Simplifying the Programming of Microcontroller-based Devices

Microsoft MakeCode and Lancester University Teams

Anonymous Author(s)

Abstract

Text of abstract ....

Keywords keyword1, keyword2, keyword3

1 Introduction

Microcontrollers, traditionally the workhorses of embedded systems, have become central to efforts in making and education. For example, the Arduino project [**?**], started in 2003, created a printed circuit board (the Uno) based on the 8-bit Atmel AVR microcontroller unit that makes most of the its I/O pins available via headers on the board. Hardware modules (shields) may be connected to the main board to extend its capability. The Arduino ecosystem, based on an open hardware design, has grown tremendously in the past 15 years, with the support of companies such as Adafruit Industries and Sparkfun Electronics, to name a few.

What has not changed much in this time is the way these boards are programmed. The C and C++ programming languages are the primary way to program microcontrollers. This is not a huge surprise, given the low-level nature of microcontroller programming, where direct access to the hardware is the order of the day. There generally is no operating system running on such boards, as they have very little RAM (2K for the Uno, for example) and lack memory protection hardware. What is more surprising about the Arduino platform is that:

- it encourages the use of polling by the end-user as the primary way to interact with sensors, which leads to monolithic sequential programs;
- its IDE lacks any code "intellisense" or common interactive features of modern IDEs;
- it loads code onto the microcontroller using 1980s era bootloader technology.

As a result, it is not simple to get started with systems based on Arduino, of which there are many. On the other hand, on the web we find many excellent environments for introducing programming to beginners. Visual block-based editors such as Scratch [**?**] and Blockly [**?**] allow the creation of programs without the possibility of syntax errors. HTML and JavaScript allow a complete programming experience to be delivered as an interactive web application, including editing with intellisense, code execution and debugging. (While the Arduino IDE recently has been ported to the web, it lacks many of the above features and requires a web connection to a server which runs a C/C++ compile tool chain to compile user code.) The programming models associated with these environments are generally event-based, freeing the user from the tyranny of polling.

We present a new programming platform that bridges the gap between the worlds of the microcontroller and the web app. The major goals of the platform are to:

- make it simple to program microcontrollers using an interactive web app that works when offline;
- allow a user's compiled program to be easily installed on a microcontroller;
- support the addition of new of software/hardware components to a microcontroller.

The platform, *MCCU*, consists of three major components (*MakeCode*, *CODAL* and *UF2*), which we now describe. The MakeCode web app (see www.makecode.com) supports both visual block-based programming and text-based programming using TypeScript, a gradually-typed superset of JavaScript, with the ability to convert between the two representations. The web app supports in-browser execution, via a device simulator, and compilation to machine code, linking against the pre-compiled CODAL C++ runtime to produce a binary for execution on an microcontroller (either 8-bit AVR and 16-bit ARM Thumb instruction set). No C/C++ compiler is invoked for a compilation of user code. The result of compilation is a binary file that is "downloaded" from the web app to the user's computer. The USB flashing format (UF2) makes copying of the binary file to the device, mounted as removable flash drive, fast and reliable, across all major operating systems. Once the web app has been loaded, all the above functionality works offline (i.e., if the host machine loses its connection to the internet).

The main innovations of MCCU are:

- The design and implementation of *MakeCode*, which bridges the worlds of JavaScript and C++, enabling beginners to get started programming microcontrollers from any modern web browser and enabling hardware vendors to innovate and safely add new components to the mix.
- *Static TypeScript*, a statically-typed subset of TypeScript for fast execution on low-memory devices and

PL'17, January 01–03, 2017, New York, NY, USA
2017.

a simple model for linking against pre-compiled C++; Static TypeScript also can be used to write safe and performant device driver code.

- *CODAL*, the Component-oriented Device Abstraction Layer, maps each hardware component to one or more software components that communicate over a message bus and schedule event handlers to run non-preemptively on fibers.
- The USB Flashing Format (*UF2*), a file format designed for flashing microcontrollers over the Mass Storage Class (removable USB pen drive) protocol. This new file format greatly speeds the installation of user programs and is robust to difference in operating systems.

MCCU combines these innovations in programming languages, language runtime, and code loading to make a simple programming experience for the end user. Through its support for Static TypeScript and a foreign function interface to C++, MCCU makes it easy for hardware manufacturers to share their C++ components with a wider audience. All of MCCU's components are open source under the MIT license, as detailed below.

MCCU targets can be seen at www.makecode.com, where the MakeCode web app for a variety of boards is available, including the micro:bit (a Nordic nRF51822 microcontroller with Cortex-M0 processor, 16K RAM), Adafruit's Circuit Playground Express (CPX: an Atmel SAMD21 microcontroller with Cortex-M0 processor, 16K RAM), and the Arduino Uno (Uno: an Atmel ATmega328 microcontroller with AVR processor, 2K RAM).

We encourage the reader to choose a board and experiment with programming it, using the simulator to explore many of each board's features, to appreciate the qualitative aspects of MCCU: its simplicity and ease of use. In this paper, we will evaluate quantitative aspects of MCCU: compilation speed, code size, and runtime performance. In particular, we evaluate:

- the compile time of Static TypeScript compile/link of user code (to machine code) with respect to the GCC-based C/C++ toolchain, as well as the size of the resulting executable;
- the time to load code onto a microcontroller using UF2, compared to standard bootloaders;
- The performance of a set of small benchmarks, written in both Static TypeScript and C++, compiled with the MakeCode and GCC-based toolchains, as well as the performance of device drivers written in Static TypeScript compared to their C++ counterparts.

[evaluate with respect to the popular Arduino toolset, for boards with 8-bit (AVR) and 32-bit (Cortex-M0) microcontrollers. Summary of evaluation]

MCCU is open source on GitHub. The MakeCode framework is at https://github.com/microsoft/pxt. (PXT is previous codename of MakeCode). MakeCode targets for the three previously mentioned boards are at microsoft/pxt-microbit, microsoft/pxt-adafruit, and microsoft/pxt-arduino-uno. The latter two targets make use of a common set of MakeCode libraries (packages) at microsoft/pxt-common-packages, Many other MakeCode packages, developed by Microsoft and hardware partners *[details later]. A few examples: XYZ.*

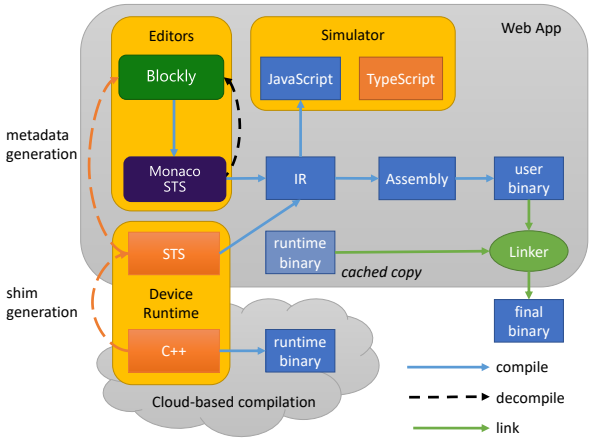

Figure 1. MakeCode Architecture

The rest of this paper is organized as follows. Section 2 presents the design and implementation of the MakeCode framework. Section 3 describes Static TypeScript and section 4 presents the CODAL C++ runtime. Section 5: USB Flashing Format; Section 6: Evaluation; Section 7: Related Work; Section 8: Conclusion and Future Directions.

2 MakeCode: Design and Implementation

The MakeCode web app encapsulates all the components needed to deliver a programming experience for microcontroller-based devices, free of the need for a C++ compiler for the compilation of user code, as shown in Figure 1. The web app is written in TypeScript. The web app also incorporates the TypeScript compiler and language service (www.typescriptlang.org) which are used in several ways, as detailed below. The remaining subsections describe the parts of Figure 1.

2.1 Device Runtime and Shim Generation

A MakeCode target is defined, in part, by its device runtime, which can be a combination of C++ and Static TypeScript (STS) code. The C++ runtime for the target microcontroller is precompiled and stored in the cloud. The C++ runtime binary is cached in the HTML5 application cache (with other assets) so that the web app can function when the browser is offline. Additional runtime components may be authored in STS, which allows the device runtime to be updated without the need for any C++ programming, and permits components of the device runtime to be shared by both the device and simulator runtimes. We will describe later how elements of

the C++ runtime are exposed to STS (via automated shim generation). 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 (for runtime components written in TypeScript) or declarations (for runtime components written in C++). A full-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 code.

MakeCode supports a simple foreign function interface from TypeScript to C++ based on namespaces, enumerations, functions, and basic type mappings. MakeCode uses top-level namespaces (in both C++ and TypeScript) to organize sets of related functions. Preceding a C++ namespace, enumeration, or function with *//%* indicates that MakeCode should map the C++ construct to TypeScript. Within the *//%* comment, attributes are used to define the visual appearance for that language construct, such as for the led namespace in the micro:bit target:

```
//% color=#5C2D91 weight=97 icon="\uf205"
namespace led {
...
```

Here is the C++ file defining the micro:bit's led namespace and its functions:

https://github.com/Microsoft/pxt-microbit/blob/master/libs/core/led.cpp

Mapping of functions and enumerations between C++ and TypeScript is straightforward. Here's an example of the C++ function plot in the led namespace that wraps a more complex function call of the underlying device runtime to set/plot an LED in the micro:bit display:

```
//% blockId=device_plot block="plot|x %x|y %y"
//% x.min=0 x.max=4 y.min=0 y.max=4
void plot(int x, int y) {
    uBit.display.image.setPixelValue(x, y, 1);
}
```

We'll describe the attribute definitions in the *//%* comment in the next section. MakeCode uses a TypeScript declaration file to describe the TypeScript elements corresponding to C++ namespaces, enumerations and functions. We call such files shim files. Since the C++ plot function is preceded by a *//%* comment, MakeCode adds the following TypeScript declaration to the shim file (shims.d.ts) and copies over the attribute definitions in the comment. MakeCode also adds an attribute definition mapping the TypeScript shim to its C++ function (shim=led::plot):

```
//% blockId=device_plot block="plot|x %x|y %y"
//% x.min=0 x.max=4 y.min=0 y.max=4 shim=led::plot
function plot(x: number, y: number): void;
```

Here is the shim file generated from the annotated C++ sources for the micro:bit device runtime:

https://github.com/Microsoft/pxt-microbit/blob/master/libs/core/shims.d.ts

To support the foreign function interface, MakeCode defines a mapping between C++ and TypeScript types. Boolean and void have straightforward mappings from C++ to TypeScript (bool âĘŠ boolean, void âĘŠ void). As JavaScript only supports number, which is a C++ float/double, MakeCode uses TypeScript's support for type aliases to name the various C++ integer types commonly used for microcontroller programming (int → int32, unsigned → uint32, short → int16, byte → uint8, sbyte → int8). This is particularly useful for saving space on 8-bit architectures such as the AVR.

MakeCode supports both untagged and tagged integers (described more later). Under the untagged strategy, a JavaScript number is interpreted as a C++ int by default, while the tagged strategy allows both interpretation as integer and double (with an optimization to convert from integer to double, as required). MakeCode includes reference counted C++ types for strings (StringData*) and parameterless lambdas (Action).

These C++ types are mapped to the TypeScript types string and ()=>void, respectively. MakeCode allows a set of C++ functions with the same first parameter (of type Foo) to be exposed as a TypeScript interface Foo as follows: this set of C++ functions must be grouped inside a namespace of the name FooMethods. See, for example, how a C++ buffer abstraction is exposed:

https://github.com/Microsoft/pxt-microbit/blob/master/libs/core/buffer.cpp

You can find the resulting TypeScript Buffer interface in the shim file for the micro:bit (already referenced above).

2.2 Block Metadata Generation

Both C++ and TypeScript APIs can be specially annotated (minimally via *//% block*) so that the MakeCode compiler generates the needed Blockly metadata to expose an API as a visual block. So, to expose the previously encountered plot function as a visual block (as well as a TypeScript function), one simply needs:

```
//% block
void plot(int x, int y) { . . . }
```

Additional attribute definitions can provide text descriptions for the block, project function parameters (thus simplifying the API available via Blockly), and describe other visual/functional characteristics of the block. MakeCode uses the types of function parameters to select appropriate Blockly widgets. For example, an enumeration is represented by a dropdown menu in blocks. For more information on the block-specific annotations, see https://makecode.com/defining-blocks. MakeCode's support for Blockly means that

for the common case, the target developer doesn't need to know anything about the Blockly framework. For more sophisticated needs, one can directly access the Blockly framework.

2.3 Editors and Code Conversion

MakeCode uses the Blockly (https://github.com/google/blockly) and Monaco (https://github.com/Microsoft/monaco-editor) editors to allow the user to code with visual blocks or TypeScript. The editing experience is parameterized by the full-typed device runtime, which provides a set of categorized APIs to the end-user, based on namespaces, as previously described. These APIs are visible in both editors via a toolbox to the immediate left of the programming area. The Blockly and Monaco toolboxes show the same set of APIs, to help in transition from coding with blocks to coding with JavaScript. More advanced TypeScript APIs can be discovered in Monaco via code intellisense.

The Blockly program representation is compiled to Static TypeScript in a syntax-directed manner (see https://github.com/Microsoft/pxt/tree/master/pxtblocks). A key issue is the need for type inference on the Blockly representation, as variables generally are defined and used without being declared in Blockly. MakeCode uses a simple unification-based type inference to assign a unique type to each variable. In the future, we expect to use TypeScript's type inference instead and eliminate the need for separate type inference over the Blockly representation. TypeScript supports programming constructs that are not available in Blockly, such as classes. Such constructs are converted into grey uneditable blocks in Blockly, with the construct's program text intact. This means MakeCode always can decompile a TypeScript program to Blockly and then recover the program text of the grey blocks when converting from Blockly back to TypeScript (see https://github.com/Microsoft/pxt/blob/master/pxtcompiler/emitter/decompiler.ts).

2.4 Compilation Pipeline

MakeCode first invokes the TypeScript language service to perform type inference and type checking on the user's program, using the TypeScript declaration files for the device runtime. It then checks that the user's program is within the STS subset through additional syntactic and type checks over the adorned abstract syntax tree (AST) produced by the language service (detailed in Section XYZ). Assuming all the above checks pass, MakeCode then performs tree shaking and compilation of the AST of the user code and device runtime 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 two backends for code generation from the IR. The first backend simply generates JavaScript, for execution against the simulator runtime. The other backend generates assembler, parameterized by a processor description. 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. Finally, a linker completes the compile chain by resolving references to the driver runtime in the user's program, producing a binary executable. The compiler chain can be found at https://github.com/Microsoft/pxt/tree/master/pxtcompiler/emitter and https://github.com/Microsoft/pxt/tree/master/pxtlib/emitter.

2.4.1 Asynchronous Functions

A key part of the compilation process is to allow users to call async functions (identified through the *//% async* annotation) as if they were regular (blocking) functions. This is done by compiling each function so that it can be suspended (at the return of a call) and later resumed (at the same point). The default behavior at a suspension point is to immediately resume execution. If a call is to an async function then the default behavior is overridden by the compiler, which suspends execution of the current function. Upon completion of the async function call, the current function then is resumed. This feature greatly simplifies the JavaScript-based programming model which relies on promises to achieve asynchronous execution, breaking up sequential (blocking) code into a series of event handlers. The JavaScript-based simulator runtime uses promises to achieve asynchronous execution in a single-threaded context, but these promises are hidden from the end user. The CODAL C++ device runtime supports fibers with the ability to pause, so for compilation to a device, the compiler simply emits a call to pause at a suspension point.

2.4.2 Untagged and Tagged Strategies

The MakeCode compiler supports the Static TypeScript language subset described in Section X, with two compilation strategies: untagged and tagged. *TODO* Untagged compilation strategy No support for doubles. 32-bit integers (signed only). Use static type system to distinguish between integers and references. No runtime support. Not fully compatible with JavaScript semantics. Null = Undefined = 0 = default integer value. Support for different integer sizes using type aliases (this is used for AVR). Tagged compilation strategy In the tagged strategy, numbers are either tagged 31-bit signed integers, or if they do not fit, boxed doubles. Code generation to check tag bit. Special constants like false, null and undefined are given special values and can be distinguished. Fully compatible with JavaScript semantics (number, null, undefined). Can accommodate a richer type system.

2.5 Simulator

A MakeCode target can provide an alternate TypeScript implementation for each API in the device runtime, for use in

the device simulator. As this code runs in the web browser (not on the actual device) and manipulates the DOM, the developer is free to use all of TypeScript/JavaScript's features. (As an aside, MakeCode also support "simulator-only" targets that have no associated device; in these cases, the "device runtime" is defined solely by the simulator APIs.) The simulator allows the user to experience the basic functions of the device in the browser and to test their code before deploying it to the actual device. The simulator has proxy widgets for sensors such as accelerometer (mouse motion), temperature and light, allowing the user to control the sensor's value. The simulator only provides basic functionality and is far from a complete device emulation. For example, it is not possible for the user to simultaneously modify two inputs to the simulated device, while it is possible with the actual device (i.e., shaking it to change the accelerometer reading while pushing one of the device's buttons).

MakeCode provides various components to make device-based simulators easier to build: board, parts, wiring, etc.

2.6 Packages and Custom Editors

Packages are MakeCode's dynamic/static library mechanism for extending a target (by adding new code/data to the device and simulator runtimes, as well as accompanying documentation). The following package extends the micro:bit target so that the micro:bit can drive a NeoPixel strip of RGB LEDs: https://github.com/Microsoft/pxt-neopixel. To see how this package is surfaced to the end-user, visit http://makecode.microbit.org/ and select the "Add Package" option from the gear menu; you will see the package "neopixel" listed in the available options. If you click on it, a new block category named "Neopixel" will be added to the editor. In this scenario, PXT dynamically loads the (white listed) neopixel package directly from GitHub, compiles it and incorporates it into the web app. Packages also can be bundled with a web app (the analog of static linking).

Hardware partners already have started to create MakeCode packages for the micro:bit. Seeed Studio (https://www.seeedstudio.com/) has created packages to add its Grove components to a micro:bit. Grove components are accessed via the I2C serial protocol, supported by the micro:bit device runtime. All micro:bit packages for the Grove components are authored in Static TypeScript (gesture, ultrasonic-ranger, 4-digital-display, two-led-matrix). These packages can be found under GitHub user "Tinkertanker", prefixed with "pxt-". Sparkfun has created MakeCode packages for its micro:bit shields (GitHub user "sparkfun").

3 Static TypeScript

TypeScript is a typed superset of JavaScript designed to enable JavaScript developers to take advantage of code intellisense, static checking and refactoring made possible by types []. You can edit and run simple TypeScript programs at http://www.typescriptlang.org/play. As a starting point, every JavaScript program is a TypeScript program. Types can be added gradually to such programs, supported by type inference. In TypeScript, the Any type represents any JavaScript value with no constraints. Type inference may assign the Any type to expressions for which no more specific type can be inferred.

In TypeScript, object types are used to describe dictionaries, functions, arrays, as well as class instances. Object types also describe objects that take on multiple of the above roles, as is common in JavaScript. Object types are related to one another by structural subtyping []. Interface declarations name object types; classes add implementation and the ability to statically inherit implementation from super classes. TypeScript also supports generics, as well as union and intersection types.

While TypeScript provides programming abstractions (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 JavaScript objects with all their dynamic semantics intact. This is not surprising, as TypeScript was designed to accommodate the dynamic nature of JavaScript and programming patterns familiar to the JavaScript programmer.

TypeScript has enough types to allow us to approach it from the viewpoint of the microcontroller programmer, who is familiar with the static (though unsound) type systems of C and C++. Our realization is that TypeScript contains a statically-typed sound subset (Static TypeScript: STS, for short) that closely resembles Java and C# in its semantics. STS arises from TypeScript by:

- *eliminating* the Any type from the type system, as well as JavaScript constructs that only can be typed with Any;
- *partitioning* the space of object types into functions, records, constructor functions, and arrays, with no casts possible between the partitions (and eliminating all other object types) - structural subtyping is used to relate records to each other, as in TypeScript;
- typing of classes *nominally* rather than structurally, as in TypeScript, and using traditional function (contravariant in the argument type, covariant in the return type) subtyping for functions and methods;
- *restricting* casts between records and classes: a class can be cast to a record, but a record cannot be cast to a class.

STS is a syntactic subset of TypeScript âĂŞ there is no change to the syntax of TypeScript. *[TODO: also, we disllow update of method property]* In terms of type checking, STS also is a subset of TypeScript. Simply put, given a STS program P, for every type compatibility or subtype relation R(P) of TypeScript, the corresponding relation RâĂŹ(P) in STS is

a subset of relation R. Every STS program is a TypeScript program. *[TODO: semantics preserved by erasure for programs written entirely in STS]* STS guarantees the absence of runtime type errors, including downcasts. The major class of runtime errors still possible is dereference of null/undefined values. STS can be supported by a simple runtime model, like that for C++, using classic vtables and interface tables, rather than the prototype-based runtime model used for full TypeScript/JavaScript.

[TODO: rest of section âĂŞ type system details, compiler representation]

3.1 Eliminating Types: Any, Union, Intersection

By default, the TypeScript compiler assigns the Any type to an expression/declaration for which it is unable to infer a more precise type. STS uses the *–noImplicitAny* option to direct the TypeScript compiler to raise an error whenever it makes an (implicit) assignment of the Any type. STS also checks for an explicit use of the Any type in the program and raises an error. Many JavaScript constructs can only be typed with the Any type, including: prototype lookup, the eval statement, the with statement, a this reference outside of class context, an index access on non-array object, and reflection on Function/Object objects. Therefore, these constructs are not in STS. STS also excludes TypeScriptâĂŹs union and intersection types.

3.2 Partitioning of Object Types

Key to TypeScript are the definitions of object types and interfaces. Per the TypeScript specification [**?**]:

- "object types are composed from properties, call signatures, construct signatures, and index signatures, collectively called members";
- "interfaces provide the ability to name and parameterize object types and to compose existing named object types into new ones.".

Object types and the interface declarations that describe them enable the typing of a JavaScript object that plays multiple roles (as a function, dictionary, etc.). For example, here is an interface that describes an object that is a function from number to number and also has property (foo) which is a string:

```
interface Bar {
    (x: number): number; // call signature member
    foo: string;         // property member
}
```

STS partitions the space of object types as follows:

1. a *record* type has only member properties and optionally, a string index signature;
2. a *function* type has exactly one member, a call signature;
3. a *constructor function* type: has at least one constructor signature and no other signature kind;
4. an *array* type has a numeric index signature and no other signatures;
5. an *other* type: object type not covered by the previous four categories.

STS excludes all "other" types, leaving us with a set of classic types XYZ.

[Interfaces describe all of the above, help to access class and record uniformly, as they support arbitrary subsetting of properties.

3.3 Nominal Typing of Classes

TypeScript introduces two (object) types for each class declaration: a constructor function type by which instances of a class are created (using new) and static properties accessed; a class type describing the instances of the class (records, as defined above). [but the class type contains a link back to the class declaration; STS treatment of classes using a nominal interpretation, names and extends for subtyping]

4 The CODAL Runtime

5 USB Flashing Format

6 Evaluation

7 Related Work

8 Conclusion

A Appendix

Text of appendix ...