Programming of Microcontroller-based Devices: Anyone, Anywhere, Anything

Anonymous Author(s)

Abstract

Microcontrollers, the low-power low-cost workhorses of embedded systems, form the majority of programmable devices in the world. Traditionally, because microcontrollers are so resource constrained in terms of processing speed, memory and power consumption, they are programmed in low-level languages like C and C++. The programming tools are often complex to set up, learn and use, and specific to individual microcontrollers. As a result, microcontroller programming requires very particular skills.

We present a new programming platform that makes it easier for *anyone* to program microcontrollers. We aim to leverage today's lightweight tooling to allow code to be developed *anywhere*, requiring only a computer with a modern web browser and a USB port. Finally, we describe how our platform has been architected to bridge between the Web and relatively impoverished microcontrollers, in accordance with our aspiration to program (*anything*) in a more accessible manner.

With the predicted growth of the Internet of Things we believe that a system such as ours, which enables both novices and developers from diverse backgrounds to develop microcontroller code quickly, will become increasingly valuable. After describing our platform in detail, we evaluate performance on a range of modern low-end microcontrollers, before ending with a review of related work.

1 Introduction

Embedded devices are growing impressively in popularity and ubiquity in recent years. This growth can be attributed to the emergence of new application domains inspired by small, highly resource-limited programmable processors known as microcontroller units (or MCUs), which may have as little as a few kilobytes of RAM. It is now the case that less than 1 percent of the world's processors reside in general purpose computers such as desktop PCs, laptops and tablets - with MCUs picking up the remainder [8]. Furthermore, demand for MCUs continues to grow due to their use in the monitoring and control of a wide variety of systems, ranging from wearables, to home automation, industrial automation, and smart grids - a phenomenon broadly referred to as the Internet of Things (IoT).

MCUs are highly resource limited devices, but are not simply smaller versions of laptop and desktop machines. They are architecturally quite distinct. If we are to understand how to optimize programming languages and their implementation for such devices it is worth dwelling on the key characteristics of MCUs, namely their storage and CPU capabilities. Table 1 highlights the core capabilities of MCU-based devices vs. a typical PC.

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

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).

While such devices are highly resource constrained compared to modern PCs (about six orders of magnitude less on some metrics), a deeper analysis derives two further observations.

Firstly, *MCUs are processor rich*. Contrary to first impressions, these devices have a *proportionally* large amount of processing power. Consider the BBC micro:bit versus a typical PC. It has about 100 times less CPU power, but 10^6 times less RAM, and 10^6 time less storage. A language/runtime designed for MCUs should therefore not only seek to provide high code density and spatial efficiency, *but to actively trade off time for space*, where possible.

Secondly, *MCUs are accumulating more features on chip*. MCUs still follow Moore's law (the number of transistors doubles roughly every two years). However, this additional capacity is not typically invested in simply optimizing processing power. Instead, *independent peripherals* are integrated onto the same MCU as the CPU. Examples include integrated radio hardware such as Bluetooth/WiFi and audio inputs and outputs. This hardware is designed to run independently of the CPU, hence an effective language/runtime should directly support *an asynchronous interaction model*.

Furthermore, with this surge in the demand for MCUs across a wide spectrum of applications, there is a need to simplify the programming of such devices so that more people can participate in the creation of MCU-based systems. From rapid product innovation to making and education, we have seen a sustained growth in the volume and diversity of

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

people wishing to program MCU-based devices. These new application developers share a common characteristic - *they are typically not professional software developers* [9, 12, 18].

Programming languages and development environments for MCUs have not kept pace with advances in hardware and diversification of user domains. Elsewhere, we have long supported inexperienced developers through languages such as Java, C#, Python and JavaScript, and also through visual programming frameworks such as Blockly [14].

In the world of the MCU, the C/C++ languages remain the standard, and for good reasons: they provide a familiar imperative programming model, with compilers that produce highly efficient code, and enable low level access to hardware features when necessary. Popular examples include the Arduino project (www.arduino.cc) [23], started in 2003, and ARM's Mbed platform (www.mbed.org) [4] - both of which rely heavily on a C/C++ programming model. However, the limitations of using C/C++ as an application programming language for *inexperienced* developers are well understood [7].

We introduce a novel architecture for MCU-based devices, that is designed with three aspirational goals in mind: (1) it should be usable by *anyone*, not just trained professionals; (2) it should be available *anywhere*, requiring no specialized software (native applications or device drivers) to operate; (3) it should run efficiently on *anything*, from devices with as little as 2kB of memory.

This paper focuses on the design, implementation, and evaluation of this architecture using the devices listed in Table 1 as exemplars. We leave a detailed treatment about usability for future work. More specifically, we present and evaluate a platform that bridges the worlds of the web and the MCU through three novel technologies:

- *WA*, a web application containing an *in-browser compiler* that supports the simplified programming of MCUs via editors for visual blocks and textual JavaScript languages (Section 2);
- *Static TypeScript*, a statically-typed subset of TypeScript (www.typescriptlang.org) that we have defined for fast execution on low-memory devices, with a simple model for linking against pre-compiled C++ (Section 3);
- *RT* , a component-based, event-driven, fiber-based C++ runtime environment that bridges the semantic gap between higher-level languages (such as TypeScript) and the hardware, modelling each hardware component as a software component (Section 4);

Our results (Section 5) show that these technologies combined can enable simplified programming through modern languages and event-based constructs 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++. Our platform was deployed about a year ago and sees daily use by thousands of users.

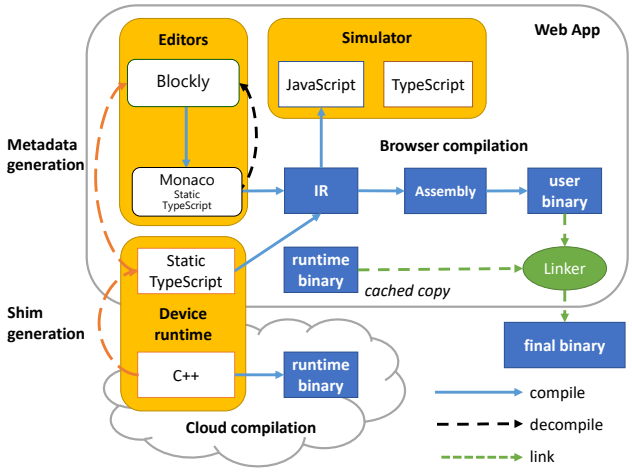

Figure 1. WA web app

1.1 Architectural Overview

Currently, due to the low-level nature of MCU programming, additional drivers and software are commonly required to be locally installed in order to program them. In restricted environments, like libraries and schools, safeguards are put in place to protect users and machines from downloading or installing additional software. Further, Internet connections are unreliable in remote locations. These factors create barriers for designing a programming environment that runs *anywhere*. Therefore, a solution is required that can: (1) operate in restricted environments; (2) operate with little, or no internet connectivity; and (3) work on any operating system.

To meet these requirements, we created the WA web app (Figure 1), the entry point of the platform. The web app can be accessed from any modern web browser and cached locally for *entirely offline use*. The WA web app incorporates the open source Blockly (*blockly*) and Monaco (*monaco-editor*) 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 Static TypeScript to machine code and linking against the C++ runtime (*RT*), pre-compiled (by a cloud service, lower left). The statically-typed runtime and type inference on user code allows users to write code that looks like plain JavaScript (see Figure 2).

WA devices appear as USB pen drives when plugged into a computer. After a user has finished developing a program, the compiled binary is "downloaded" locally to the users

computer and then transferred (flashed) to the MCU (exposed as USB pen drive) by a simple file copy operation. No additional installation is required to program the MCU as drivers for pen drive come pre-installed on many operating systems (MacOS, Windows, Linux, Android, ChromeOS).

These advances enable beginners to get started programming MCUs from any modern web browser, and offer a safe environment for hardware vendors to innovate and add new components using Static TypeScript. All of the platform's components are open source on GitHub.

1.2 Running Example: firefly simulation.

Figure 2 shows a JavaScript program that implements a simple "firefly" example for the Adafruit Circuit Playground Express (CPX), which has an onboard light sensor and 10 RGB LEDs (among other sensors and output devices).

This program demonstrates an emergent, time-based algorithm, run on multiple CPXs in a dark environment. Over time they will synchronize to a common clock using light pulses — akin to fireflies synchronizing their blinking in the wild.

At the top-level, the program has three statements: the first initializes the global variable *clock* (line 1); the second registers an event handler (a lambda function) to execute each time the CPX's light detector senses a bright light (line 3); the third registers a lambda function to run forever on a fiber (line 18), to keep track of time and pulse the CPX's 10 LEDs at regular intervals.

Note that although this program appears as a JavaScript program (there are no types mentioned explicitly), this program also is a Static TypeScript program: the static type of every variable and expression can be inferred and the program (with inferred types) passes the more restrictive type checks of Static TypeScript, as detailed in Section 3.

The program also demonstrates the use of the non-preemptive concurrency model supported by both WA (for JavaScript) and RT (for C++). The forever procedure executes the lambda inside a "while (true)" loop that yields (via a call to *loops.pause*) after each loop iteration. This gives the light-detection event handler a chance to execute upon user input (in a separate fiber). Although the global variable *clock* is shared by the two fibers, there is no data race due to the non-preemptive scheduling model.

We will refer to this example throughout.

2 WA: Design and Implementation

The key technical contribution of WA is to provide users with a safe environment to write programs for MCUs, enabling a simple progression from block-based programming, to text-based programming in Static TypeScript (STS), while leveraging C++ on the backend for efficient use of MCU resources.

```
1  let clock = 0
2
3  input.onLightConditionChanged(
4    LightCondition.Bright, () => {
5      if (clock < 8) {
6          clock += 1     // catchup to neighbor
7      }
8  })
9
10 loops.forever(() => {
11   if (clock >= 8) {
12       // notify neighbors
13       light.setAll(Colors.White)
14       loops.pause(200)
15       light.clear()
16       clock = 0          // reset the clock
17   } else {
18       loops.pause(100)
19       clock += 1
20   }
21 })
```

Figure 2. Running example: firefly simulation.

Figure 3 shows a screen snapshot of the WA web app for the CPX, which 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 toolbar provides access to device-specific APIs and programming elements; (D) the programming canvas (showing here the blocks corresponding to the JavaScript from Figure 2); (E) the download button invokes the compiler to produce 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 is built from a WA "target" which parameterizes the WA framework for a particular device. The remaining subsections describe the essential components of the web app from Figure 1.

2.1 Device runtime and Shim Generation

A WA target is defined, in part, by its device runtime, which can be a combination of C++ and STS code, as shown in the lower-left of Figure 1. All the target's C++ files are precompiled (by a C++ compiler in the cloud) into a binary, which is stored in the cloud in a single blob, 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 WA's design.

Whether runtime components are authored in C++ or STS, all runtime APIs are exposed as fully-typed TypeScript

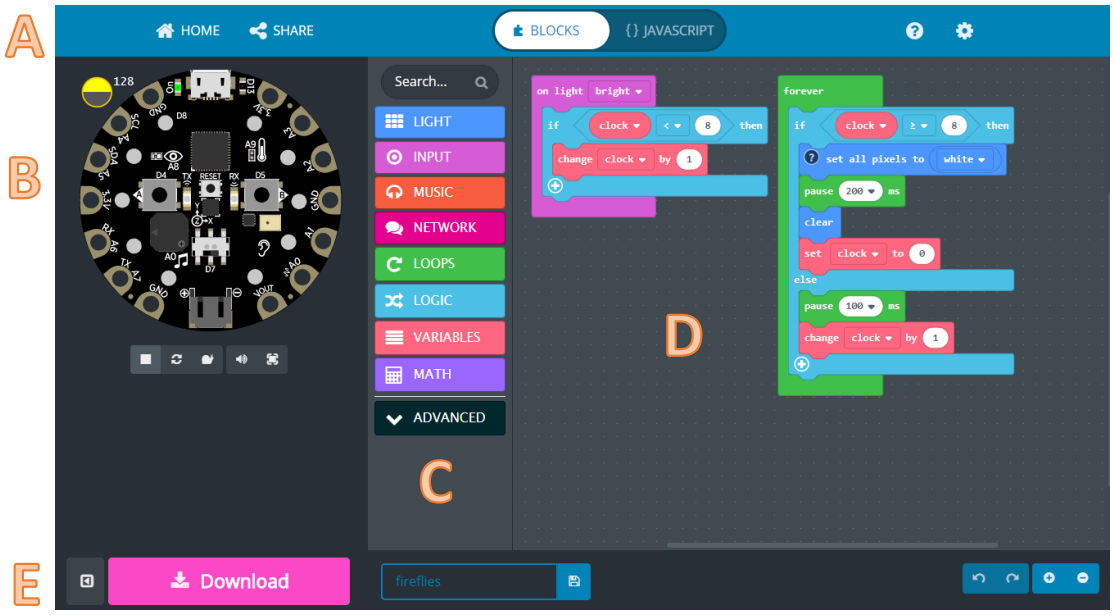

Figure 3. Screen snapshot of the WA web app, showing the Blockly view of the JavaScript from Figure 2.

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, as in Figure 2.

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

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

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

Mapping of functions and enumerations between C++ and STS is straightforward and performed automatically by WA . Here's an example of the C++ function *onLightConditionChanged* in the namespace *input* that wraps the more complex C++ needed to update the sensor and register the (Action) handler with the underlying RT runtime:

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

WA uses a 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, WA adds the following TypeScript declaration to the shim file and copies over the attribute definitions in the comment. WA also adds an attribute definition mapping the TypeScript shim to its C++ function:

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

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

To support the foreign function interface, WA 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++ float/double, WA 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. WA 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.

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

2.2 Metadata generation

Both C++ and TypeScript APIs can be specially annotated (minimally via *//% block*) so that the WA compiler generates the needed Blockly metadata to expose an API as a visual block. 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. WA uses the types of function parameters to select appropriate Blockly widgets. For example, given a C++ enumeration such as:

```
1 enum class LightCondition {
2     Dark = ANALOG_THRESHOLD_LOW,
3     Bright = ANALOG_THRESHOLD_HIGH
4 };
```

a parameter of type *LightCondition*, as found in the function *onLightConditionChanged*, will use a Blockly drop-down menu so the user can select one of the two possible values (Dark, Bright).

WA's support for Blockly means that for the common case, the WA 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

WA uses the Blockly and Monaco editors to allow the user to code with visual blocks or STS. 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 (Figure 3(C)) to the immediate left of the programming area.

WA compiles the Blockly program representation to STS in a syntax-directed manner. 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. WA uses a simple unification type inference to assign a unique type to each variable. STS 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 WA always can decompile an error-free STS program to Blockly and then recover the program text of the grey blocks when converting from Blockly back to STS.

2.4 Browser compilation

When a user requests a download of the compiled binary, WA first invokes the TypeScript language service to perform type inference and type checking on the user's program, the device runtime written in STS, as well as 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 (detailed in Section 3). Assuming all the above checks pass, WA then performs tree shaking of the AST to remove unused functions. The reduced AST then is 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 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 and resolves runtime references, producing a final binary executable. A third backend generates bytecode instructions. WA can encode the resulting binary in several formats, including Intel's HEX format [16] and the FF format (created by the authors for faster device flashing) documented in the Appendix (Section B).

2.4.1 Asynchronous Functions

An important part of the compilation process is to allow users to call asynchronous STS functions (identified through the *//% async* annotation) as if they were blocking functions. For example, in Figure 2, the runtime function *loops.pause(1000)* (called at line 10) is an async function that sets a countdown timer (for 1000 milliseconds) and then yields to the scheduler. When the timer reaches zero, the function continuation is queued for execution.

For execution in the single-threaded context of the JavaScript runtime, every function is compiled 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. For a call to an async function, 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. The RT device runtime supports fibers with the ability to yield, so for compilation to a device, the compiler simply emits a call to yield at a suspension point.

Async functions are written by runtime developers, not end-users, and greatly simplify the JavaScript programming model for end-users. For example, although the JavaScript simulator runtime uses promises to achieve asynchronous execution in a single-threaded context, these promises are hidden from the end user.

2.4.2 Untagged and Tagged Strategies

The WA compiler supports the STS language subset of TypeScript described in Section 3, 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 given special values and can be distinguished. The tagged execution strategy has the capability to fully support JavaScript semantics. This strategy is used for all SAMD21 targets, including the CPX.

2.5 Simulator

A WA target can provide an alternate 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 shown in Figure 3(B), 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 the light detector, temperature and accelerometer, allowing the user to control the sensor's value.

3 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 (http://www.typescriptlang.org/). 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.

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 JavaScript objects with all their dynamic semantics intact.

We approach TypeScript from the viewpoint of the MCU 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), 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 statements and expressions that only can be typed with Any;
- *partitioning* TypeScript's 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;
- typing of functions/methods via traditional function subtyping (contravariant in the argument type, covariant in the return type)
- *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 – there is no change to the syntax of TypeScript. 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 $R'(P)$ in STS is a subset of relation R. Every STS program is a TypeScript program.

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 JavaScript.

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* and *with* statements, 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 TypeScript's union and intersection types.

3.2 Partitioning of Object Types

In TypeScript, object types are used to describe dictionaries, functions, arrays, class instances, as well as 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. Per the TypeScript specification [19]:

- "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. 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;
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 the four kinds of types. The partitioning is further reflected in STS's subtype relation: types T and U from different partitions (in the first four partitions above) are not related by STS's subtype relation.

3.3 Treatment of Classes and Methods

TypeScript introduces two (object) types for each class declaration:

- a constructor function type by which instances of a class are created (and static properties accessed), as seen in the previous section;
- a class type describing the instances of the class, which is a record type.

TypeScript's treatment of class types as record types means that it is possible to assign to a property representing a method, destroying the class abstraction that ties together code and data. This also eliminates the possibility of using a shared vtable implementation for classes.

Instead, STS adds a new partition to represent class types, separate from record types, describing the instances of classes. In STS, a class type is a record type with an explicit association to its class declaration. Classic nominal subtyping is applied to class types, with function subtyping used to relate methods with the same name in subclass and superclass, per TypeScript, just as record types are treated. Furthermore, STS permits a cast from a class type to a record supertype, but does not allow any casts from record types to class types. This means it is possible to use interfaces to abstract over classes, as in Java and C#.

Finally, STS enforces that a function may not be assigned to a property of a class type.

3.4 Subtyping and Type Checking

Given the above, STS's subtype relation is unsurprising and defined in the appendix. STS type checking for (pure) expressions remains the same as in TypeScript. Type checking at assignments, where a value of type S flows (is assigned to) into a location of type T uses the new subtyping relation: S must be a subtype of T. Downcasts (a value of supertype flowing to where a subtype is expected) are not permitted in STS.

4 The RT Runtime

RT is a lightweight, object-oriented, componentized C++ runtime for microcontrollers designed to provide an efficient abstraction layer that is the target for higher level languages. RT is composed of 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;
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 presented as a C++ class;
5. *a parameterized object model* composed from these components that represents a physical device.

4.1 Message Bus and Events

RT 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 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 RT 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.

In our running example, a listen call to the MessageBus with a component ID of the light sensor and a threshold

```
#include "CircuitPlayground.h"
CircuitPlayground cplay;

void onBright() {
    // user defined code
}

int main() {
    cplay.messageBus.listen(ID_LIGHT_SENSOR,
        LIGHT_THRESHOLD_HIGH, onBright);
}
```

Figure 4. Example of the RT MessageBus.

event is the underlying mechanism to implement the runtime function *onLightConditionChanged* from Figure 2, as illustrated by the equivalent C++ code snippet in Figure 4.

Unlike simple function pointers, RT event handlers can be parameterized by the event listener to provide decoupling between the context of the code raising the event. The receiver of an event can choose to either receive an event in context of the fiber that created it, or be decoupled and executed via an Asynchronous Procedure Call (APC). The former provides performance, while the latter provides elegant 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.

4.2 Fiber Scheduler

RT provides a *non-preemptive* fiber scheduler with an elegant asynchronous semantics and power efficient implementation. RT 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 RT scheduler makes use of two novel mechanisms to optimize for MCU hardware. Firstly, RT 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 brings 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 RT 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 RT 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(). Any functions which do not block therefore do not incur a full 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 in then restored, and execution continues from the point at which the APC was first invoked. This mechanism provides potentially high RAM savings for processing of MessageBus event handlers in particular.

4.3 Memory Management

RT implements its own lightweight heap allocator, introducing reentrant versions of the libc malloc family of functions to permit 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.

RT 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, at the expense of suffering from known issues of circular references. We take the view that such scenarios are rare in MCU applications, making the overhead of more complex garbage collection difficult to justify.

4.4 Device Driver Components

RT 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. RT 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).

```
class CircuitPlayground : public RTDevice
{
  public:
    MessageBus                messageBus;
    CPlayTimer                timer;
    mbed::Serial              serial;
    CircuitPlaygroundIO       io;
    Button                    buttonA;
    mbed::I2C                 i2c;
    LIS3DH                    accelerometer;
    NonLinearAnalogSensor     thermometer;
    AnalogSensor              lightSensor;
    ...
};
```

Figure 5. Device Model for Adafruit CPX

This approach brings the benefits of abstraction and reusability to RT , but 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. This can be seen 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 provides for an elegant OO API for programming a device and also a static representation that forms an ideal target for the WA linker to bind high level STS interfaces to low level optimized code. An example device model for the Adafruit CPX is shown in Figure 5 for reference.

WAis further supported by an annotated C++ library (*WA wrappers*) defining and abstracting the mapping from RT to TypeScript. The use of WA wrappers ensures that different WAtargets that use RT share a common TypeScript and block API vocabulary.

5 Evaluation

Our platform has been actively deployed for over a year, bringing the benefits of a safe programming environment for MCUs to 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 insights into the performance of WA and RT across three devices (Uno, micro:bit, and CPX).

Throughout this section we break down results into the two basic layers (RT and WA) to give an insight into how each layer performs. (we refer to *WA wrappers* as WA throughout this evaluation):

- *RT* : the device runtime, against which we write C++ programs;
- *WA wrappers*: the C++ and STS wrapping RT to expose it to WA , against which we write STS programs.

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 WA and RT when running on a representative selection of real hardware devices. Throughout, we use the C++ RT benchmarks as a baseline; the STS benchmarks show the overhead added by WA .

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 SAMD21 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 Uno has no onboard components other than the microcontroller which supports GPIO, Serial, and I2C. The micro:bit has a 32-bit Cortex-M0 clocked at 16MHz, with 16kB RAM and 256kB of flash. It has numerous onboard components including a 5x5 LED matrix display, a compass, accelerometer, buttons, GPIO, I2C, Serial, and Bluetooth Low Energy. 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. On board the CPX are: RGB pixels, an infrared transmitter and receiver, a touch sensor, buttons, accelerometer, speaker, microphone, GPIO, I2C, and Serial.

The Uno and micro:bit WA targets use the untagged strategy, while the CPX target takes the tagged strategy (see Section 2.4.2).

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, 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 will log a map of memory before executing the operation, execute the operation, and log a map of memory at the end of the operation. A serial terminal captures the output of these tests.

It is important to 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 WA in context, we perform a comparative evaluation of WA against two state-of-the-art solutions, using native C++ as our baseline. The two points of comparison are MicroPython [15], an implementation of

	UNO	micro:bit	CPX
RT	171ms	102ms	31ms
WA	2.4x	2.1x	7.3x
WA VM	15.3x	-	-
MicroPython	-	98x	183x
Espruino	-	1133x	-

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

Python for microcontrollers, and Espruino [24], an implementation of JavaScript for micrcontrollers. For the CPX, a fork of MicroPython known as "CircuitPython" was used. Both MicroPython and Espruino uses virtual machine (VM) approaches.

To give an indicative general case execution time cost of each solution, we created a simple program that simply 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 only count to 25000, to fit in the 16 bit `int`, and scale up the results.

For the micro:bit, MicroPython and Espruino are *two or more orders of magnitude slower* than a native RT program. WA performs only 2x slower. The slowdown reflects the simple code generator of our STS compiler. It should be noted that WA 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 WA outperforms both the VM-based solutions of MicroPython and Espruino by at least an order of magnitude.

MicroPython and similar environment cannot run on Arduino Uno due to flash and RAM size limitations. We have also run into these, and as a result developed two compilation modes for AVR. One compiles STS to AVR machine code, and the other (WA 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 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 RT's scheduler we conducted a test that created two fibers, continuously swapped context, and measured the time taken to complete the context switch. We performed this test in both STS and C++ and the resulting

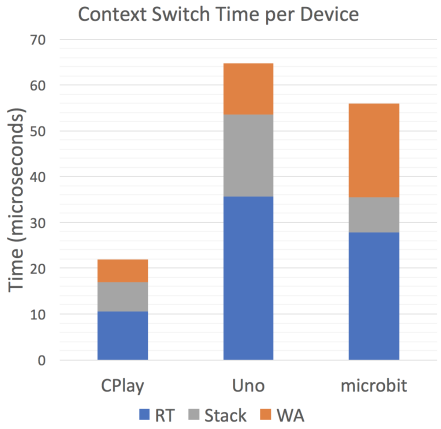

Figure 6. Base context switch profiles per device.

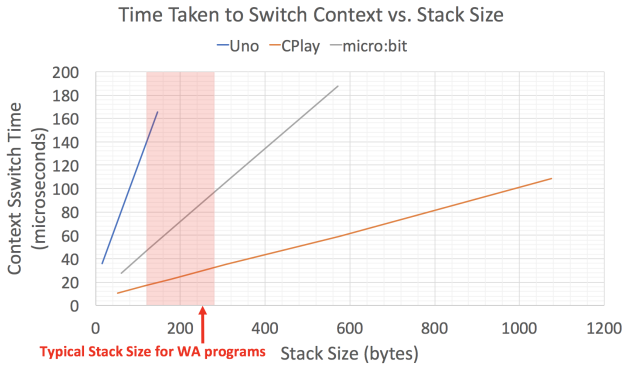

Figure 7. Time taken to perform a context switch against stack size.

profiles can be seen in Figure 6, which breaks the context switch down into three phases: (1) RT , the time it takes to perform a context switch in RT ; (2) Stack, the time taken to page out the WA stack; and (3) WA, the overhead added by the WA wrappers.

From these results, we observe that context switches generally take tens of microseconds. The cost of RT'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 7 therefore profiles the time a context switch takes with an increasing stack size across all three devices in RT . This test is similar to the previous test, however, 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 processor word size, which means more instructions are required to copy the stack, therefore as the stack size increases, so does context switch time. The vertical band indicates typical stack sizes for WA programs based on a representative set of examples.

	RAM Overhead (bytes)	Processing Overhead (microseconds)
Create a Fiber	136	35.4
APC	32	4.01
APC with Sleep	204	32.4

Table 3. RAM consumption and processing time for various asynchronous operations in RT .

	CPX	micro:bit	Uno
WA	20.46	12.14	7.79
RT	29.85	34.35	13.7
Mbed and Supporting Libraries	14.99	24.28	-
C++ Standard Library	43.14	24	1.03

Table 4. The total flash consumption of code required to support WA (kB).

5.4 Performance of Asynchronous Operations

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

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. For non-blocking calls, the best case, this has a small overhead of 32 bytes of RAM and is not processor intensive, versus the worst case, which incurs a large overhead of 204 bytes of RAM and 32.4 microseconds of processor time.

5.5 Flash Memory Usage

MCUs make use of internal non-volatile FLASH memory to store program code. Table 4 shows the per device flash consumption of each software library used in the final WA binary. To obtain these numbers, we profiled the final map file produced after compilation. The ordering of the table aligns with the composition of the software layer: WA builds on RT which builds on the C++ standard library and optionally, Mbed.

From the bottom up, the profile of the standard library changes dramatically for each device: The Uno has a very lightweight standard library; the microbit 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.

	CPX	micro:bit	Uno
WA	0.612	1.069	0.074
RT	0.369	0.214	0.156
mbed and Supporting Libraries	0.312	0.923	-
C++ Standard Library	0.161	0.149	0.074

Table 5. The total static RAN consumption for an WA binary (kB).

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

5.6 RAM Memory Usage

Table 5 shows the per device RAM consumption of each software library used in the final WA binary. To obtain these numbers, we profiled the final map file produced after compilation. At runtime WA also dynamically allocates additional memory: 1.56 kB for the CPX, 560 bytes for the micro:bit, and 644 bytes for the Uno. From the table, we can see that in all cases, the RAM consumption of WA and RT is well within the RAM available of each device.

5.7 Compiling Static TypeScript

When compiling, the entire STS program, including the runtime, first 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 WA compilation to native code takes about 1ms per statement. While the TS compiler has been optimized for speed, WA's native compilation process has not been. For example, the CPX TS pass is dominated by compilation of the device runtime and takes about 100ms, whereas the WA pass typically only includes a small user program and a small bit of the runtime, resulting in less than 100ms. Thus, typical compilation times are under 200ms for typical user programs of 100 lines or less.

6 Related Work

Related work breaks into three parts, following the flow of the paper: the programming of MCUs, type systems for JavaScript, and embedded runtime environments for MCUs.

6.1 Programming microcontrollers

There are four ways to program MCUs, (ordered roughly from lower to greater resource usage):

1. a program executing on a host computer sends commands (via some communication protocol) to a fixed program loaded onto the MCU (e.g., Scratch's device extensions [21], Arduino's Firmata library [3]).
2. a binary for the MCU is compiled on a host computer and loaded onto the MCU - this is the common method for Arduino [23];
3. bytecode is compiled on the host and loaded onto MCU, where it is interpreted (e.g., Java for embedded systems [11];
4. a complete compiler and virtual machine are put on the MCU, as with MicroPython [15] and Espruino [24];

In the educational and maker [12] settings, MCUs often are embedded in projects. Here, option 1 has limited utility, as in such projects the MCU is powered by battery and is not connected to a host. Option 2 does not operate well on computers where installation of programs and additional software is forbidden, common in educational environments. Options 3 and 4 suffer in terms of performance on small MCUs, operating slower than native C++ code, and consuming most of the resources (flash and RAM) of the MCU.

Our platform slots into options 2 and 3, using the WA architecture and USB pen drives to avoid the need for installation of native applications or device drivers. Our use of STS provides better performance than the MicroPython and Espruino solution, as demonstrated in the evaluation, and allows us to fit in the very constrained space of the Uno.

6.2 Types for JavaScript

By design, TypeScript does not provide a type soundness guarantee [6].

Safe TypeScript [20] separates the worlds of typed and untyped objects: the *dynamic* type is an implicit union type of all *known* static type, while Any is an *unknown* type, coming from the untyped JavaScript world. Type tags are needed to distinguish the two worlds; Safe TypeScript also adds extra runtime checks to prevent untyped JavaScript from destroying the type invariants of typed JavaScript.

StrongScript [22] extends TypeScript with a type constructor (!) for *concrete* types, which allows the programmer to choose between untyped code, optionally-typed code, and concretely typed code, which provides traditional type soundness guarantees. As with TypeScript, every JavaScript program is a StrongScript program.

STS differs from these efforts by outlawing untyped or optionally typed code: it is a non-goal of STS to support arbitrary JavaScript programs. In this sense, 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. STS goes further by outlawing downcasts (necessary for our untagged implementation).

STS could benefit by enlarging the set of types that can be inferred, so that more JavaScript idioms could be allowed. Chandra's work on type inference for static compilation of JavaScript [10] is one path forward.

6.3 Embedded Runtime Environments

Typically written in C and/or C++, environments for MCU programming all share a common design goal: to support developers by providing primitives and programming abstractions. Platforms can range from simple C++ classes that control hardware, to real-time operating systems (RTOSs) with scheduling and memory management.

Arduino [23] 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 through a heavy emphasis on global variables.

TinyOS [17], Contiki [13], RIOT OS [5], Mynewt [1] mbed OS [4], Zephyr [2] 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. RT bridges that semantic gap between the higher level language and the microcontroller, offering appropriate abstractions and higher level primitives written natively in C++.

7 Conclusion

We have presented and evaluated a new platform designed to bring modern language features and tooling to microcontrollers. Our aim was to do this in an extensible way which supports novice programmers with block-based programming while providing a progression path to a text-based scripting language and ultimately to C++. Our platform includes a new C++ runtime called RT which is designed to make efficient use of the limited resources on a microcontroller. A statically-typed subset of TypeScript forms the basis for both blocks- and text-based programs, created and compiled using a new web-based IDE named WA .

Our aspiration is to enable a new paradigm for programming pretty-much anything, even an Arduino Uno-class MCU, by anyone – novices and professional developers alike, from anywhere, i.e. without the need for traditional heavyweight embedded toolchains and IDEs. Our open-source implementation is in daily use, with thousands of users writing programs targeted at several different MCU-based devices. In this sense, we have achieved our goal. We also have anecdotal evidence that our platform – in terms of both language and tooling – is intuitive to professional developers with no experience of embedded development.

References

[1] [n. d.]. Apache Mynewt. https://mynewt.apache.org/. ([n. d.]). (Accessed on 11/16/2017).

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

[3] Arduino. 2017. Firmata Library. (2017). https://www.arduino.cc/en/Reference/Firmata

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

[5] 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.

[6] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[7] 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.

[8] Gaetano Borriello and Roy Want. 2000. Embedded computation meets the world wide web. *Commun. ACM* 43, 5 (2000), 59–66.

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

[10] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. 2016. Type inference for static compilation of JavaScript. In *OOPLSA 2016*. 410–429. http://doi.acm.org/10.1145/2983990.2984017

[11] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. 2000. Java Bytecode Compression for Low-end Embedded Systems. *ACM Trans. Program. Lang. Syst.* 22, 3 (May 2000), 471–489. https://doi.org/10.1145/353926.353933

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

[13] 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.

[14] Neil Fraser. 2015. Ten Things We've Learned from Blockly. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) (BLOCKS AND BEYOND '15)*. 49–50. http://dx.doi.org/10.1109/BLOCKS.2015.7369000

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

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

[17] 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.

[18] 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.

[19] Microsoft. 2016. TypeScript Language Specification. (2016). https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md

[20] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 167–180. http://doi.acm.org/10.1145/2676726.2676971

[21] 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

[22] 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

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

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

A Static TypeScript Subtype Relation

See Supplementary Material Document.

B USB Flashing Format

See Supplementary Material Document.