Swift Things

Programming the Internet of Things with Swift

Steven Gray
SoftSource Consulting
Portland, Oregon USA
steven.gray@sftsrc.com

Minimalist Autobiography

- Apple hacker from the early days
- Detours through many platforms over the years
- Early mobile entreprenuer starting in 2004

- Today I lead consultants in various research areas
  - IoT is a current concentration

Call to Action! (start at the end)

— Will this talk convince you to create IoT solutions with Swift?

Call to Action!

— Will this talk convince you to create IoT solutions with Swift?

Doubtful. This is a non-goal for our next 29 minutes.

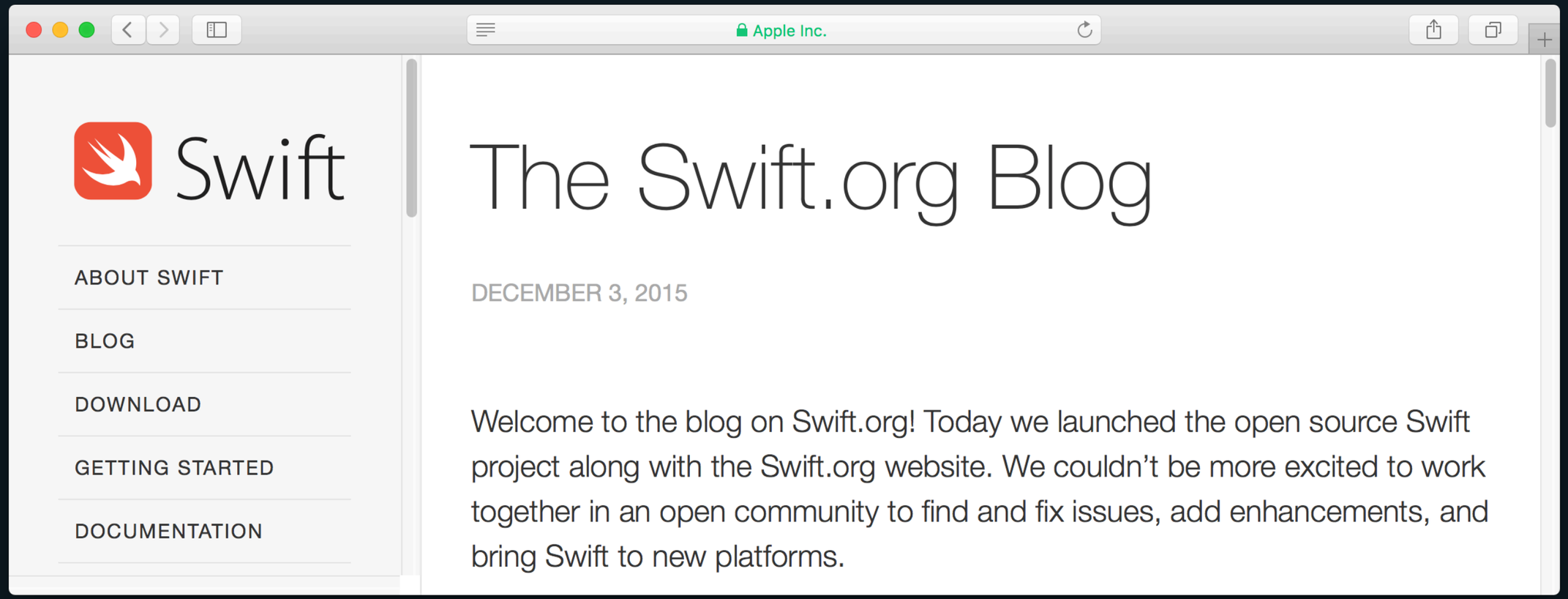
And given that time allotment, I will proceed with diligent haste.

Because of this, I will be leaving out many details. Catch me any time this week for more details.

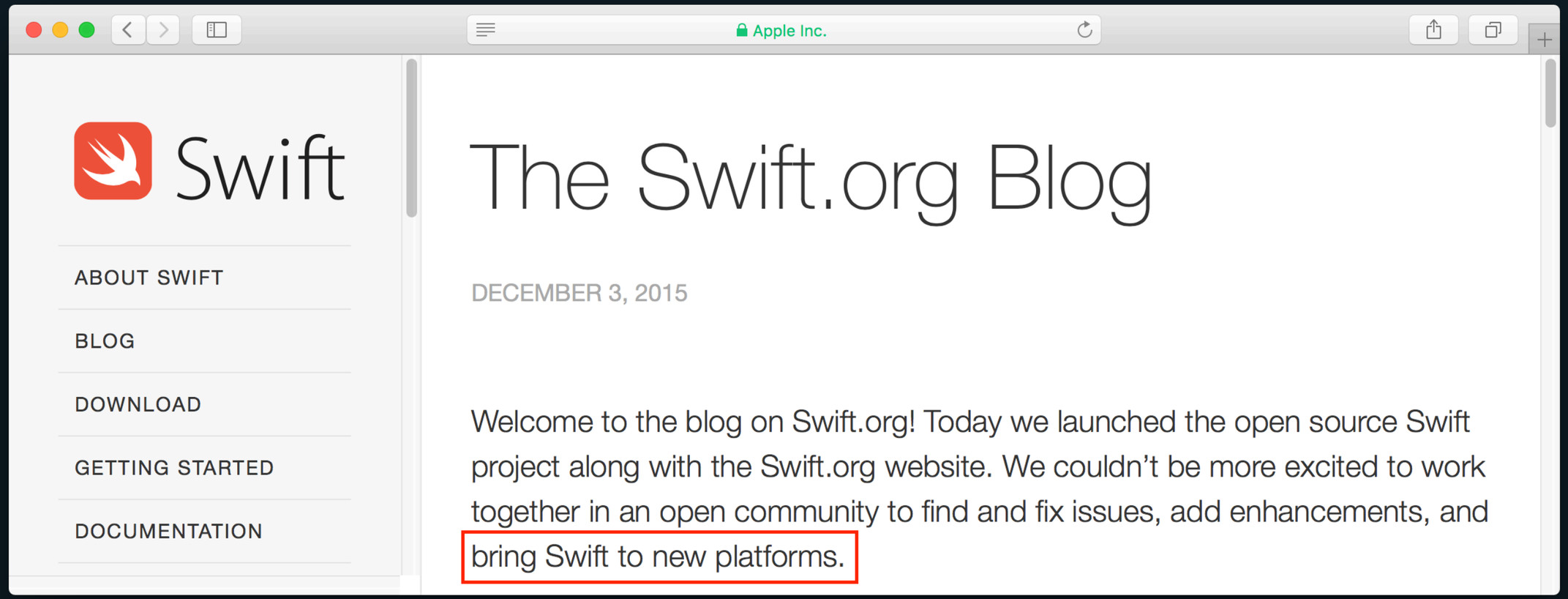
Today's agenda

1. Discuss whether a Swift-based IoT solution is even possible
2. Share our research
3. Begin a dialog with you for this week (and beyond)

Wayback Machine to December 2015



Challenge accepted!



Internet of Things (IoT)

— Means a lot of things to a lot of people.

For Today

Let us define the Internet of Things as:

— Small, resource-constrained devices*
— Battery powered
— Capable of connecting to the Internet in some way
— Inexpensive (< £25)*

*They won't always be this way

Texas Instruments CC2650 SensorTag

	SensorTag
Chip	ARM Cortex M3
Clock	48 MHz*
RAM	20 KB
Arch	32-bit
Battery (size of a 10p coin)	Months
Wireless	BLE & 802.15.4

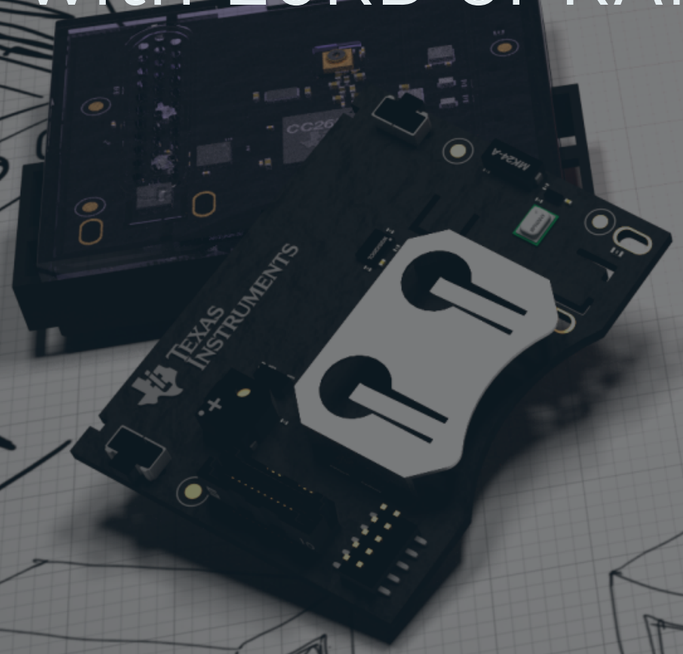
*46.9 times faster than an Apple][!

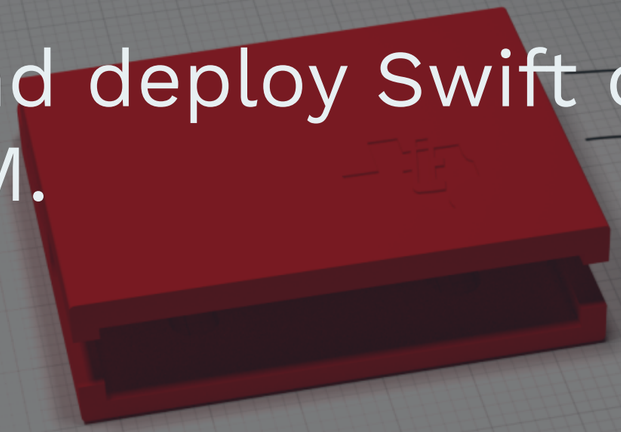
	SensorTag	iPhone 7
Chip	ARM Cortex M3	A10 Fusion
Clock	48 MHz	2.34 GHz*
RAM	20 KB	2-3 GB
Arch	32-bit	64-bit
Battery	Months	Hours
Wireless	BLE & 802.15.4	BT & 802.11

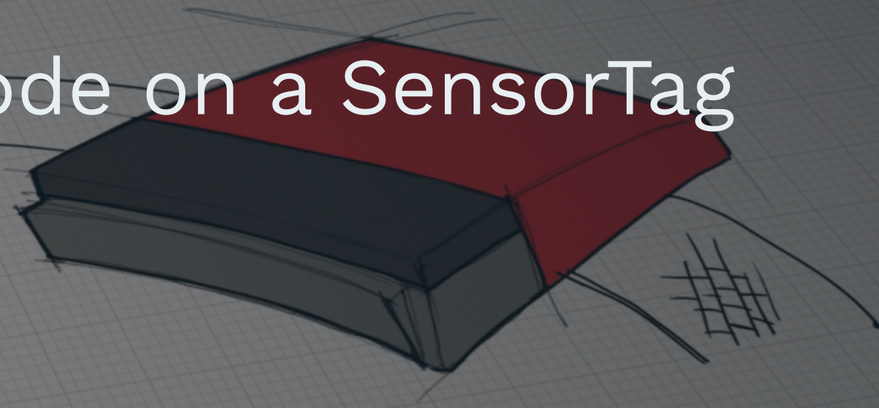
*2287.39 times faster than an Apple][

Our Challenge

Try to compile and deploy Swift code on a SensorTag with 20KB of RAM.

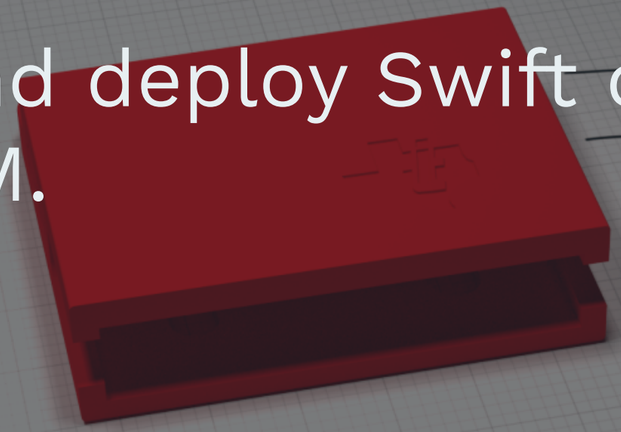


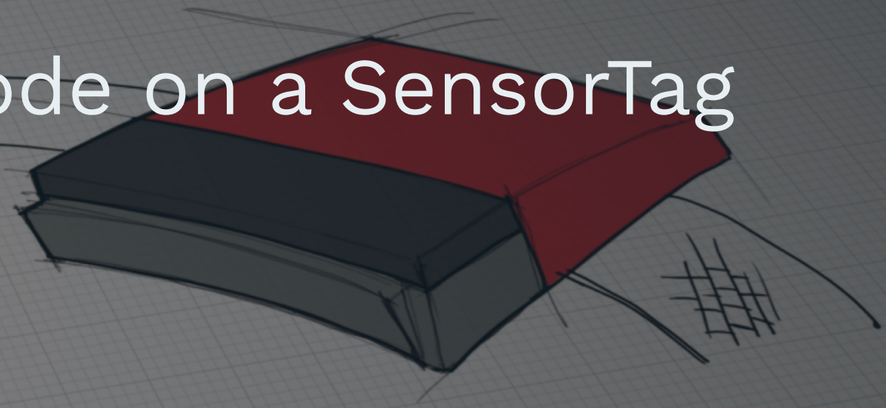




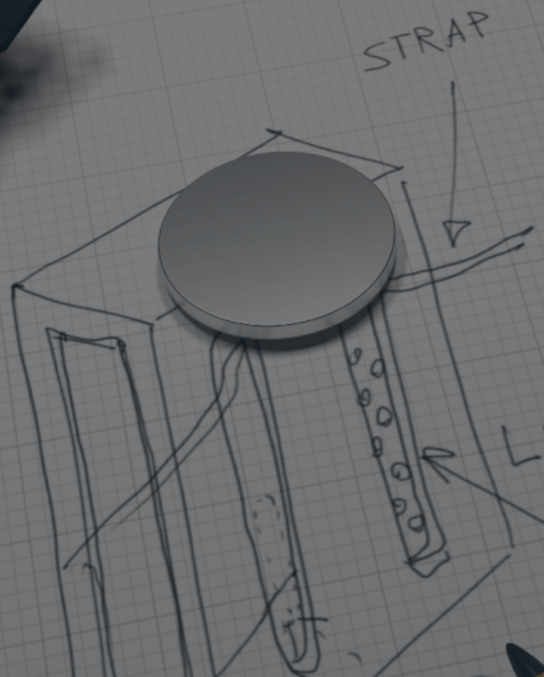
Our Challenge

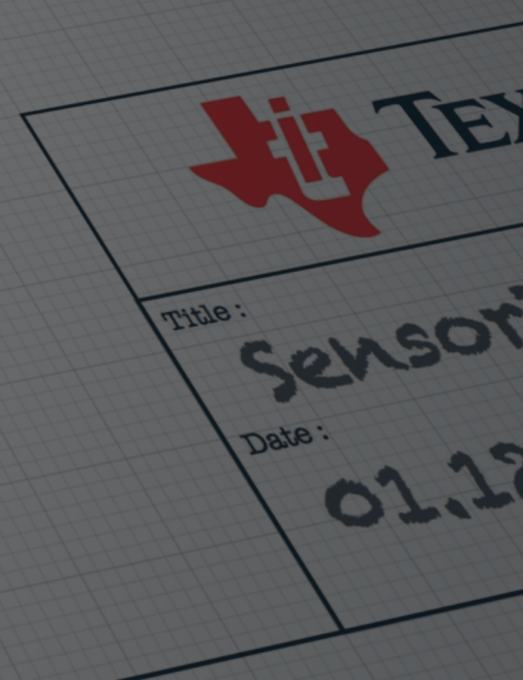
Try to compile and deploy Swift code on a SensorTag with 20KB of RAM.





Why?



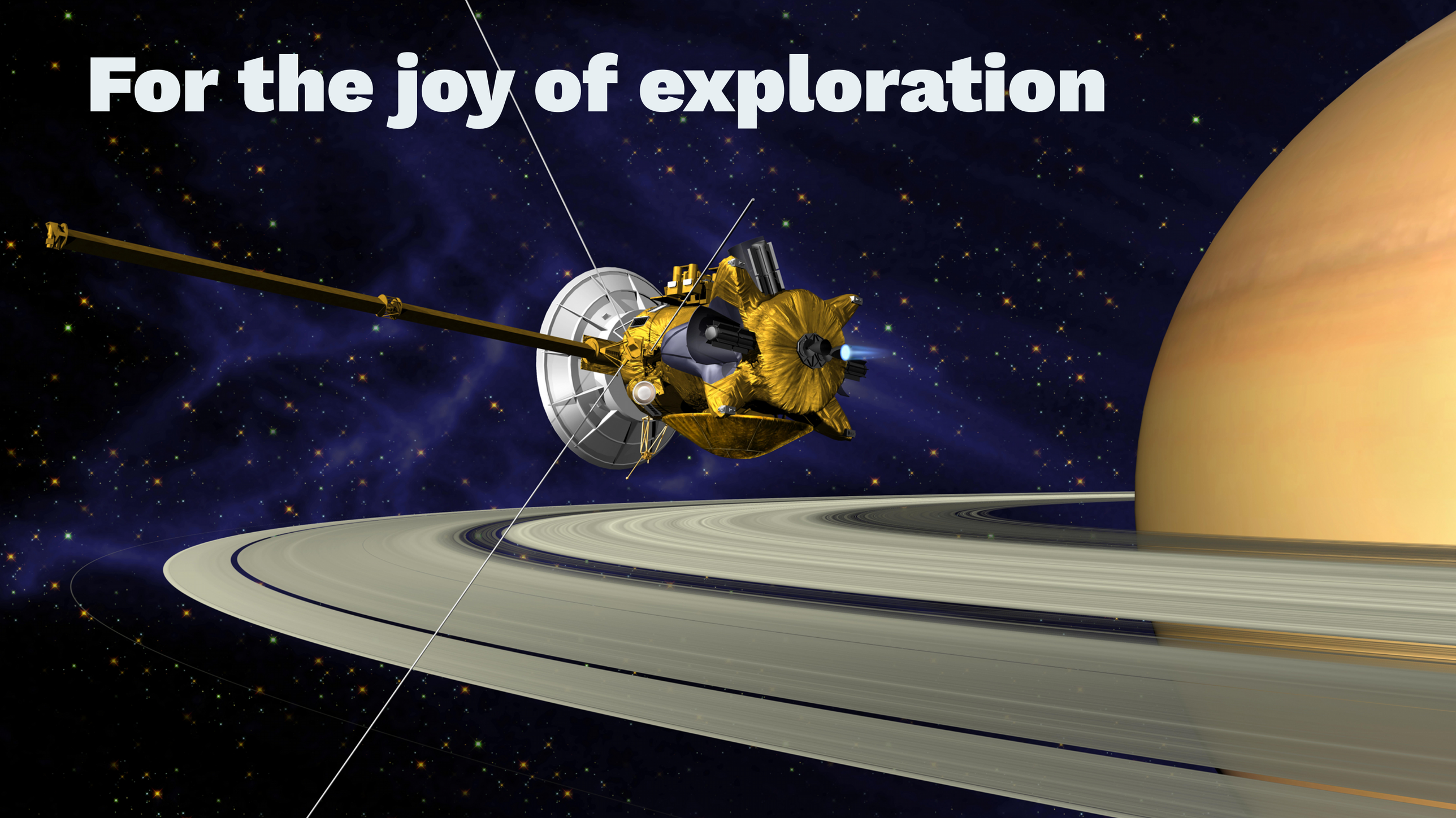


The Trite Answer

Why not?

So, really, why?

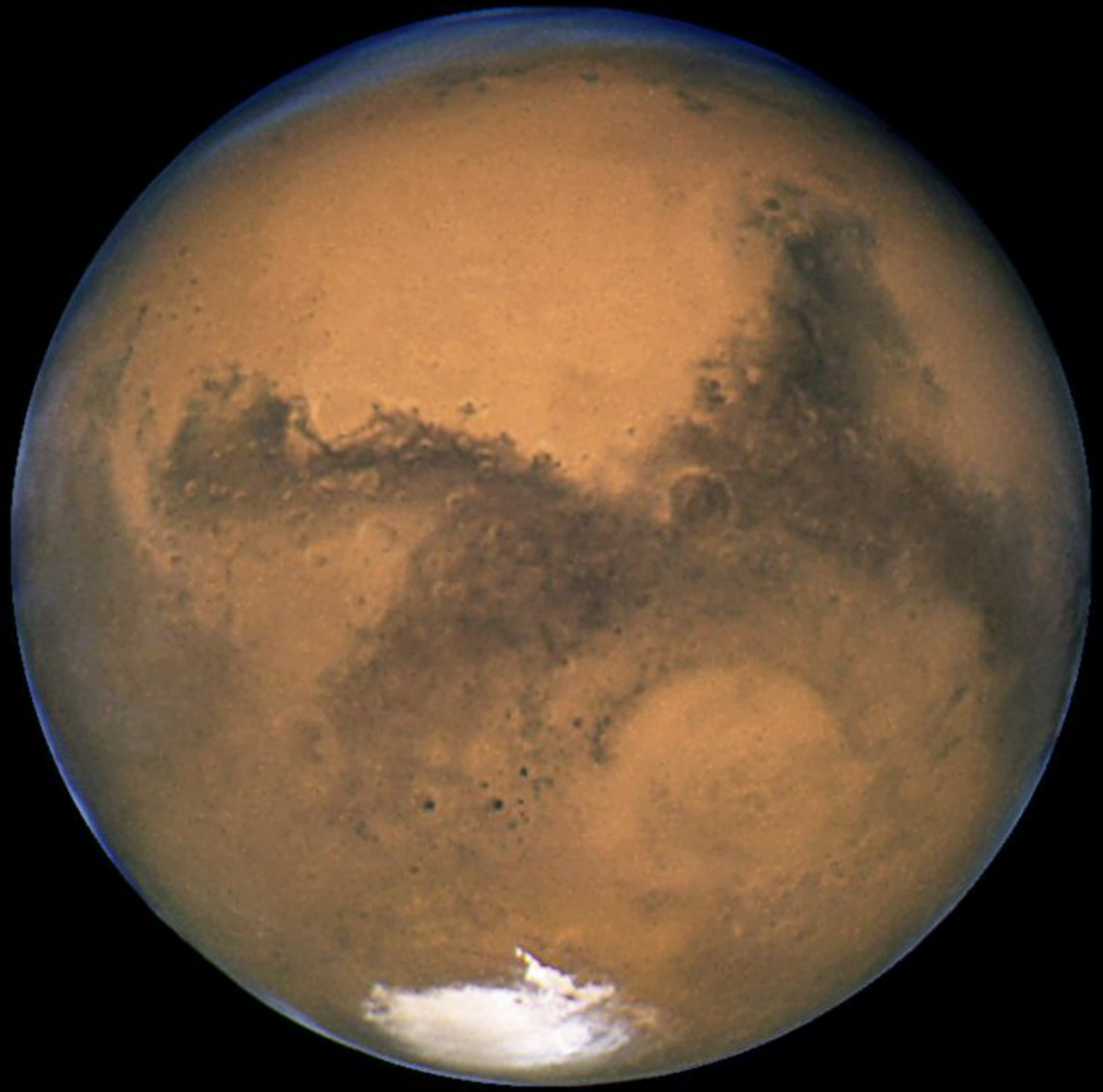
For the joy of exploration



Swift on iOS (Apple & many others)

Like exploring Earth

Swift on macOS (Apple & several others)



Server-side Swift (IBM & some others)

Swift on IoT (open frontier) [1]

[1] You would not believe how I was taught to pronounce Uranus as a child...

Swift on IoT: Questions to be Answered

1. Is it possible to go there?

Swift on IoT: Questions to be Answered

1. Is it possible to go there?
2. How do we get there?

Swift on IoT: Questions to be Answered

1. Is it possible to go there?
2. How do we get there?
3. What have we learned along the way?

Let's get on with it, shall we?[2]

[2] Obligatory reference considering "Aberystwyth Uni studies Monty Python's 'lasting appeal'"

With any trip (into space), we have constraints

- Require nothing fancy
  - Standard Mac (or Windows or Linux)
  - Standard build of Swift tools
    - Why?

Launching our first ship (to Uranus)

We're there when this works, right?

```
print("hello, world")
```

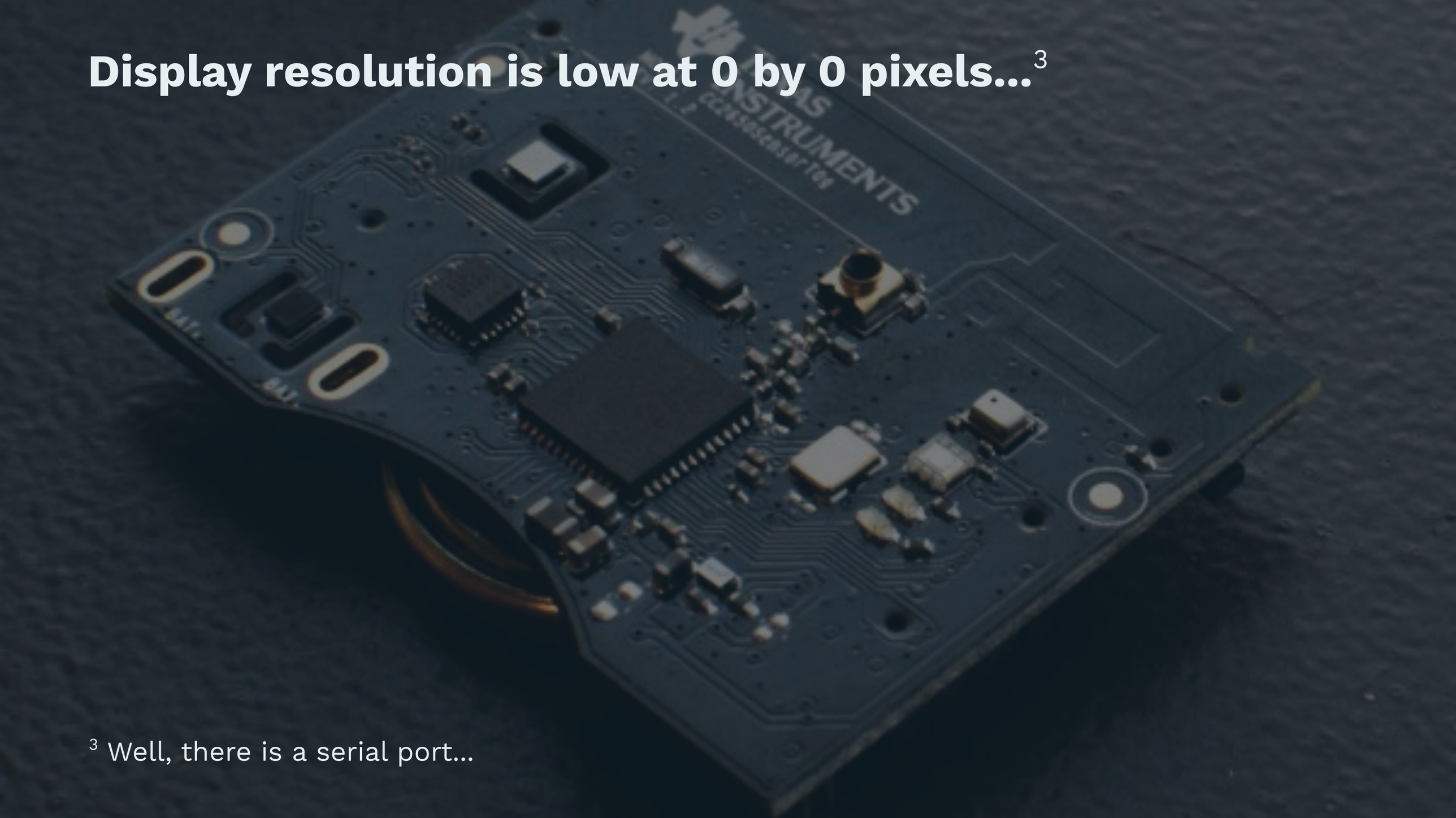
Launching our first ship (to Uranus)

We're there when this works, right?

```
print("hello, world")
```

Sure. But there's a problem...

Display resolution is low at 0 by 0 pixels...[3]



[3] Well, there is a serial port...

We're in orbit with: Morse code!

As there is no display,
blink the LED and sound the buzzer in Morse code:

```
.... . .-.. .-.. ---     .-- --- .-. .-.. -..
h    e l    l    o       w   o   r   l    d
```

So, is Swift for Morse code on an IoT device possible?

I'll give away the ending...

So, is Swift for Morse code on an IoT device possible?

Indeed. It is quite possible.

Demo time...

Swift on IoT

— Is it possible to go there?

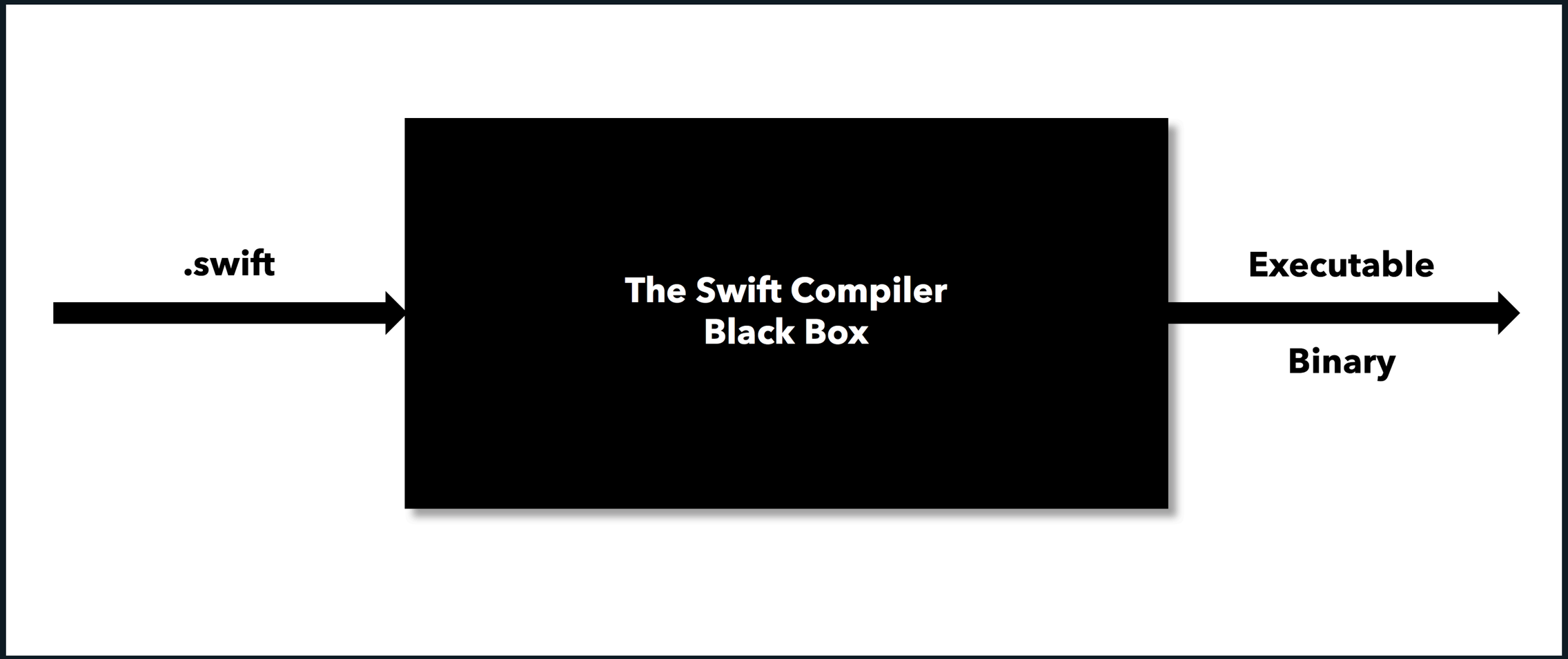
  — Yes.

— How do we get there?

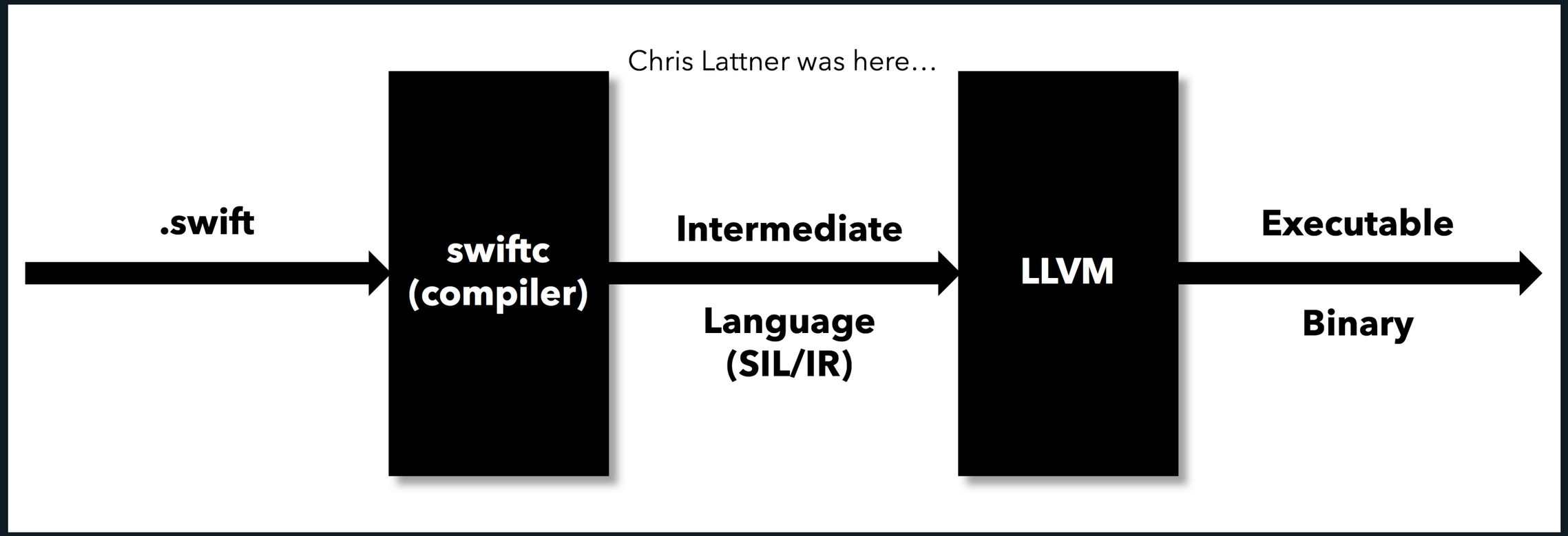
— What have we learned along the way?

How do we get there?

Let's crack open the black box



Black box first level of detail



Two key aspects of Swift for IoT

1. The Swift compiler (swiftc) generates intermediate language (IL) code.
   - **SIL** is platform independent.
   - **IR** is consumed by LLVM to produce a target-specific binary
2. Once in IR, we can combine with code from other languages:
   - Like C from Clang
   - And Rust from `rustc`

From traditional to IoT:

hello.swift:

```
print("hello, world")
```

```
$ swift hello.swift
hello, world
```

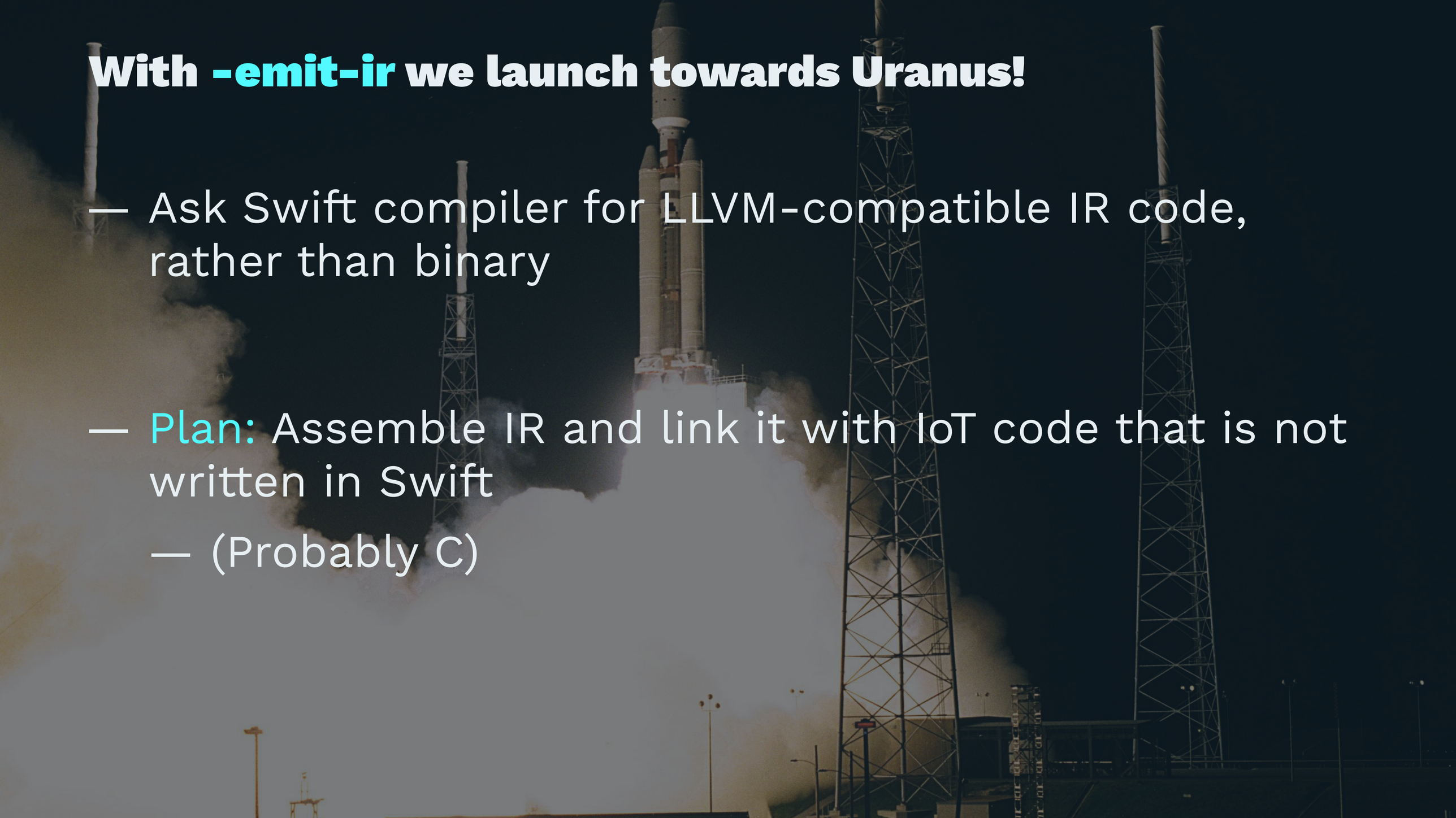
```
$ swiftc -emit-ir hello.swift > hello.ll
```

With -emit-ir we launch towards Uranus!

— Ask Swift compiler for LLVM-compatible IR code, rather than binary

— Plan: Assemble IR and link it with IoT code that is not written in Swift

  — (Probably C)



$ swiftc -emit-ir hello.swift > hello.ll

```
; program entry point
define i32 @main(i32, i8**) #0 {
entry:
...
  ; put string literal "hello, world" into String object
  %10 = call { i64, i64, i64 }
    @_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1__SS(
      i8* getelementptr inbounds ([13 x i8], [13 x i8]* @0, i64 0, i64 0),
      i64 12, i1 true)
...
  ; call print() function
  call void @_TFs5printFTGSaP__9separatorSS10terminatorSS_T_(
    %swift.bridge* %5, i64 %17, i64 %18, i64 %19, i64 %21, i64 %22, i64 %23)
  ret i32 0
}
```

$ swiftc -emit-ir hello.swift > hello.ll

```
; program entry point
define i32 @main(i32, i8**) #0 {
entry:
...
  ; put string literal "hello, world" into String object
  %10 = call { i64, i64, i64 }
    @_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1__SS(
      i8* getelementptr inbounds ([13 x i8], [13 x i8]* @0, i64 0, i64 0),
      i64 12, i1 true)
...
  ; call print() function
  call void @_TFs5printFTGSaP__9separatorSS10terminatorSS_T_(
    %swift.bridge* %5, i64 %17, i64 %18, i64 %19, i64 %21, i64 %22, i64 %23)
  ret i32 0
}
```

$ swiftc -emit-ir hello.swift > hello.ll

```
; program entry point
define i32 @main(i32, i8**) #0 {
entry:
...
  ; put string literal "hello, world" into String object
  %10 = call { i64, i64, i64 }
    @_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1__SS(
      i8* getelementptr inbounds ([13 x i8], [13 x i8]* @0, i64 0, i64 0),
      i64 12, i1 true)
...
  ; call print() function
  call void @_TFs5printFTGSaP__9separatorSS10terminatorSS_T_(
    %swift.bridge* %5, i64 %17, i64 %18, i64 %19, i64 %21, i64 %22, i64 %23)
  ret i32 0
}
```

$ swiftc -emit-ir hello.swift > hello.ll

```
; program entry point
define i32 @main(i32, i8**) #0 {
entry:
...
  ; put string literal "hello, world" into String object
  %10 = call { i64, i64, i64 }
    @_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1__SS(
      i8* getelementptr inbounds ([13 x i8], [13 x i8]* @0, i64 0, i64 0),
      i64 12, i1 true)
...
  ; call print() function
  call void @_TFs5printFTGSaP__9separatorSS10terminatorSS_T_(
    %swift.bridge* %5, i64 %17, i64 %18, i64 %19, i64 %21, i64 %22, i64 %23)
  ret i32 0
}
```

Challenge #1: Cross-compiling target issues

— We've encountered our first challenge

— By default, the Swift compiler emits IR for the host platform. We are not x64 running macOS, so...

```
$ swiftc -target armv7-apple-ios9.0 -emit-ir
hello.swift > hello.ll
```

```
; ModuleID = '-'
source_filename = "-"
target datalayout = "e-m:o-p:32:32-f64:32:64-v64:32:64-v128:32:128-a:0:32-n32-S32"
target triple = "armv7-apple-ios9.0"
```

Why armv7-apple-ios9.0?

Next, compile to an object file

Once we have IR, we use LLVM directly to compile to a binary object file.

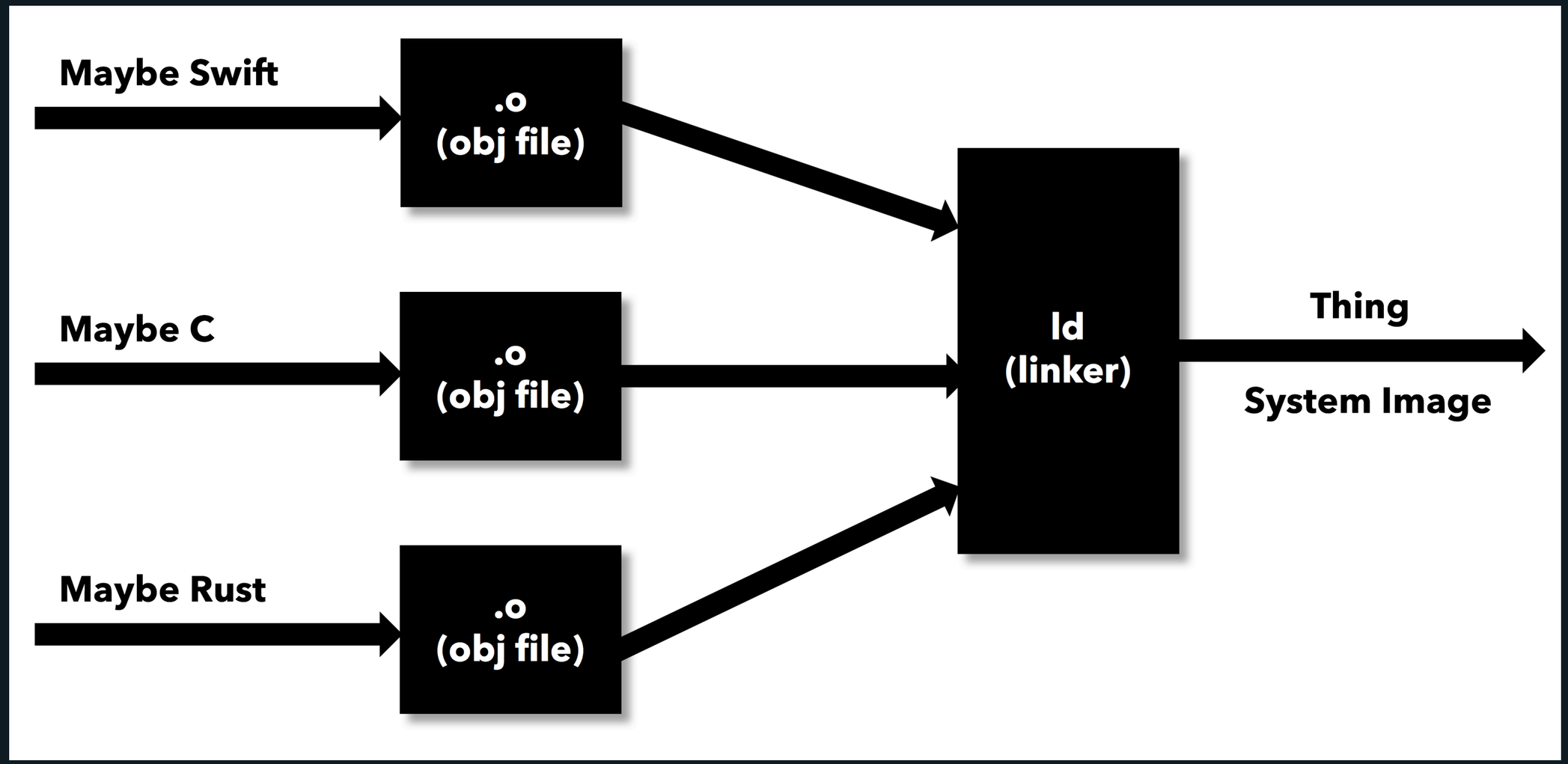
```
$llc -mtriple=thumbv7-none-eabi -mcpu=cortex-m3
 -filetype=obj hello.ll
$ ls -al *.o
```

Yields an object file:

```
-rw-r--r-- steven.gray  1868 Feb  5 14:30 hello.o
```

Challenge #1 sorted. Cool.

What's our black box look like now?



Adding Swift object files into an IoT image

Let's modify a boilerplate SensorTag makefile, adding our Swift module:

```
cc26xx/Makefile:
LDFLAGS += hello.o
```

```
$ make BOARD=sensortag/cc2650 cc26xx-demo
```

This yields...

...no joy (lots of errors).

```
# segment errors
../arm-none-eabi/bin/ld: cc26xx-demo.elf section '.ARM.extab' will not
  fit in region 'FLASH_CCFG'
../arm-none-eabi/bin/ld: region 'FLASH_CCFG' overflowed by 36 bytes

# undefined references
hello.o: In function 'main':
hello.ll:(.text+0x12): undefined reference to
  '_TFs27_allocateUninitializedArrayurFBwTGSax_Bp_'
hello.ll:(.text+0x18): undefined reference to '_TMSS'
hello.ll:(.text+0x22): undefined reference to '_TMSS'
hello.ll:(.text+0x2a): undefined reference to
  '_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1(void) static'

# standard C library dependencies
../arm-none-eabi/lib/thumb/v7-m/libc.a(lib_a-abort.o): In function 'abort':
abort.c:(.text.abort+0xa): undefined reference to '_exit'
../arm-none-eabi/lib/thumb/v7-m/libc.a(lib_a-signalr.o): In function '_getpid_r':
signalr.c:(.text._getpid_r+0x0): undefined reference to '_getpid'
```

...no joy (lots of errors).

```
# segment errors
../arm-none-eabi/bin/ld: cc26xx-demo.elf section '.ARM.extab' will not
  fit in region 'FLASH_CCFG'
../arm-none-eabi/bin/ld: region 'FLASH_CCFG' overflowed by 36 bytes

# undefined references
hello.o: In function 'main':
hello.ll:(.text+0x12): undefined reference to
  '_TFs27_allocateUninitializedArrayurFBwTGSax_Bp_'
hello.ll:(.text+0x18): undefined reference to '_TMSS'
hello.ll:(.text+0x22): undefined reference to '_TMSS'
hello.ll:(.text+0x2a): undefined reference to
  '_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1(void) static'

# standard C library dependencies
../arm-none-eabi/lib/thumb/v7-m/libc.a(lib_a-abort.o): In function 'abort':
abort.c:(.text.abort+0xa): undefined reference to '_exit'
../arm-none-eabi/lib/thumb/v7-m/libc.a(lib_a-signalr.o): In function '_getpid_r':
signalr.c:(.text._getpid_r+0x0): undefined reference to '_getpid'
```

Next Challenge: Standard C library issues.

— The CC2650 does not provide a typical standard C library. Not enough room. No concept like standalone processes on which to call functions like `getpid()`.

— So, let's build a library instead of a standalone process.

```
$ swiftc -parse-as-library -target armv7-apple-ios9.0
  -emit-ir hello.swift > hello.ll
```

`-parse-as-library` compiler option

— `-parse-as-library` tells the compiler we're not attempting to create a standalone executable.

  — Removes the process-related undefined errors.

However, that gives us a new error:

```
hello.swift:1:1: error: expressions are not allowed
  at the top level
```

No problem. Redefine hello.swift.

```
func print_hello() {
  print("hello, world")
}
```

-This compiles fine into `hello.o`.
-In fact, rerunning Make, combining Swift and C code,
now returns no errors!

— Why?

If we're ever going to do anything useful,
we must call our Swift function from C

- Swift, like C++, **mangles** its names
- So `print_hello` is not simply known as `print_hello` to the linker
- It's known as `_TF5hello11print_helloFT_T_`.
- How do we know this?
  - Look inside hello.ll (the IR file).

(Minor) Challenge: Dealing with name mangling issues

— Let's cheat a bit here and hide the name mangling...

```
#define print_hello _TF5hello11print_helloFT_T_
extern void print_hello();
```

— Consider this just syntatic sugar.

Call Swift when user presses the SensorTag's left button

```
inside event loop...

if(event_data == CC26XX_SENSOR_1) {
  printf("Left button pressed\n");

  print_hello();

...
```

Then rerun Make.

Errors are back. That's actually a good sign.

```
# undefined references
hello.ll:(.text+0x12): undefined reference to
  '_TFs27_allocateUninitializedArrayurFBwTGSax_Bp_'
hello.ll:(.text+0x18): undefined reference to '_TMSS'
hello.ll:(.text+0x22): undefined reference to '_TMSS'
hello.ll:(.text+0x2a): undefined reference to
  '_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1(void)'
```

Errors are back. That's actually a good sign.

```
# undefined references
hello.ll:(.text+0x12): undefined reference to
  '_TFs27_allocateUninitializedArrayurFBwTGSax_Bp_'
hello.ll:(.text+0x18): undefined reference to '_TMSS'
hello.ll:(.text+0x22): undefined reference to '_TMSS'
hello.ll:(.text+0x2a): undefined reference to
  '_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1(void)'
```

— We must now consider:

  — Arrays

  — Strings

Strings

```
hello.ll:(.text+0x2a): undefined reference to
  '_TFSSCfT21_builtinStringLiteralBp17utf8CodeUnitCountBw7isASCIIBi1(void)
  static'
```

— What is this?

Decoding a managled name

— There are string tokens in the mangled name that might be useful:

1. `_builtinStringLiteral`
2. `utf8CodeUnitCount`
3. `isASCII`

`grep` through the Swift source

— Searching the Swift source code finds the result in `String.swift`:

```
extension String : _ExpressibleByBuiltinStringLiteral {
  public init(
    _builtinStringLiteral start: Builtin.RawPointer,
    utf8CodeUnitCount: Builtin.Word,
    isASCII: Builtin.Int1) ...
```

— What is this?

`String.init()`

— This is an **extension method** within the Swift standard library which instantiates a `String` object from a string literal. For example:

```
let foo = "hello, world" // foo is String
```

— Standard Swift library?? Does our SensorTag have that?

  — No, it does not.

— Are we done? Never!

The Swift Standard library (as you probably know…)

- Contains lots of core functionality, including:
  - Fundamental data types such as String
  - Common data structures such as Array
  - Global functions such as print()
- To build `"hello, world"` in Morse, we likely need all of that

(Major) Challenge: Design dilemma

Three options:

1. Build `"hello, world"` without Strings, Arrays, and the print() method
2. Pull in bits and pieces of the Swift Standard library source and cross-compile (top-down approach)
3. Build a replacement standard library specific for Internet of Things devices (bottom-up approach)

Option 1: No Standard library dependencies. This works!

```
// actual, working code on CC2650
public func timerTick(val: UInt32) -> UInt32 {
  let freq: UInt32 = 1 // tenths of a second
  switch val {

  // 'h' == ....
  case 0:             // dot
    led_onoff(1)
  case freq:          // pause
    led_onoff(0)
  case 2 * freq:      // dot
    led_onoff(1)
  case 3 * freq:      // pause
    led_onoff(0)
  ...
```

Problem

— You'd be hard pressed to call this Swift code
— More like "Swifty macro assembly language" (SMAL)

Option 2: Pull in pieces of the Swift Standard library

- **This does not work well**
  - (trust me)

- Standard library code assumes it's on a machine with plenty of memory, CPU power, etc.
- Using even a small piece creates a hefty dependency tree
  - `Strings` and `Arrays` pull in thousands of lines of related code
- Just not practical for IoT devices

Option 3: A custom, miniature standard library for IoT

- A miniature library would have Swift features provided as makes sense for Things
  - e.g., Fixed-size String and Array.

- But no Unicode support.
- No ArraySlice.
- No…lots of stuff that easily swamps 20 KB of RAM.

Tradeoffs

- With a custom standard library, you can do straight Swift ports to IoT
  - But straight ports is a non-goal
- Key goal is a Swift-based ecosystem
  - Light-resource-use Swift code for Things
  - Traditional-resource-use for devices higher in the stack

So, is a custom "standard" Swift library even possible?

— Yes!

— MicroStdlib shows the way...

— The key is yet another command-line argument that informs the Swift compiler we are providing our own standard library.

```
$ swiftc -parse-stdlib ...
```
[4]

[4] Lots of details omitted here

Where we are

- We've proven to ourselves that running Swift on Things is technically possible.
- We've determined the need for a bottom-up, miniature standard library to better host Swift code on Things.
- We will open source this library on GitHub when it's ready.
- Contact us or see our website for updates or if you'd like to help.

  — www.sftsrc.com

Swift on IoT: Questions to be Answered

— Is it possible to go there?

  — Yes.

— How do we get there?

  — Via a meandering path.

— What have we learned along the way?

What have we learned along the way?

- Swift on IoT is possible
  - It was fun researching how to do this
    - Cannot simply Google your way to an answer
    - That joy of figuring it out for ourselves
    - For every roadblock, there was a workaround (so far)

Why even make the attempt?

- Because the nobody else is trying this
  - (There's one photo of Uranus...)
- We don't yet know justifications for this exploration
- But...

We might just find that Swift on IoT rains diamonds, just like it does on Uranus[5]

[5] http://www.astronomy.com/news/2017/08/researchers-recreate-diamond-rain

Farewell for now Uranus

Thank you.

Steven Gray
SoftSource Consulting
steven.gray@sftsrc.com

Image credits

- Apple II - Rama - https://commons.wikimedia.org/wiki/File:Apple_II_IMG_1107.jpg
- Safari Logo - Oorehov - https://commons.wikimedia.org/wiki/File:Safariapplelogo.png
- SensorTag background image - http://43oh.com/2015/06/getting-to-know-the-cc2650-sensortag-v2
- SensorTag-blueprint - https://e2e.ti.com/blogs_/b/connecting_wirelessly/archive/2015/06/03/create-develop-prototype-3d-print-your-iot-idea-with-simplelink-sensortag
- Earth - https://pixabay.com/p-1617121
- Mars - NASA.gov
- Saturn - NASA.gov
- Uranus - https://commons.wikimedia.org/wiki/File:Uranus_(Edited).jpg
- Get on with it - Monty Python and the Holy Grail, youtube.com
- Cassini launch - NASA.gov