MetalScript

Bare-metal JavaScript for small embedded devices

Michael Hunter
elec.mike@gmail.com
coder-mike.com
metalscript.com

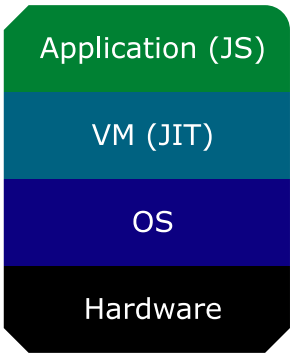
2018-10-16 For ECMA TC53, October 2018, Boston U.S.A

Note to readers: metalscript.com is really just a stub at this point – it doesn't have anything real on it. Rather go to my blog.

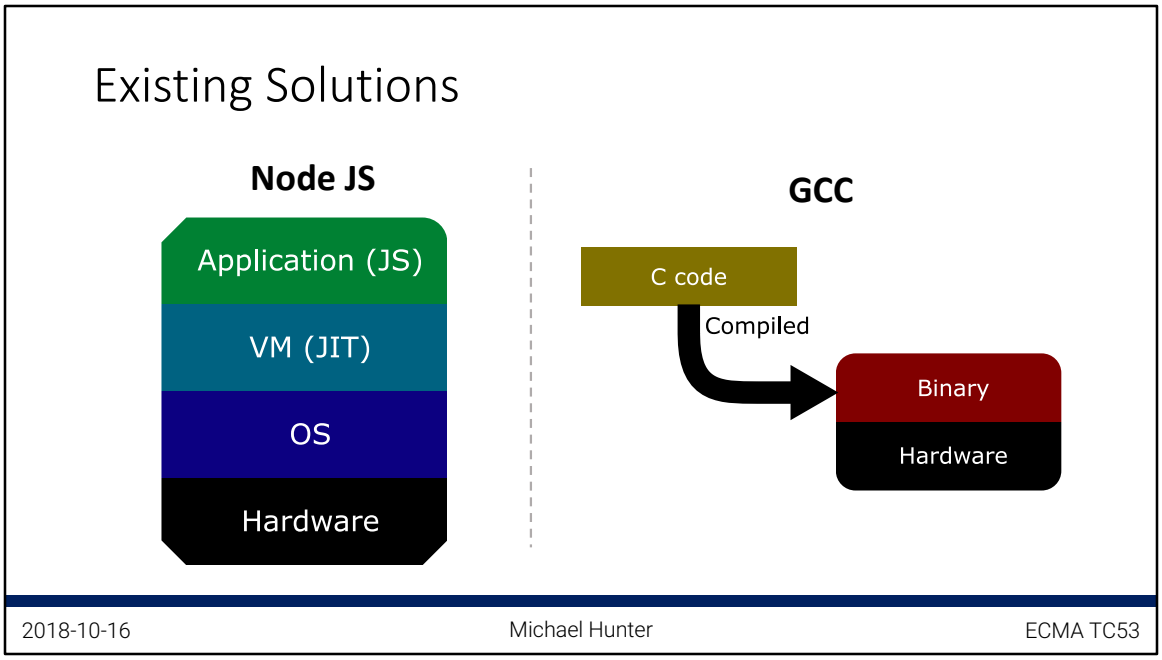
The notes I've added in the bottom of these slides are mostly for people who want a bit of extra information beyond what I presented, or in some cases documenting things I would have said verbally but weren't in the slides.

Existing Solutions

Node JS Approach

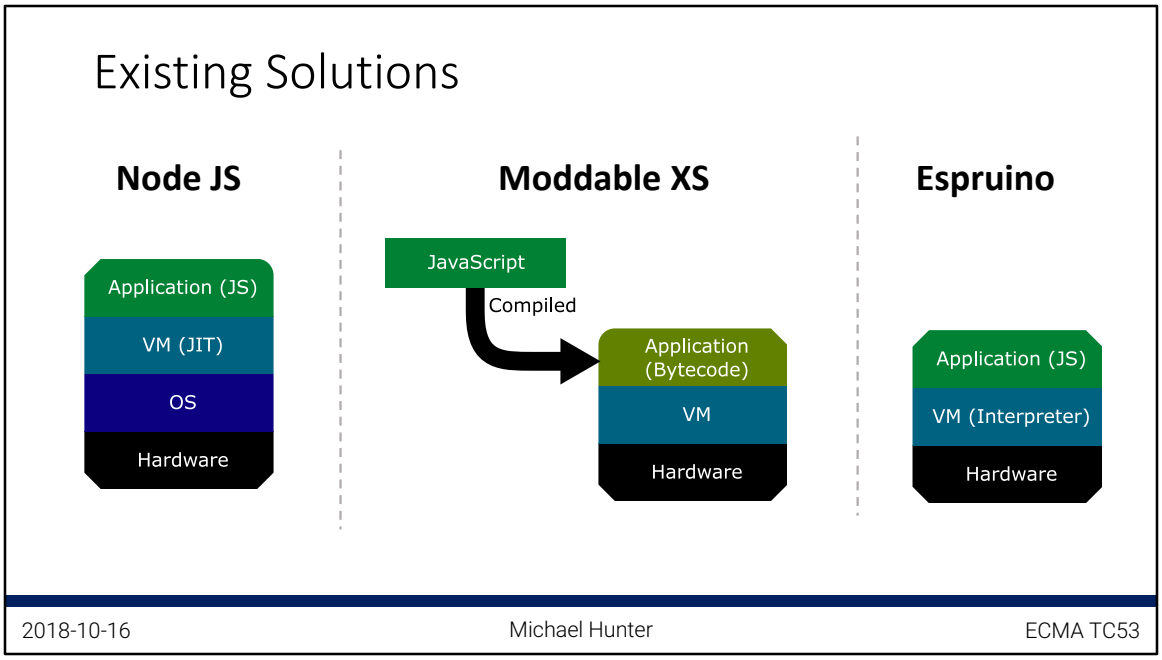


- Embedded Linux
- Tessel



End 5:15 ← if you're reading these notes, this time-stamp is just to help me avoid going over time while presenting. It starts around 5 minutes because I spent some time introducing myself and my history before getting here.

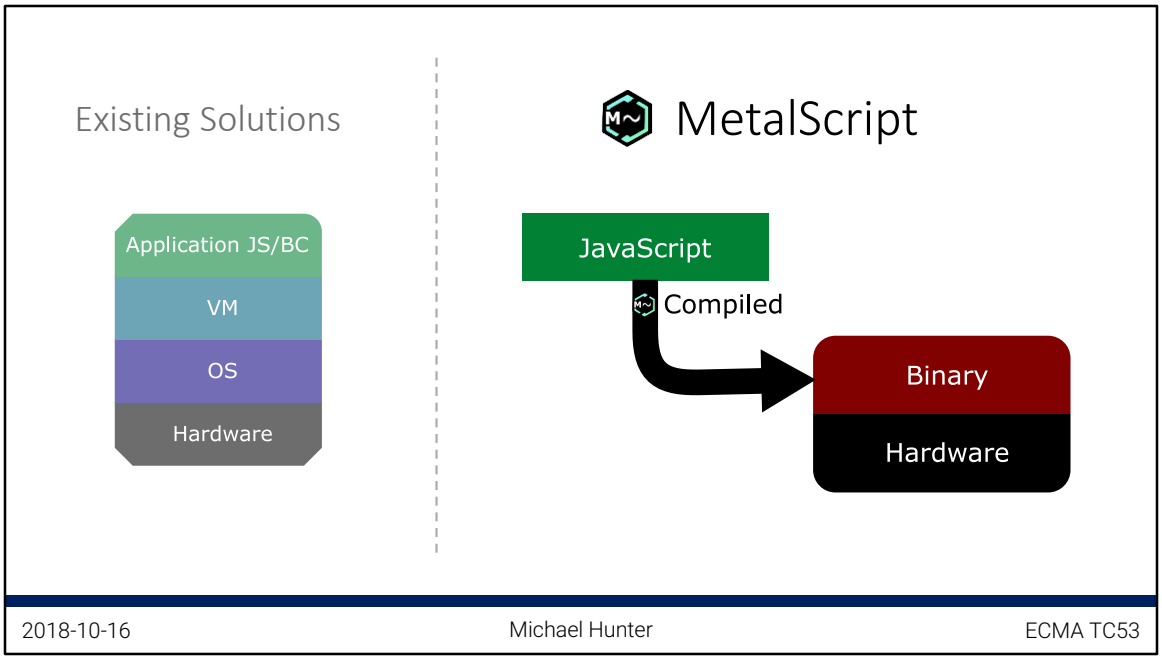
Slide is contrasting node-js solution with opposite end of the spectrum of bare-metal C.



End 06:00

Other solutions in between node-js and bare C.

Note that this is over simplified. You can run the XS parser and bytecode compiler on the device if you want a more “interpreted” solution, and Espruino offers some minimal pre-compilation behavior for certain limited functions. Also, just because you don’t need an OS or RTOS doesn’t mean you can’t have one.

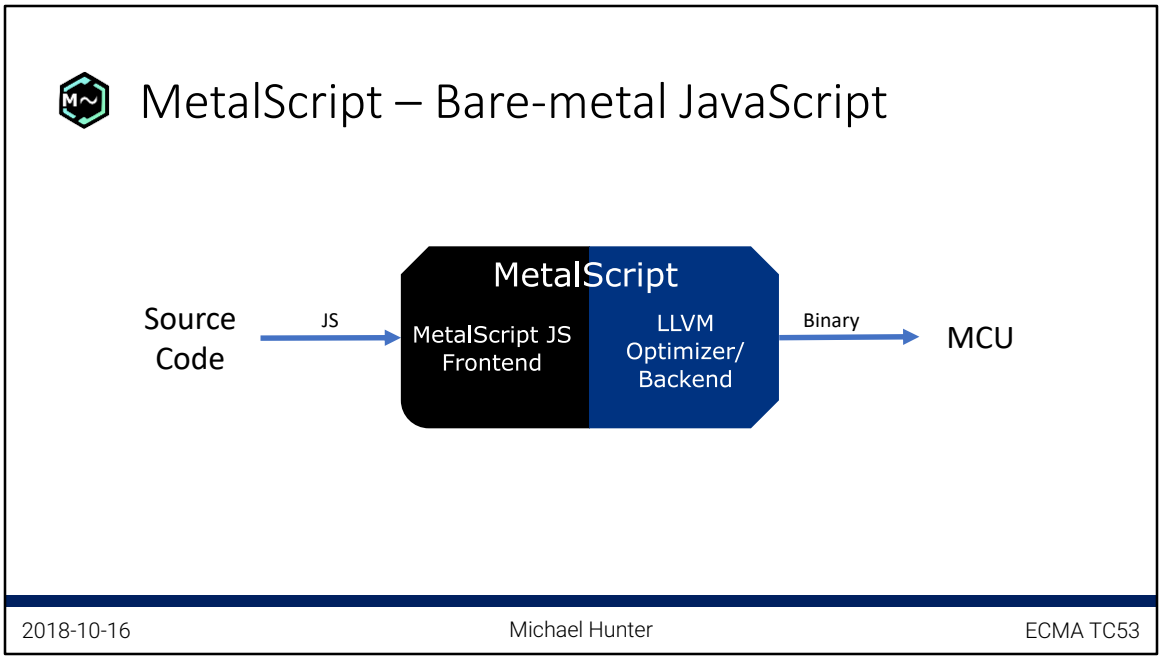


End 06:20

Need I say more?

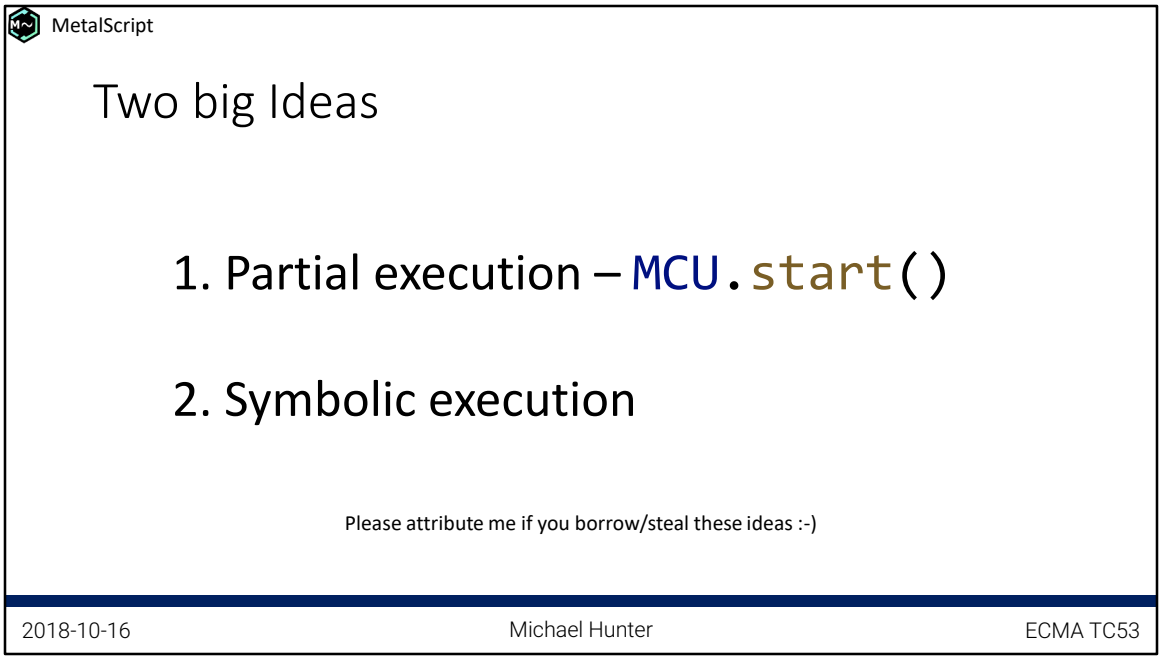
MetalScript – Bare-metal JavaScript

End 06:40



End 07:00

For those reading this who don't know LLVM, it's the part of the Clang C/C++ compiler that handles optimization and machine code output, with support for more a ton of hardware architectures. This makes MetalScript automatically capable of targeting pretty much every major platform, including ARM, X86, and WebAssembly for example.



End 07:20

Even better than attributing me, don't copy my ideas and just help me get them going in MetalScript. ;-)

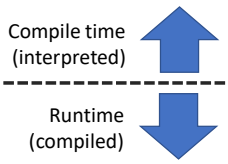
Partial Execution

- Program starts executing in the compiler
- Program is **suspended** when it calls `MCU.start()`
- Program is **resumed** on the device

End 07:50

MetalScript

Partial Execution



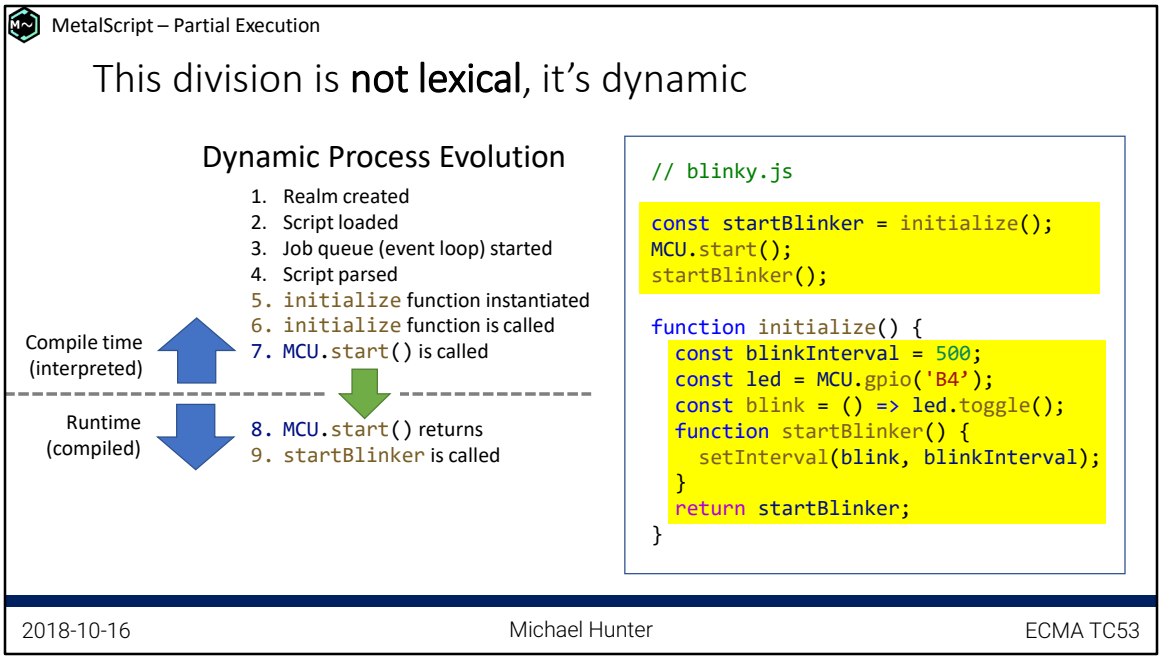

```
// blinky.js

const blinkInterval = 500;
const led = MCU.gpio('B4');
const blink = () => led.toggle();

MCU.start();

setInterval(blink, blinkInterval);
```


End 8:30



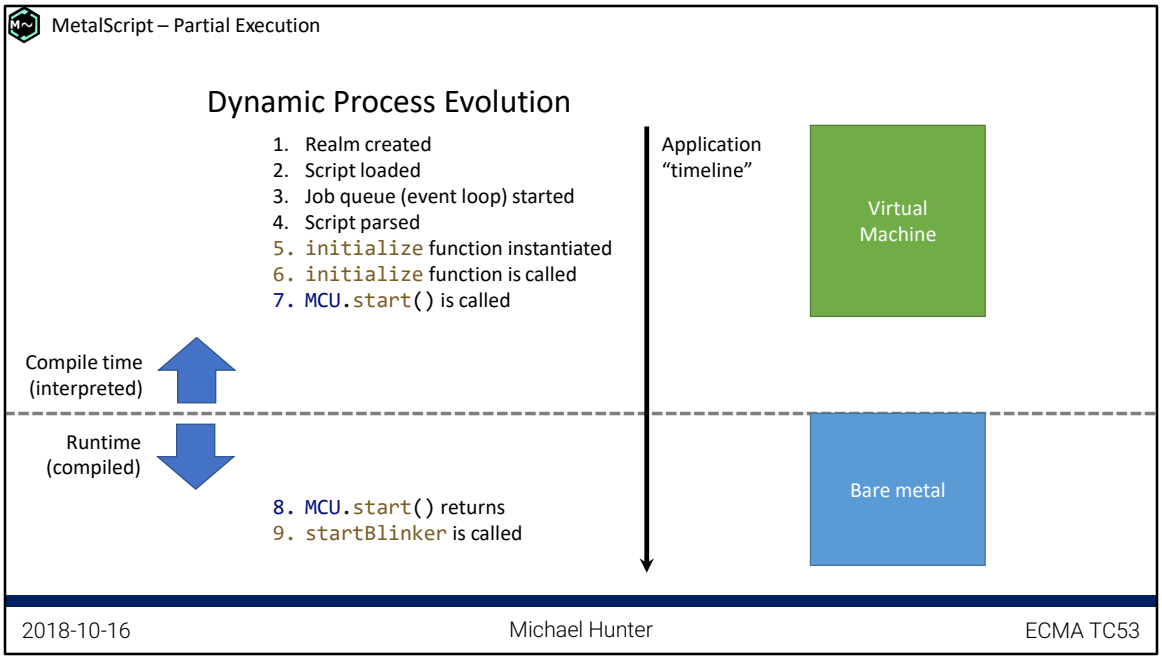
End 10:40

The key takeaways for people reading this:

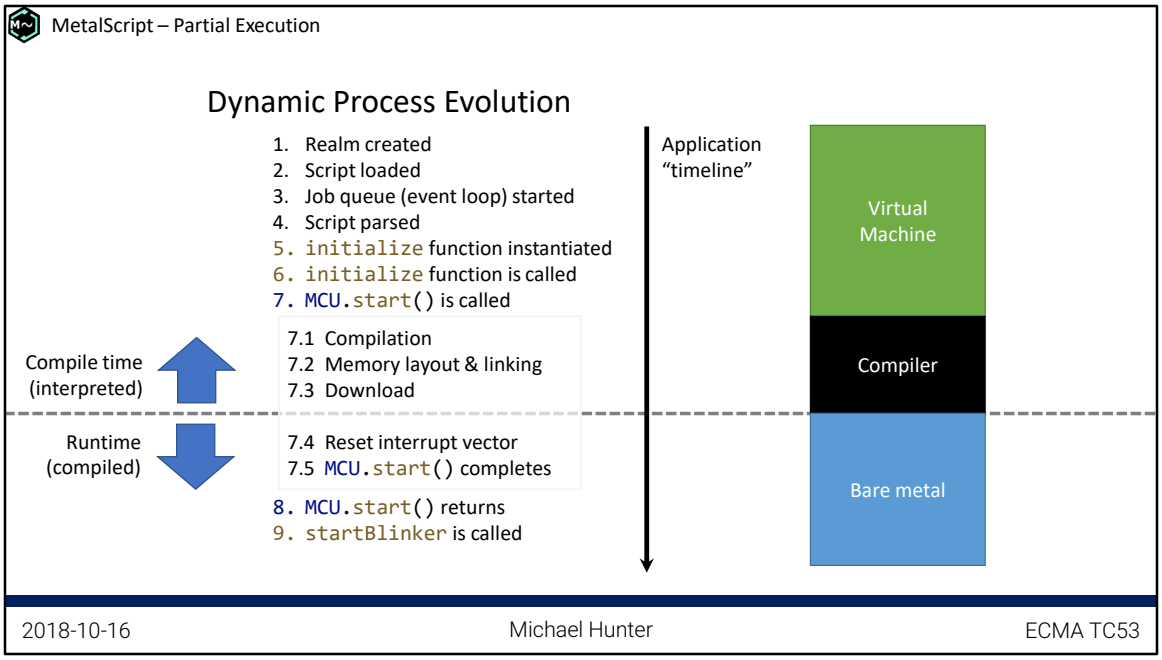
- Note that the closure state is transferred seamlessly from compile time to runtime
- The order of lines of code has nothing to do with what is compile time or runtime. It's all about the order of execution.

For anyone who cares, what I mean about the `initialize` function being *instantiated* is this https://tc39.github.io/ecma262/#sec-globaldeclarationinstantiation
Or to put it more simply, remember that:

1. Functions are objects (and objects must be instantiated to exist)
2. Functions are hoisted (they are created at the beginning of the enclosing variable scope)


MetalScript – Partial Execution
Dynamic Process Evolution
1. Realm created
2. Script loaded
3. Job queue (event loop) started
4. Script parsed
5. initialize function instantiated
6. initialize function is called
7. MCU.start() is called
Application "timeline"
Virtual Machine
Compile time (interpreted)
Runtime (compiled)
8. MCU.start() returns
9. startBlinker is called
Bare metal
2018-10-16
Michael Hunter
ECMA TC53

End 11:50



End 11:50

The reset interrupt vector jumps straight to the remainder of MCU.start, which will do low level things like initializing memory, etc. (this functionality is not implemented yet). This is equivalent to the invisible code that comes before "main" in a C program (normally assembly code linked in automatically by the compiler based on the target). When MCU.start is done, it jumps to the remainder of the application – in "C" this is "main", and in MetalScript this is internally called "resume".

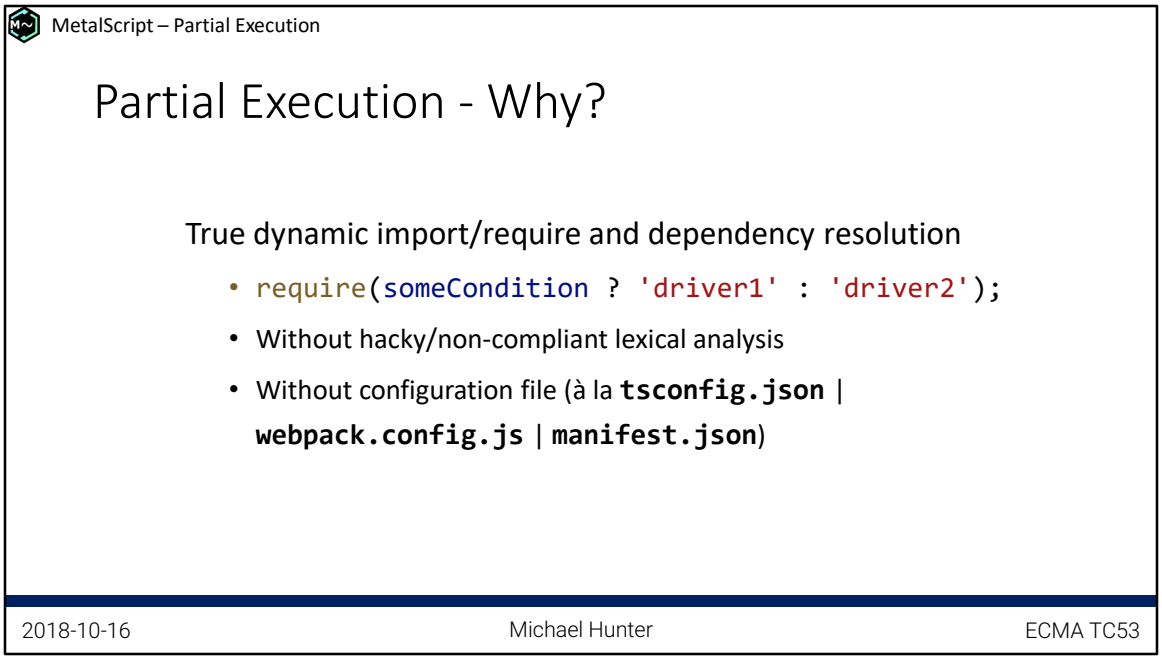
Partial Execution - Why?

"Instant" startup time

Faster than C

End 12:50

Potentially faster than C because C forces you to write code at runtime that could otherwise be at compile time, for setting up the initial state. Although, as it's been noted by others I've spoken to, this difference might not be significant in a typical application anyway – both C an MetalScript will be "fast enough" for most purposes.



End 13:35

To readers:

Would also work fine with top-level-async and dynamic `import` (currently proposal ES features).

The kind of lexical analysis I'm talking about is where the compiler might search for the string "require" or a statement that calls a function named "require" in the global scope. Both of these are "hacky" since they would break if called `require` by a different name or with a non-literal argument.

Partial Execution - Why?

No need for a “different language” for compile-time behavior

- C preprocessor
- C++ template metaprogramming
- Linker scripts
- Makefiles

End 14:00

Partial Execution - Why?

Precompute data/functions at compile time (ROM)

- DSP FIR filters
- Encryption/CRC tables
- Conversion/calibration tables
- Internal registries

End 14:35

“Internal registries” is a more subtle one. But for example, in the cold chain monitoring firmware, there’s a module that manages the power modes. It needs to know about every other module in the system that may want to influence or react to power state changes such as “battery-low”, but you don’t want this to be a code dependency where the power module code is coupled to every other module. Short of doing clever linker tricks, the easiest solution in C is to have each module register itself with the power module at startup. In MetalScript, this registration/subscription can happen at compile time.

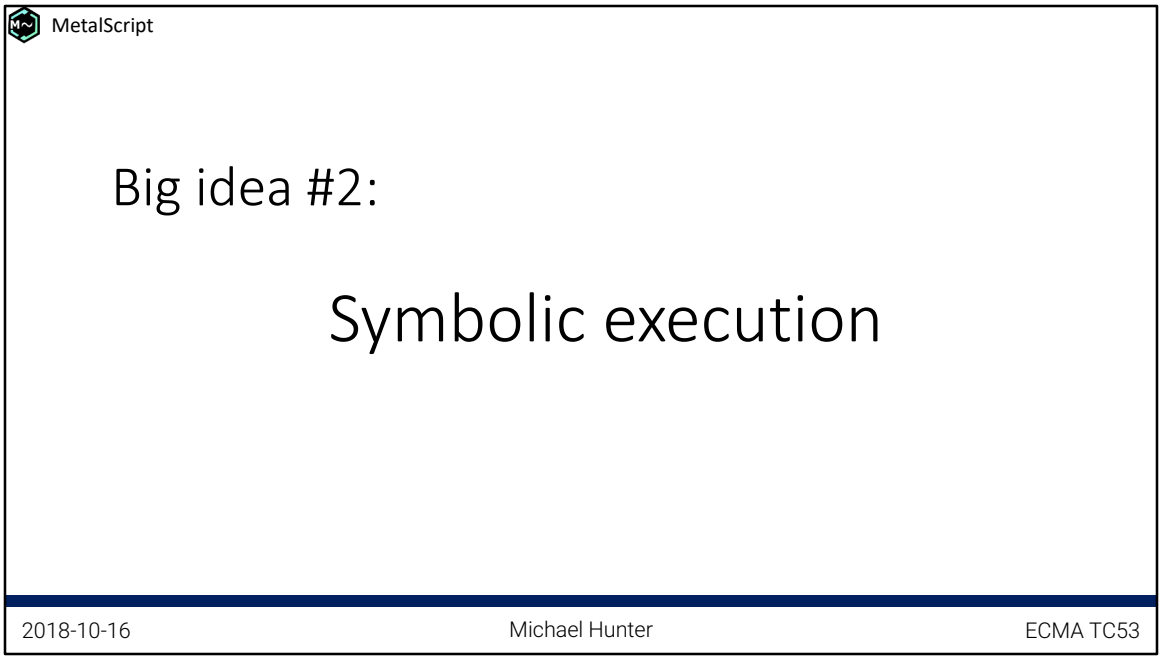
Similar “registration” processes are quite frequent anywhere you have inversion-of-control. E.g. your serial port protocol needs to register to receive messages/events from the serial port. For performance you may want this to be a direct call from the serial port, but for code abstraction you want it to be registered. With MetalScript you can have the best of both.

Partial Execution - Why?

Shared code

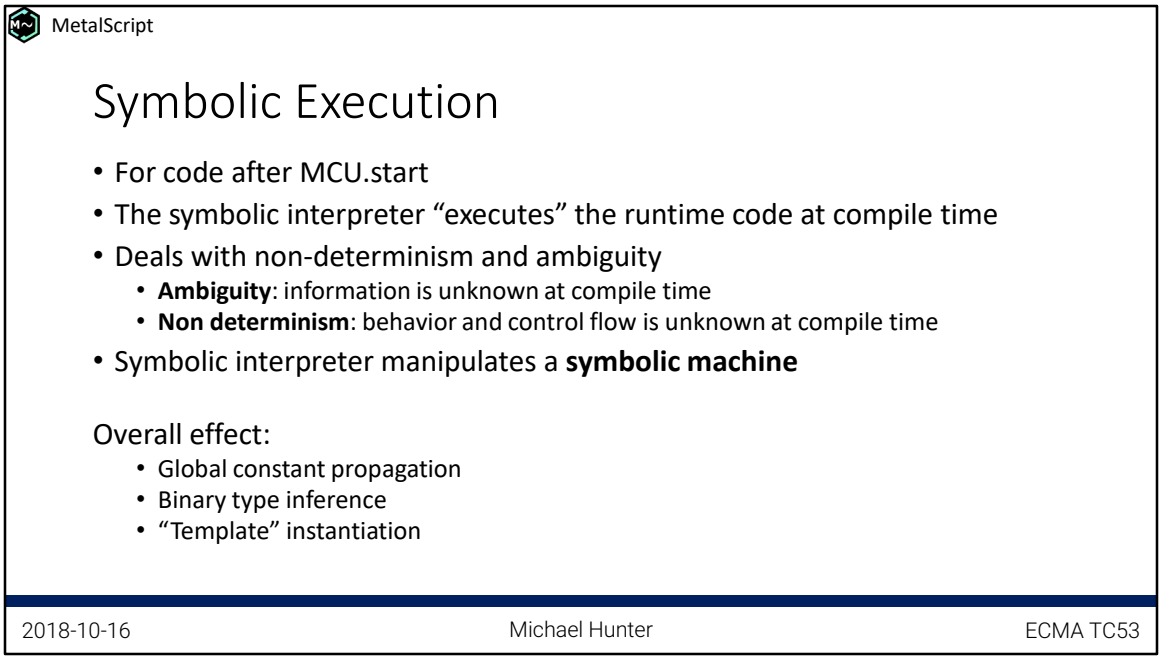
- Shared between compile time and runtime
- Code at any time can use third party libraries (e.g. NPM)

End: 15:20



15:20

To anyone reading these notes. This was a tough one to figure out how to best present. There are a number of different analogies that could be used to describe symbolic execution. If it's confusing and you want to know more, just shoot me an email at elec.mike@gmail.com.



End 17:20

To readers –

The term “template” here refers to something like C++ function templates, which don’t exist at runtime in their own right but rather exist because they’re “called” from code that does exist. In MetalScript, all JS code is considered to be templates and don’t actually exist at runtime unless they’re invoked/used. Code is invoked directly or indirectly from the entry points to the program, which includes:
 * the continuation after MCU.start (if MCU.start is called – see my blog post on MetalScript in C for an example where you might not have an MCU.start but can still export other entry points)
 * as well as things such as callbacks. When you do `setInterval(callback, ..)`, the callback becomes a secondary entry point to the system, and in turn the callback will be “instantiated”.

Caveat

Symbolic execution in MetalScript
is a **work-in-progress**
(started recently)

End 17:40

Symbolic Execution - Example

- MCU object is known
- gpio property can be statically resolved
- gpio function can be statically invoked
- Resulting GPIO object can exist statically (at compile time only)
- Toggle method can be invoked inline (zero overhead for call)

```
MCU.gpio('B4').toggle()
```


End 18:40

JavaScript

```
MCU.gpio('B4').toggle()
```

IL

```
t0_18 = sys_op("resolveBinding", "MCU", false, undefined);
t0_19 = sys_op("getValue", t0_18);
t0_20 = sys_op("requireObjectCoercible", t0_19);
t0_21 = sys_op("newRef", t0_20, "gpio", false);
t0_22 = sys_op("getValue", t0_21);
t0_23 = sys_op("getThisValue", t0_21);
t0_24 = list_new();
t0_25 = copy("B4");
t0_26 = sys_op("getValue", t0_25);
list_push(t0_24, t0_26);
t0_28 = sys_op("throwIfNotCallable", t0_22);
t0_29 = sys_op("call", t0_22, t0_23, t0_24);
t0_30 = sys_op("getValue", t0_29);
t0_31 = sys_op("requireObjectCoercible", t0_30);
t0_32 = sys_op("newRef", t0_31, "toggle", false);
t0_33 = sys_op("getValue", t0_32);
t0_34 = sys_op("getThisValue", t0_32);
t0_35 = list_new();
t0_36 = sys_op("throwIfNotCallable", t0_33);
t0_37 = sys_op("call", t0_33, t0_34, t0_35);
t0_38 = sys_op("getValue", t0_37);
```


End 19:10

I've highlighted the main points in a brighter color to make it easier to follow.

For anyone reading this who is interested in more detail:

* sys_op is an IL operation that is like a function call, but calls an ECMAScript abstract operation (it works exactly the same as a function call but searches in a different namespace)
* I've tried to keep the names of the abstract operations as close as possible to the ES spec for maintainability. For example, `resolveBinding` refers to this operation: https://tc39.github.io/ecma262/#sec-resolvebinding

The lists created are the argument lists passed during function calls.

If you compare with XS bytecode, MetalScript IL is significantly more expanded/verbose, since it doesn't need to run on the device but needs to be easy for other stages of the pipeline to process. See the extra slides after the end of the presentation for more details.

GPIO Toggle Example – Full Code

```
// ------------------ SYSTEM INITIALIZATION ---------------------

const ioMap = {
  // ...
  'B4': {
    address: 0x40020814,
    mask: 0x10
  }
  // ...
}

function gpio(name) {
  const pinInfo = ioMap[name];
  return {
    toggle() {
      const value = MCU.getIO(pinInfo.address);
      const newValue = value ^ pinInfo.mask;
      MCU.setIO(pinInfo.address, newValue);
    }
  };
}

MCU.gpio = gpio;

// ------------------------ USER CODE ---------------------------
MCU.start();

MCU.gpio('B4').toggle()
```

End 20:20
(this is a compiling example)

It's maybe worth noting that the `MCU` object is mutated but the gpio property is still statically resolvable. This is different from C++ where constexpr or template parameters cannot be mutated. In MetalScript this mutation can either be tracked by the partial execution or the symbolic execution depending on whether it happens before or after the `start` call, but would be better for static analysis if it happened before `start`.
I mention this because of its relevance to frozen realms – MetalScript allows user code to mutate the realm if needed, if you want to override or wrap built-in behavior or objects for testing, instrumenting, or polyfills for yet-unsupported features.

I mentioned in the talk, the system initialization code is only in this file because of how limited MetalScript is today, and writing the JavaScript for the system functions was quicker than making this built in.

GPIO Toggle Example – Output

JavaScript

11,557 symbolic instruction cycles

Compiles to

ARM Assembly

```
  .globl  resume ; -- Begin function resume
  .p2align  2
  .code32 ; @resume
resume:
; BB#0: ; %block1
  ldr r0, .LCPI0_0
  ldr r1, [r0]
  eor r1, r1, #16
  str r1, [r0]
  mov pc, lr
  .p2align  2
; BB#1:
.LCPI0_0:
  .long 1073874964 ; 0x40020814
; -- End function
```

Read
XOR
Write

2018-10-16 Michael Hunter ECMA TC53

End 21:50

The 11 thousand instructions (most IL instructions in this example are executed exactly once) is both a little worrying (for the performance of the compiler) but also highlights the potential power of this method for condensing a lot of meaning into a small space. I haven't paid much attention to reducing this number, so there is still quite a lot of room for improvement. But I want to get it working first before optimizing the compiler performance.

Most of the 11,000 instructions come from inlining abstract operations. The IL for the actual user code is much smaller.

Understandably, toggling a pin is a trivial example. Hopefully soon I will have some more comprehensive examples to show.

GPIO Toggle Example – Output

JavaScript

11,557 symbolic instruction cycles

Compiles to

x86 Assembly

```
  .globl  _resume # -- Begin function resume
  .p2align  4, 0x90
_resume: # @resume
# BB#0: # %block1
  xorl  $16, 1073874964
  retl
# -- End function
```


End: 22:05

Obviously we wouldn't be running x86 on an embedded device. But there's no reason MetalScript can't be used for desktop development one day, and compiling this example down to 1 assembly instruction just drives the point further.

Symbolic Execution – Why?

- Efficient
- Low-overhead abstraction
  - Dependency injection
  - Inversion of control/callbacks
  - Options-hashes
- Generic library can handle a lot of cases, specialized on-the-fly for your use case
- Application can adapt to different hardware
  - (e.g. product variations)
  - Might previously be done with `#if` directives
- Unit testing
- APIs can be designed for usability and generality

End 24:50

</end>

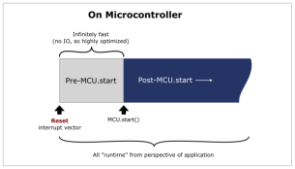
Read more on my blog…

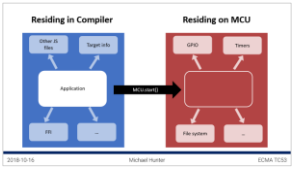
coder-mike.com

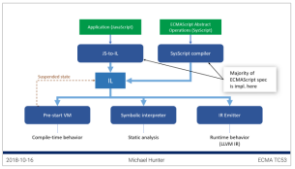
End: 25:35

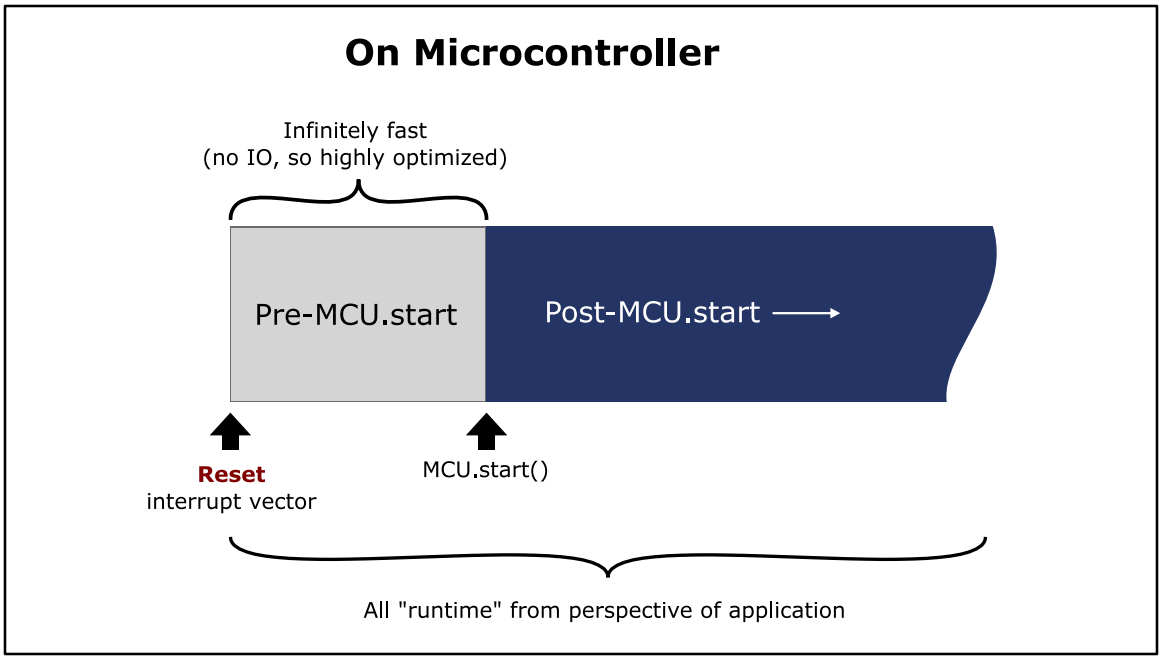
Questions/Discussion

Extra Sides


On Microcontroller
Pre-MCU.start
Post-MCU.start
Reset
interrupt vector
MCU.start()
All "runtime" from perspective of application

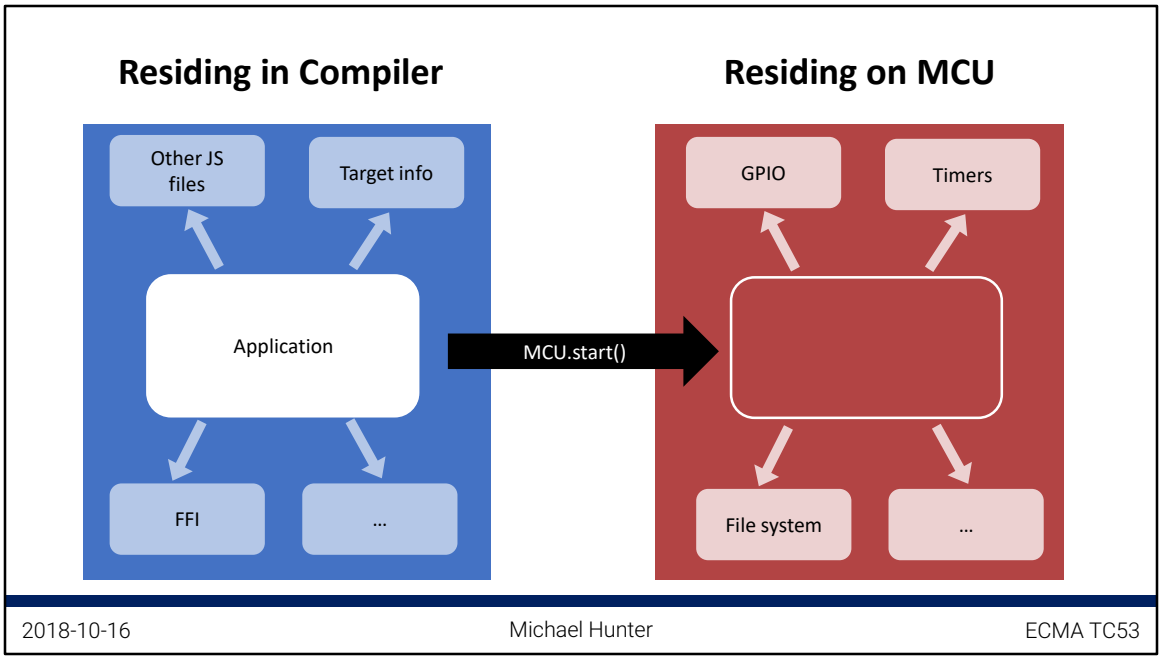

Residing in Compiler
Residing on MCU
Application
GPIO
Timers
File system
2018-10-16
Michael Hunter
ECMA TC53


Compile-time behavior
Static analysis
Runtime behavior
2018-10-16
Michael Hunter
ECMA TC53



This is a diagram to explain the alternative way of thinking about the pre-start code. You can think of all user code as running on the device, but because pre-start code is not permitted to access IO, it is subject to much better optimization, to the extent where it consumes no runtime cycles at all.

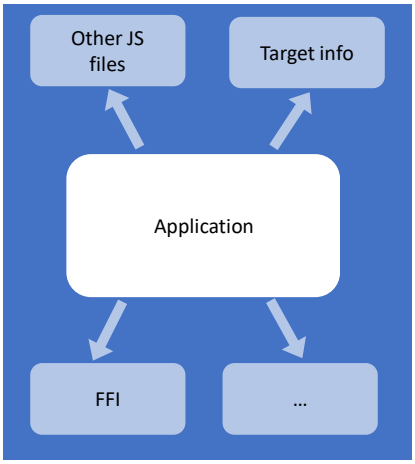
I've taken this model to the extreme. If there is a parse error in the source files, this is valid ECMAScript behavior and clearly defined by the spec (it has the effect of invoking the abstract operation HostReportErrors) – MetalScript will simply produce a corresponding binary image that has equivalent behavior (not much). Likely for practical purposes it will also warn the user, but that's to be nice, not as core functionality. The main way to protect against stupid mistakes in JS (ahead of running on the target) is use a linter or program in TypeScript. The objective of MetalScript, as stated earlier in the slides, is to take JS input and produce a binary output [of equivalent behavior].



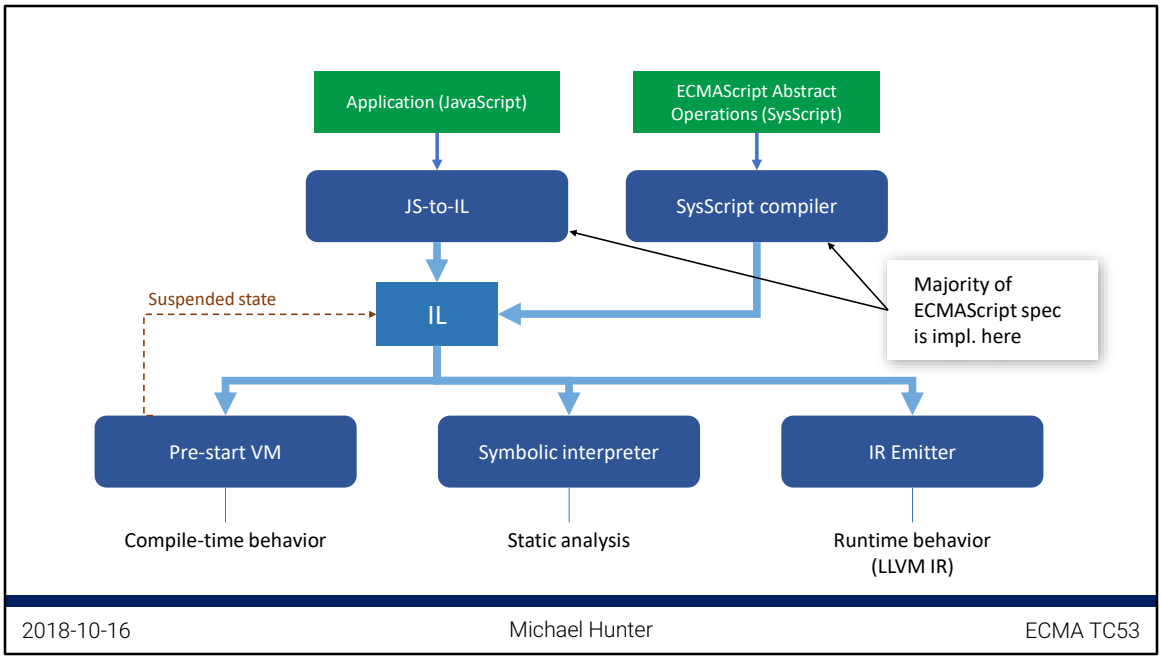
Illustrates how the application can access different external systems in different environments. From the perspective of the application, this just means that something (most things) in the environment/outside-world change as a result of calling a JavaScript function (MCU.start), which is not an uncommon paradigm – e.g. contrast to calling a function to delete a file in a nodejs application, which will also change the environment since now there is no file where was once one. Opening the hypothetical file will be an error after it's deleted but not before, so the set of valid external operations has changed as a result of calling the delete function. Similarly in MetalScript, for example, `require` will be valid before `start` but not `after`.

For the sake of fitting in the graphic, this doesn't mention all the differences between compile time and runtime. For example, it's fine to call setTimeout/setInterval at compile time, but it will only start "ticking" after the MCU is started (refer to the previous slide as to why this makes sense, rather than having the timeout expire during compilation itself). Similarly, setting up initial IO state (e.g. pin direction) makes sense at compile time, but will only affect the real world on the next (first) clock cycle, figuratively speaking, so some level of GPIO operation may be permitted at compile time.

Residing in Compiler



Alternative representation of the previous diagram (only makes sense if it's animated).



The IL (intermediate language) representation used internally in MetalScript is designed to be very simple since the semantics of IL is implemented in multiple places. The symbolic interpreter especially is complicated enough without also having to understand the whole ES spec. Rather, most of the spec is implemented in a domain-specific-language (DSL) I call SysScript.

The IL shown earlier in the slides is the output of JS-to-IL, a lightweight conversion that translates ES syntax into equivalent IL operations.