus with a component ID of the light sensor and a threshold event is the underlying mechanism enabled by the runtime, as illustrated by the equivalent C++ code snippet in Figure 4.

Unlike simple function pointers, CODAL event handlers can be parameterized by the event listener to provide decoupling from the context of the code raising the event. The receiver of an event can choose to either receive an event in the context of the fiber that created it, or can be decoupled and executed via an Asynchronous Procedure Call (APC). The former provides performance, while the latter provides decoupling of low level code (that may be executing, say, in an interrupt context) from user code. Each event handler may also define a threading model, so they can be reentrant or run-to-completion depending upon the semantics required.

3.2 Fiber Scheduler

CODAL provides a *non-preemptive* fiber scheduler with asynchronous semantics and a power efficient implementation. CODAL fibers can be created at any time but will only be descheduled as a result of an explicit call to yield(), sleep() or wait_for_event() on the MessageBus. The latter enables condition synchronization between fibers through a wait/notify mechanism. A round-robin approach is used to schedule runnable fibers. If at any time all fibers are descheduled, the MCU hardware is placed into a power efficient sleep state.

The CODAL scheduler makes use of two novel mechanisms to optimize for MCU hardware. Firstly, CODAL adopts a stack paging approach to fiber management. MCUs do not support virtual memory and are heavily RAM constrained, but relatively cycle rich. Therefore, instead of overprovisioning stack memory for each fiber (which would waste valuable RAM), we instead dynamically allocate stack memory from heap space as necessary and copy the physical stack into this space at the point at which a fiber is descheduled (and similarly restored when a fiber is scheduled). This copy operation clearly incurs a small CPU overhead, but brings greater benefits of RAM efficiency - especially given that MCU stack sizes are typically quite small (~200 bytes is typical).

Secondly, the CODAL scheduler supports transparent APCs. Any function can be *invoked* as an APC. Conceptually, this is equivalent to calling the given function in its own fiber. However, the CODAL runtime provides a common-case transparent optimization for APCs we call *fork-on-block* - whereby a fiber will only be created at the point at which the given function attempts a blocking operation such as sleep() or wait_for_event(). Functions which do not block therefore do not incur all of the context switch overhead.

When invoking an APC, the scheduler snapshots the current processor context and stack pointer (but not the whole stack). If the scheduler is re-entered before the APC completes, a new fiber context is created at the point of descheduling, and placed on the appropriate wait queue. The previously stored context is then restored, and execution continues from the point at which the APC was first invoked. This mechanism provides potentially high RAM savings for the processing of MessageBus event handlers in particular.

CODAL's scheduling and eventing models are shared by both high and low level languages, and therefore handled uniformly. As a result, when a foreign function call is mapped to C++, that C++ function is capable of blocking the calling fiber without infringing on the concurrency model of the higher level language. This enables, for example, a C++ device driver to block a JavaScript program when awaiting data without changing the behavior of other JavaScript code acting asynchronously (as in Figure 3).

3.3 Memory Management

CODAL implements its own lightweight heap allocator, introducing reentrant versions of the libc malloc family of functions, permitting universal access to heap memory in user or interrupt code. The heap allocator is flexible and reconfigurable, allowing the specification of multiple heaps across memory and it is optimized for repeat allocations of memory blocks that are commonplace in embedded systems.

CODAL also makes use of simple managed types, built using C++ reference counting mechanisms. C++ classes are provided for common types such as strings, images, and data buffers. A generic base class is also provided for the creation of other managed types. This simple approach brings the benefits of greater memory safety for application code, but with the expense of suffering from the issues related to circular references. We take the view that such scenarios are rare in MCU applications, justifying this approach over a more complex garbage collection scheme and its overhead.

3.4 Device Driver Components

CODAL drivers abstract away the complexities of the underlying hardware into reusable, extensible, easy-to-use components. For every hardware component there is a corresponding software component that encapsulates its behavior in a C++ object. CODAL has three types of drivers:

1. A hardware agnostic abstract specification of a driver model (e.g. a Button, or an Accelerometer). This is provided as a C++ base class.
2. The concrete implementation of the abstract driver model, which is typically hardware specific. This is implemented as a subclass of a driver model, such as a LIS3DH accelerometer, as manufactured by ST Microelectronics.
3. A high level driver that relies only on the interfaces specified in a driver model (e.g. a gesture recognizer based on an Accelerometer model).

This approach brings the benefits of abstraction and reusability to CODAL, without losing the hardware specific benefits seen in flat abstraction models where every MCU is made to look the same, even though their capabilities are different (as in the Arduino and mbed APIs, for example).

Finally, we group together the components of a physical device to form a *device model*. This is a singleton C++ class that, through composition of device driver components, provides a configured representation of the capabilities of a device. Such a model allows: an elegant OO API for programming a device, and a static representation that forms an ideal target for the MakeCode linker to bind high level STS interfaces to low level optimized code.

MakeCode is further supported by an annotated C++ library (*MakeCode wrappers*) defining the mapping from CODAL to TypeScript and Blockly. The use of MakeCode wrappers ensures that different MakeCode targets that use CODAL share a common TypeScript and block API vocabulary[6].

4 UF2

The UF2 bootloader enables efficient, universal microcontroller programming through a USB pen drive interface — no drivers are required, as operating systems support pen drives out of the box. The UF2 bootloader builds upon the work of DAPLink [3], but offers a simpler implementation via a new flashing format, UF2.

DAPLink exposes a small virtual 512-byte block FAT file system (VFS), with an empty file allocation table and root directory. When the OS tries to read a block, DAPLink computes what should be there. During file system writes, DAPLink detects blocks of files in Intel HEX format [13], decodes them, and flashes the file's contents into the target microcontroller's memory. Other file system writes are ignored.

DAPLink implements many heuristics to deal with quirks of FAT file system implementations in various operating systems (order of writes, various meta-data files that are created and need to be ignored, etc.). However, we have found that the heuristics are fragile with respect to operating system changes, which are not infrequent.

Our new file format, UF2, consists of one or more 512-byte self-contained blocks (aligned to the block size of the VFS), removing the need for the complex heuristics. The blocks have magic numbers, the payload data to be written to flash, and the address where it should be written. Thus, on every 512-byte write via the USB controller, the bootloader can quickly and easily check if the block being written is part of a UF2 file (by comparing magic numbers) and if so, write it immediately in a streaming fashion.

For simplicity, the 512-byte UF2 blocks usually contain 256 bytes of payload data. While 50% density might seem low, the industry standard 16-byte-per-line HEX format has a density of around 35%. However, the files are small by modern computer standards (under 1000k) and we have not found the lower density to be a problem. On the MCU side, the bottleneck there is speed of flash erase, not the USB bus (which reaches 1000k/s).

The minimal implementation of the UF2 bootloader consumes *just 1-2 kB of flash memory* and *less than 100 bytes of RAM*, with some variability due to the microcontroller instruction set and USB hardware interfaces in use.

5 Evaluation

Our platform has been actively deployed for over a year, bringing the benefits of a safe programming environment for MCUs to hundreds of thousands of active users. In this section we provide a broad, quantitative evaluation of the cost at which these benefits are realized. We do this with several micro-benchmarks that give insight into the performance of MakeCode and CODAL across the Uno, micro:bit, and CPX devices. We break down results by layer (CODAL and MakeCode) to give an insight into how each performs.

5.1 Benchmarks, Devices, and Methodology

To analyze the performance of our solution, we have written a suite of programs to evaluate different aspects of MakeCode and CODAL on a representative selection of real hardware devices. Throughout, we use the C++ CODAL benchmarks as a baseline; the STS benchmarks show the overhead added by MakeCode. These programs were written in both C++ and STS, and evaluated on the three devices listed in Table 1: The micro:bit (Nordic nRF51 MCU), the CPX (Atmel ATSAMD21 MCU), and the Uno (Atmel ATmega MCU).

The Uno is the simplest of these devices, consisting of an 8-bit processor running at 16 MHz, with only 2kB of RAM and 32kB of flash. The micro:bit has a 32-bit Cortex-M0 clocked at 16MHz, with 16kB RAM and 256kB of flash. The CPX is a 32-bit Cortex-M0+, which offers greater energy efficiency and performance; it clocks at 48 MHz, has 32kB of RAM and 256kB of flash. The Uno and micro:bit MakeCode targets use the untagged compilation strategy, while the CPX target uses the tagged strategy (see Section 2.3). The benchmarks are classified into two types, each with their own methodology:

1. *Performance Analysis*: Tests that capture time taken to perform a given operation. For these benchmarks, we toggle physical pins on the device at key points in the test code. We then measure the time to execute the operation, by using a calibrated oscilloscope observing these pins. This allows us to derive highly accurate real time measurements without biasing the experiment.
2. *Memory Analysis*: Tests that capture the RAM or FLASH footprint of a certain operation. A map of memory is logged before and after the execution of an operation, allowing us to compute the cost. A serial terminal captures the output of these tests.

Note that memory and performance analysis are done in separate runs to ensure logging does not affect time-related measurements.

5.2 Tight Loop Performance

To place the performance of MakeCode in context, we perform a comparative evaluation of MakeCode against two state-of-the-art solutions adopted by educators in the classroom, using native C++ as our baseline. The two points of comparison are MicroPython [12], an implementation of Python for MCUs, and Espruino [29], an implementation of JavaScript for MCUs. For the CPX, a fork of MicroPython known as "CircuitPython" was used. Both MicroPython and Espruino use virtual machine (VM) approaches.

[6]See https://github.com/microsoft/pxt-common-packages

Table 2. A comparison of execution speed between: native C++ with CODAL; MakeCode compiled to native machine code; MakeCode compiled to AVR VM; MicroPython; and Espruino. The first line lists the C++ time, while subsequent lines are slowdowns with respect to the C++ time. The 6.4x slowdown of MakeCode VM compared to native MakeCode on the Uno is compensated with 5x better code density.

	UNO	micro:bit	CPX
CODAL	171ms	102ms	31ms
MakeCode	2.4x	2.1x	7.3x
MakeCode VM	15.3x	-	-
MicroPython	-	101x	183x
Espruino	-	1139x	-

To give an indicative general case execution time cost of each solution, we created a simple program that counts from 0 to 100,000 in a tight loop in each solutions' respective language; the results are shown in Table 2. On AVR we count to 25,000 (to fit within a 16 bit `int`) and scale up the results.

For MicroPython and Espruino on the micro:bit, the run is *two or more orders of magnitude slower* than a native CODAL program. MakeCode performs only 2x slower. The slowdown reflects the simple code generator of our STS compiler. It should be noted that MakeCode for the CPX uses the tagged approach, which allows for seamless runtime switching to floating point numbers, resulting in a further 3x slowdown. For both devices, we can observe that MakeCode outperforms both the VM-based solutions of MicroPython and Espruino by at least an order of magnitude.

MicroPython and similar environments cannot run on the Uno due to flash and RAM size limitations. We also ran into these limitations, and as a result, developed two compilation modes for AVR. One compiles STS to AVR machine code, and the other (MakeCode VM) generates density-optimized byte code for a tiny (~500 bytes of code) interpreter. The native strategy achieves code density of about 60.8 bytes per statement, which translates into space for 150 lines of STS user code. The VM achieves 12.3 bytes per statement allowing for about 800 lines. For comparison, the ARM Thumb code generator used in other targets achieves 37.5 bytes per statement, but due to the larger flash sizes we did not run into space issues.

5.3 Context Switch Performance

To evaluate the performance of CODAL's scheduler we conducted a test that created two fibers, continuously swapped context, and measured the time taken to complete a context switch. We performed this test in both STS and C++ and the resulting profiles can be seen in Figure 5, which breaks the context switch down into three phases: (1) CODAL, the time it takes to perform a context switch in CODAL; (2) Stack, the time taken to page out the MakeCode stack; and (3) MakeCode, the overhead added by MakeCode.

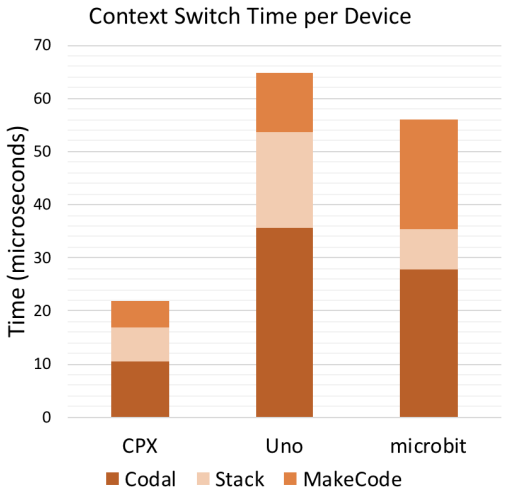


Figure 5. Base context switch profiles per device.

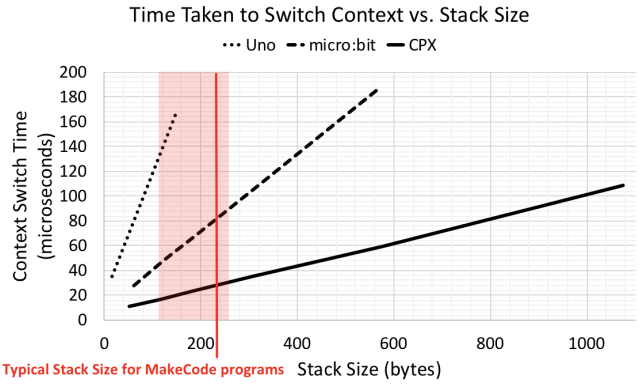


Figure 6. Time taken to perform a context switch against stack size.

From these results, we observe that context switches generally take tens of microseconds. The cost of CODAL's stack paging approach can also be a significant, but not dominant cost. The cost of stack paging would of course grow with stack depth. Figure 6 profiles the time a context switch takes with an increasing stack size across all three devices in CODAL. This is similar to the previous test, except we placed bytes (in powers of 2) on the stack of each fiber, starting from 64 and finishing at 1024. The difference in gradients, and ranges of values can be put down to device capability. For instance, the Uno has an 8-bit word size, which means more instructions are required to copy the stack, this results in a steeper gradient than the other two devices. The vertical band indicates typical stack sizes for MakeCode programs based on a representative set of examples.

Table 3. Flash consumption of a MakeCode binary (kB)

	CPX	micro:bit	Uno
MakeCode	20.46	12.14	7.79
CODAL	29.85	34.35	13.7
Supporting Libraries	14.99	24.28	-
C++ Standard Library	43.14	24	1.03

5.4 Performance of Asynchronous Operations

To gauge the cost of asynchronous operations in CODAL, we tested three commonly used code paths, designed to determine the efficiency of CODAL's *fork-on-block* Asynchronous Procedure Call (APC) mechanism that underpins all event handlers in MakeCode and CODAL. We measured the RAM and processor cost of: (1) creating a fiber; (2) handling a non-blocking APC call; and (3) handling a blocking APC call. We used the CPX for this experiment.

Non-blocking APC calls, the best case, have a small overhead of 32 bytes of RAM and 4.01 microseconds of processing time. Blocking APC calls, the worst case, incur a large overhead of 204 bytes of RAM and 32.4 microseconds of processor time. Creating a fiber costs 136 bytes of RAM and 35.4 microseconds of processing time. These results highlight the performance gains of the opportunistic fork-on-block mechanism over a naive approach that would execute every event handler in a separate fiber.

5.5 Flash Memory Usage

MCUs make use of internal non-volatile FLASH memory to store program code. Table 3 shows the per device flash consumption of each software library used in the final MakeCode binary. To obtain these numbers, we analyzed the final map file produced after compilation. The ordering of the table aligns with the composition of the software layer: MakeCode builds on CODAL which builds on the C++ standard library and supporting libraries. MakeCode and CODAL consume 108 kB of flash, whereas CircuitPython consumes 201 kB, MicroPython consumes 228 kB, and Espruino consumes 142 kB of flash. This means that users can write sizeable applications in MakeCode, without the worry of running out of flash memory.

From the bottom up, the profile of the standard library changes dramatically for each device: The Uno has a very lightweight standard library; the micro:bit uses 64-bit integer operations (for timers) which requires extra standard library functions; and the CPX requires software floating point operations pulling in more standard library functions.

The size of CODAL and MakeCode scales linearly with the amount of functionality a device has, due to the component oriented nature of CODAL and transitively MakeCode. For instance, the Uno has few onboard components when compared to the CPX and micro:bit. The modular composition of CODAL allows us to support multiple devices with a variety of feature sets, while maintaining the same API at the MakeCode layer.

Table 4. Static RAM consumption of a MakeCode binary (kB)

	CPX	micro:bit	Uno
MakeCode	0.612	1.069	0.074
CODAL	0.369	0.214	0.156
Supporting Libraries	0.312	0.923	-
C++ Standard Library	0.161	0.149	0.074

5.6 RAM Memory Usage

Table 4 shows the per device RAM consumption of each software library used in the final MakeCode binary. To obtain these numbers, we analyzed the final map file produced after compilation. At runtime, MakeCode dynamically allocates additional memory: 1.56 kB for the CPX, 560 bytes for the micro:bit, and 644 bytes for the Uno. We also can see that in all cases, the RAM consumption of MakeCode and CODAL is well within the RAM available of each device.

MakeCode and CODAL consume a small amount of resources in comparison: CircuitPython (a derivative of MicroPython) consumes 12.8 kB, MicroPython consumes 9.5 kB, and Espruino consumes 5.3 kB of RAM. On the micro:bit, the Bluetooth stack requires 8 kB of RAM to operate. Due to MicroPython's RAM consumption this means that Bluetooth is inoperable. Comparatively, Espruino does enable the Bluetooth stack, but users have just ~300 bytes available for their programs due to the overhead incurred.

5.7 Compiling Static TypeScript

During compilation, the entire STS program (including the STS runtime) is passed to the TypeScript (TS) language service for parsing. Then, only the remaining part of the program (after code shaking) is compiled to native code. On a modern laptop, using Node.js, TS parsing and analysis takes about 0.1ms per statement, and MakeCode compilation to native code takes about 1ms per statement. While the TS compiler has been optimized for speed, MakeCode's native compilation process has not. For example, the CPX TS pass is dominated by compilation of the device runtime and takes about 100ms, whereas the MakeCode pass typically only includes a small user program and a small bit of the runtime, resulting in less than 100ms. Thus, compilation times are under 200ms for typical user programs of 100 lines or less.

5.8 Extensibility

Adding a new device in CODAL is trivial once a MCU has been ported. The porting of a MCU is where we observe the largest development overhead, as low-level implementations of drivers for I2C, Serial, and SPI may have to be re-written. Due to CODAL's abstraction model, once low-level drivers have been implemented, drivers for higher level components like Accelerometers (which depend on high-level interfaces for low-level drivers) can be immediately adopted if hardware is present. A similar technique is used in MakeCode for simulators.

6 Related Work

6.1 Novice Programming Environments

Arduino [24] is an environment for programming microcontrollers, aimed at novices. However, its C++ based APIs introduces barriers for novice programmers [8]. Scratch [22] is a widely adopted, event-based visual programming environment designed to introduce novice programmers to computer science concepts. Extensions enable the programming of physical devices with Scratch. However, devices require constant tethered connections to operate, restricting potential projects [10]. ArduBlock [2] brings visual programming to the Arduino, but it lacks the event-based blocks Scratch users are familiar with.

With the environments above, additional software must be installed — this creates barriers for novice users in restrictive environments. MakeCode and CODAL require *no installation* to support a diverse user base and support *event-based higher-level languages* to help beginners get a head start in the world of the microcontroller.

6.2 Virtual Machine-based Languages

Recently, virtual machines supporting most of the semantics of higher level languages like JavaScript, Java, and Python, have been ported to 32-bit microcontrollers by maker communities [10]. Examples include: MicroPython [12], CircuitPython, and Espruino [29]. These VMs consume a large amount of RAM and flash memory, and run significantly slower than native languages.

The research community has worked to bring higher level languages to microcontrollers [14, 25, 28]. Rather than running a full-featured VM, others enable higher level languages to run efficiently by stripping out advanced language features, in favor of efficient, native execution [27]. Comparing these solutions to our solution is challenging due to a misalignment in evaluation metrics and microcontrollers. For example, the PICOBit uses an 8-bit MCU, and evaluates the cost of a VM, without the cost of a runtime environment. Simply accounting for a 32-bit MCU in this case, results in factor of 4 multiplication of most metrics.

Our approach bears most similarity to [27], where we compile higher level languages to an *optimized, event-driven* C++ runtime (CODAL).

6.3 Embedded Runtime Environments

Arduino [24] is an example of a simple platform where the developer uses high-level APIs to control hardware; there is no scheduler and memory management is discouraged, with a heavy emphasis on the use of global variables.

TinyOS [15], Contiki [11], RIOT OS [7], Mynewt [4], mbed OS [6], and Zephyr [5] are RTOS solutions known widely in the systems community. The majority focus on the networking features of sensor based devices and commonly adopt a preemptive scheduling model, which leads to competition over resources resolved using locks and condition synchronization primitives. Contiki has a cooperative scheduler but uses proto-threads to store thread context — local variables are not allowed as the context of the stack is not stored.

Although the platforms above are widely used by C/C++ developers, none of these existing solutions align well with the programming paradigms seen in higher level languages. CODAL *bridges the semantic gap between the higher level language and the microcontroller*, offering appropriate abstractions and higher level primitives written natively in C++.

6.4 Flashing Microcontrollers

There are two common ways to transfer a program to the flash of a microcontroller: for embedded developers, a specialized debugger chip; for hobbyists, a custom serial protocol [1]. Both approaches require operating system drivers. ARM's mbed platform provides DAPLink [3], firmware that presents itself to an external computer as a USB pen drive. DAPLink exposes a virtual file system that caters for normal file system behavior and handles the decoding of Intel HEX files [13] — the firmware consumes 66 kB of flash and 13 kB RAM. UF2 contributes a new file format that *greatly simplifies* the virtual file system approach, reducing complexity of the firmware and code size.

7 Conclusion

We have presented MakeCode: a *no installation*, web-based programming environment, that supports novice programmers with *block-based and text-based higher-level languages*, and compiles programs *in the browser*. So as to not compromise the spatial efficiency of the microcontroller, we created CODAL: a C++ runtime that *bridges the semantic gap* between higher level languages in MakeCode and C++. To transfer programs compiled by MakeCode to the microcontroller without the installation of any drivers, we created UF2: a new bootloader and file format that enables the *simplified*, *driverless* programming of microcontrollers.

Combined, our approach to running higher level languages on microcontrollers is up to 50x more performant compared to other approaches. Further, by using modern tooling, and higher level languages, our approach lowers the barrier to entry for microcontroller programming.

Acknowledgements

The authors would like to thank the members of the MakeCode team for their many contributions to the success of the platform: Abhijith Chatra, Sam El-Husseini, Caitlin Hennessy, Guillaume Jenkins, Richard Knoll, Galen Nickel, Jacqueline Russell, and Kesavan Shanmugam.

A Artifact appendix

Submission and reviewing guidelines and methodology: *http://cTuning.org/ae/submission.html*

A.1 Abstract

This artifact allows others to reproduce the results seen in this paper for MakeCode and CODAL, using the BBC micro:bit. The artifact contains an offline build environment for CODAL and MakeCode, allowing evaluators to test and build programs locally. In addition, we also provide espruino and micropython virtual machines to further increase repeatability of our results. Evaluators should download the virtual machine containing all pre-requisite tools, and use an oscilloscope to observe wave forms (used for timing) generated by the micro:bit, and a serial terminal to observe results reported from the micro:bit over serial.

A.2 Artifact check-list (meta-information)

- **Program:** MakeCode & CODAL
- **Compilation:** arm-none-eabi-gcc
- **Binary:** espruino, and micropython binaries included; others compiled during testing
- **Run-time environment:** CODAL
- **Hardware:** BBC micro:bit
- **Output:** Waveforms, and serial output
- **Publicly available?:** Yes
- **Artifacts publicly available?:** Yes
- **Artifacts functional?:** Yes
- **Artifacts reusable?:** Yes
- **Results validated?:** Yes

A.3 Description

A.3.1 How delivered

The artifact is available hosted on GitHub:

https://lancaster-university.github.io/lctes-artefact-evaluation/

Alternately, the latest release is available for download:

https://doi.org/10.5281/zenodo.1242627

Finally, a virtual machine, based on debian, containing all the required software to reproduce our results is available here:

https://doi.org/10.5281/zenodo.1242605

A.3.2 Hardware dependencies

- A BBC micro:bit
- An oscilloscope
- A computer capable of running a virtual machine

A.3.3 Software dependencies

- A virtual machine obtained from the URL above.
- A serial terminal.

A.4 Installation

Use virtual box to install the image located at:

https://drive.google.com/open?id=1nxiorz6NRqjen89G59RCOEMklqAyaUv7

and the VirtualBox extension pack:

https://www.virtualbox.org/wiki/Downloads

A.5 Experiment workflow

Tests generally follow the following sequence of steps:

1. Perform small program modifications.
2. Compile the program.
3. Transfer program to the micro:bit (flashing).
4. Observe either a waveform generated by the micro:bit using an oscilloscope, or serial output from the micro:bit using a serial program.

A.6 Evaluation and expected result

We expect the results to be the same as those reported in the paper. The observed waveforms may differ in time due to different compilers, oscilloscopes, and oscilloscope calibration.

A.7 Experiment customization

All tests provided have a clear set of corresponding instructions that evaluators should follow to observe the same results. Any steps involving customisation have been minimised.

A.8 Notes

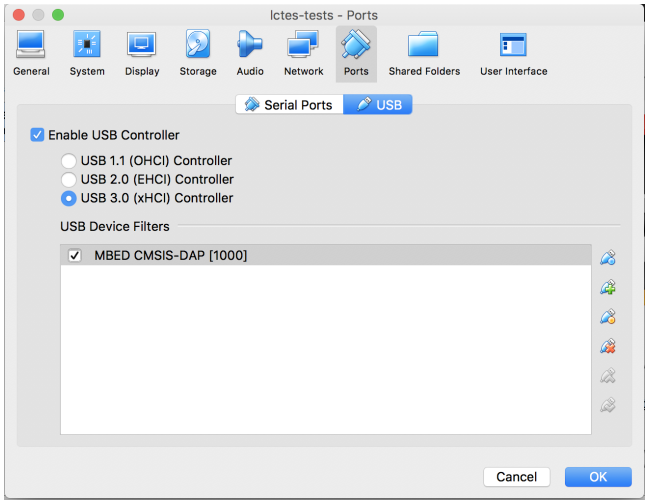
The virtual machine contains a folder named 'evaluators' which is placed in the home directory of the lctes user. The username for the virtual machine is: *lctes* and the password is: *lctes2018*. To become super user, type *su* in a terminal, and enter the same password (*lctes2018*).

Once logged in, and in the 'evaluators' directory, you can view the tests as markdown files in the 'docs' directory. Alternately, these markdown documents can also be viewed on the web by running 'mkdocs serve' in the evaluators folder, or browsing to:

https://lancaster-university.github.io/lctes-artefact-evaluation/

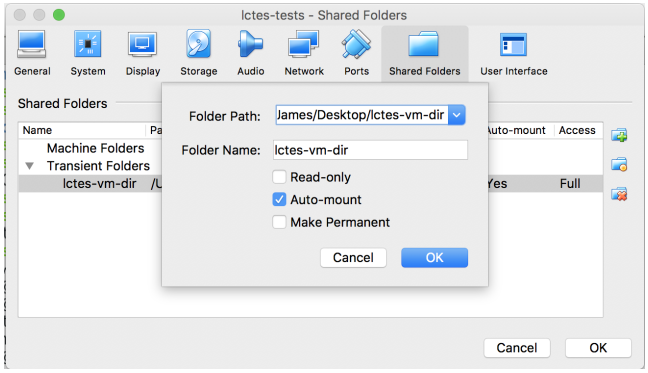
Which is a pre-built, and hosted version produced from the same source.

We recommend that you add the micro:bit usb device using the machine settings tab in virtual box as shown in the image below:



We also have a convenience script for mounting a shared folder between the host and the vm. Simply create a shared folder named

'lctes-vm-dir' and run 'sh mount.sh' (contained in evaluators) as a super user to mount the shared folder to vb-share (also contained in evaluators). Shared folder creation in VirtualBox is pictured below:



References

[1] 2010. AVRDUDE - AVR Downloader/UploaDEr. https://www.nongnu.org/avrdude/. (2010). (Accessed on 02/22/2018).

[2] 2012. Ardublock | A Graphical Programming Language for Arduino. http://blog.ardublock.com/. (2012). (Accessed on 02/22/2018).

[3] 2013. GitHub - ARMmbed/DAPLink. https://github.com/ARMmbed/DAPLink. (2013). (Accessed on 02/22/2018).

[4] 2015. Apache Mynewt. https://mynewt.apache.org/. (2015). (Accessed on 11/16/2017).

[5] 2017. Home - Zephyr Project. https://www.zephyrproject.org/. (2017). (Accessed on 11/16/2017).

[6] ARM. 2017. The Arm Mbed IoT Device Platform. (2017). https://www.mbed.com/

[7] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. IEEE, 79–80.

[8] Paulo Blikstein. 2013. Gears of our childhood: constructionist toolkits, robotics, and physical computing, past and future. In *Proceedings of the 12th International Conference on Interaction Design and Children*. ACM, 173–182.

[9] Rebecca F Bruce, J Dean Brock, and Susan L Reiser. 2015. Make space for the Pi. In *SoutheastCon 2015*. IEEE, 1–6.

[10] Dale Dougherty. 2012. The maker movement. *innovations* 7, 3 (2012), 11–14.

[11] A Dunkels, R Quattlebaum, F Österlind, G Oikonomou, M Alvira, N Tsiftes, and O Schmidt. 2012. Contiki: The open source OS for the Internet of things. *Retrieved October* 13 (2012), 2015.

[12] Damien George. 2017. MicroPython. (2017). http://micropython.org/

[13] Intel. 1988. Hexadecimal Object File Format Specification. (1988). https://web.archive.org/web/20160607224738/http://microsym.com/editor/assets/intelhex.pdf

[14] Joel Koshy and Raju Pandey. 2005. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*. ACM, 243–254.

[15] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. *Ambient intelligence* 35 (2005), 115–148.

[16] Mirjana Maksimović, Vladimir Vujović, Nikola Davidović, Vladimir Milošević, and Branko Perišić. 2014. Raspberry Pi as Internet of things hardware: performances and constraints. *design issues* 3 (2014), 8.

[17] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 16.

[18] John H Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. 2008. *Programming by choice: urban youth learning programming with scratch*. Vol. 40. ACM.

[19] The micro:bit Educational Foundation. 2016. micro:bit : Blast off: School launches BBC micro:bit into space! https://www.microbit.co.uk/rishworth-space. (2016). (Accessed on 05/13/2017).

[20] The micro:bit Educational Foundation. 2016. micro:bit : Use the BBC micro:bit in the BLOODHOUND Rocket Car contest. https://www.microbit.co.uk/bloodhound-rocket-car. (2016). (Accessed on 05/15/2017).

[21] Microsoft. 2016. 20160705_microsoft_bloodhound_0409 - Microsoft News Centre UK. https://news.microsoft.com/en-gb/2016/07/07/26785/20160705_microsoft_bloodhound_0409. (2016). (Accessed on 05/15/2017).

[22] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67. http://doi.acm.org/10.1145/1592761.1592779

[23] 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

[24] Charles R. Severance. 2014. Massimo Banzi: Building Arduino. *IEEE Computer* 47, 1 (2014), 11–12. https://doi.org/10.1109/MC.2014.19

[25] Vincent St-Amour and Marc Feeley. 2009. PICOBIT: a compact scheme system for microcontrollers. In *International Symposium on Implementation and Application of Functional Languages*. Springer, 1–17.

[26] Franklyn Turbak, Mark Sherman, Fred Martin, David Wolber, and Shaileen Crawford Pokress. 2014. Events-first programming in APP inventor. *Journal of Computing Sciences in Colleges* 29, 6 (2014), 81–89.

[27] Ankush Varma and Shuvra S Bhattacharyya. 2004. Java-through-C compilation: An enabling technology for java in embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, Vol. 3. IEEE, 161–166.

[28] Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. 2015. Programming Microcontrollers in Ocaml: the OCaPIC Project. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 132–148.

[29] Gordon Williams. 2017. *Making Things Smart: Easy Embedded JavaScript Programming for Making Everyday Objects into Intelligent Machines*. Maker Media.