Porting OCaml to nRF52

OCaml on bare-metal devices using RIOT OS and OMicroB

Ben Andrew
@benmandrew
ba405@cam.ac.uk

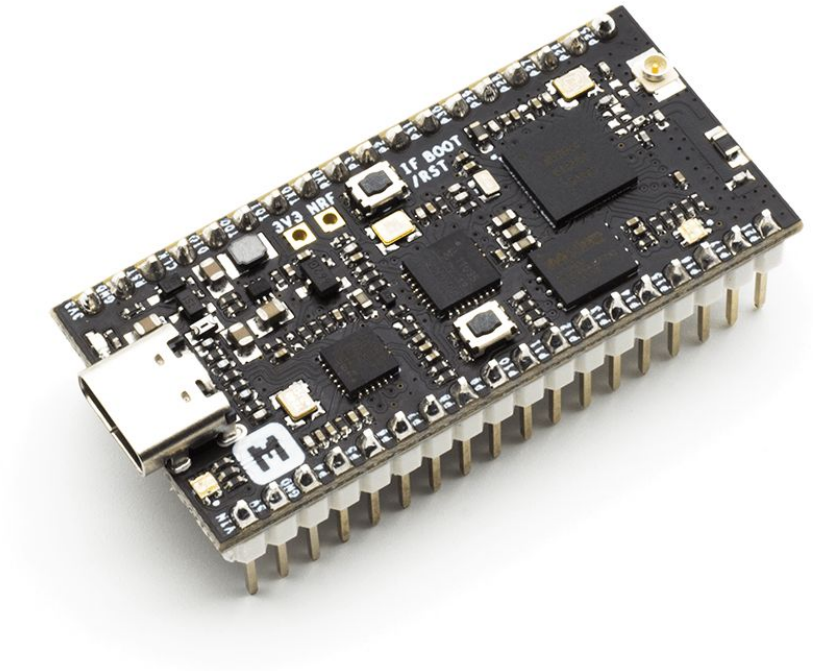
Tarides

The Board - nRF52840

ARM Cortex-M4 processor

1MB of flash memory

256KB of RAM



Running OCaml

```
let () =
 Printf.printf "Hello World!"
 ;;
```

```
0x00000054, 0x000002df,
0x00000000, 0x00000057,
0x000f0001, 0x00000010,
0x00000013, 0x0000001c,
0x00000025, 0x0000002e,
0x00000037, 0x00000040,
...
```

Compiling OCaml - Native, ocamlopt

OCaml can be compiled to native code for the target device, run like any other standalone program.

Choosing native would require writing a backend to the compiler that targets the architecture. This sounds difficult.

Compiling OCaml - Bytecode, ocamlc

Bytecode is a portable instruction set, interpreted by an OCaml bytecode interpreter.

With bytecode, we only need to compile the interpreter to run on the device; then the portable bytecode can be loaded in as a simple array of data.

The Runtime

Implements the interpreter, exceptions, garbage collection, stacks, etc.

```
Instruct(GETVECTITEM):
  accu = Field(accu, Long_val(sp[0]));
  sp += 1;
  Next;
Instruct(SETVECTITEM):
  caml_modify(&Field(accu, Long_val(sp[0])), sp[1]);
  accu = Val_unit;
  sp += 2;
  Next;
```

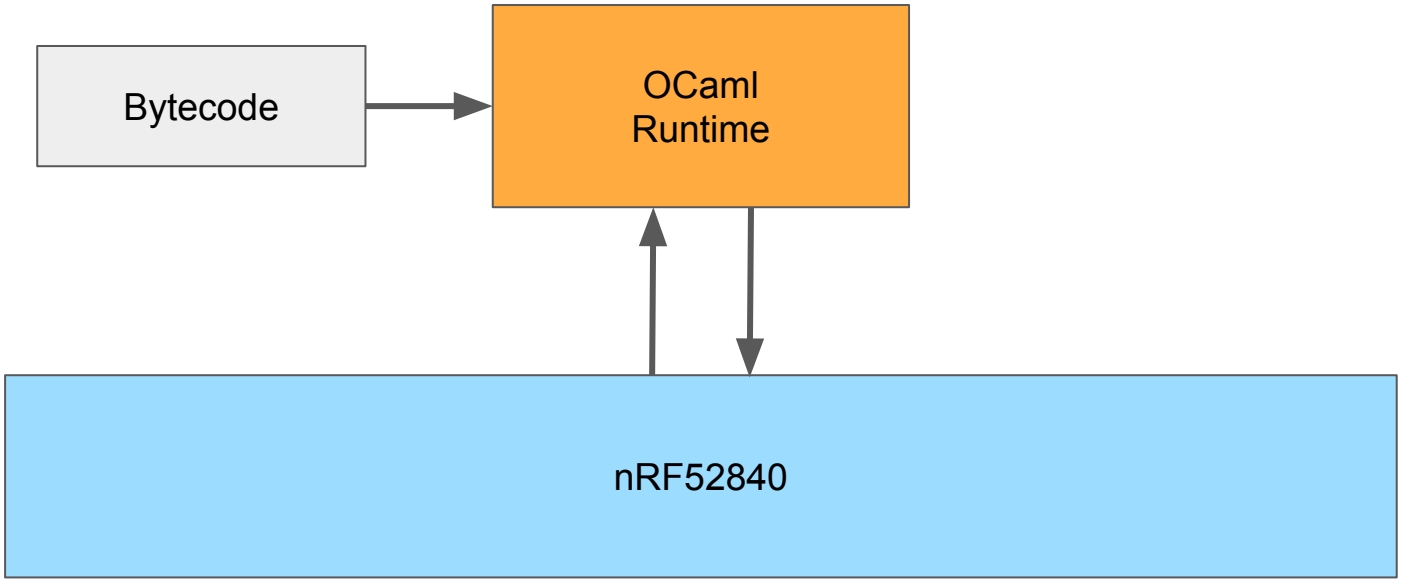
C Calls

The bytecode interacts with the runtime through the C calling interface.

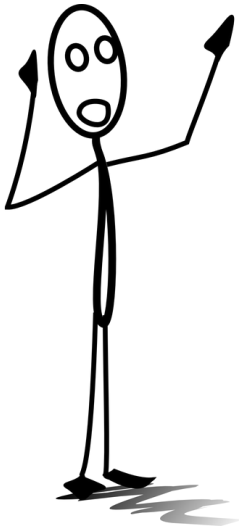
These C functions are called ‘primitives’

```
Instruct(C_CALL1):
  Setup_for_c_call;
  accu = Primitive(*pc)(accu);
  Restore_after_c_call;
  pc++;
  Next;
```

Overview

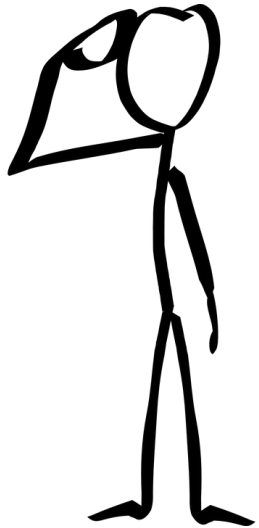


Quel heure est-il ?



OCaml

What?



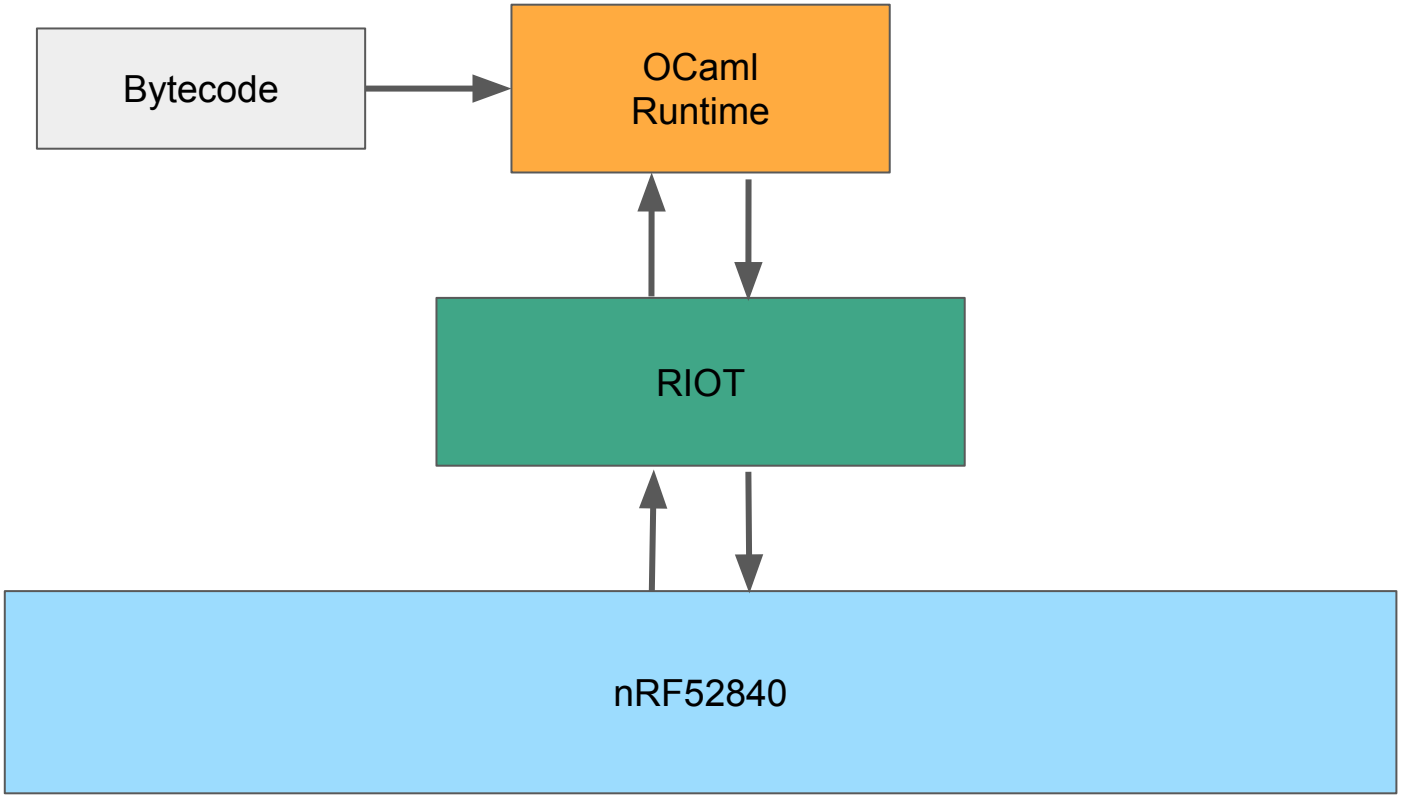
nRF52

The RIOT operating system

An energy-efficient microkernel for interfacing with the board.

Does the background work of initialising the board and CPU, exception handling, threading, etc.

Also provides a robust build system for generating flags, compiling, and linking to create an image that can be flashed to the device.


Bytecode
OCaml
Runtime
RIOT
nRF52840

The Build System

Given the board, RIOT works out all of the required compiler and linker flags needed, as well as C files, linker scripts, etc etc.

RIOT automatically pulls in C files from an ‘application’ directory, so we can compile our ML files to bytecode, then dump to a C file containing arrays of data. RIOT then compiles this and includes it in the linking step.

The OCaml runtime is compiled using the same ARM GCC, and packed into a static library (libcamlrun.a). This is then injected into RIOT’s linking step so that the runtime is included in the final image.

OCaml to bytecode in C

```
let main =
  10 + 5
  ;;
```

```
ocamlc main.ml -output-obj -o main_caml.c
```

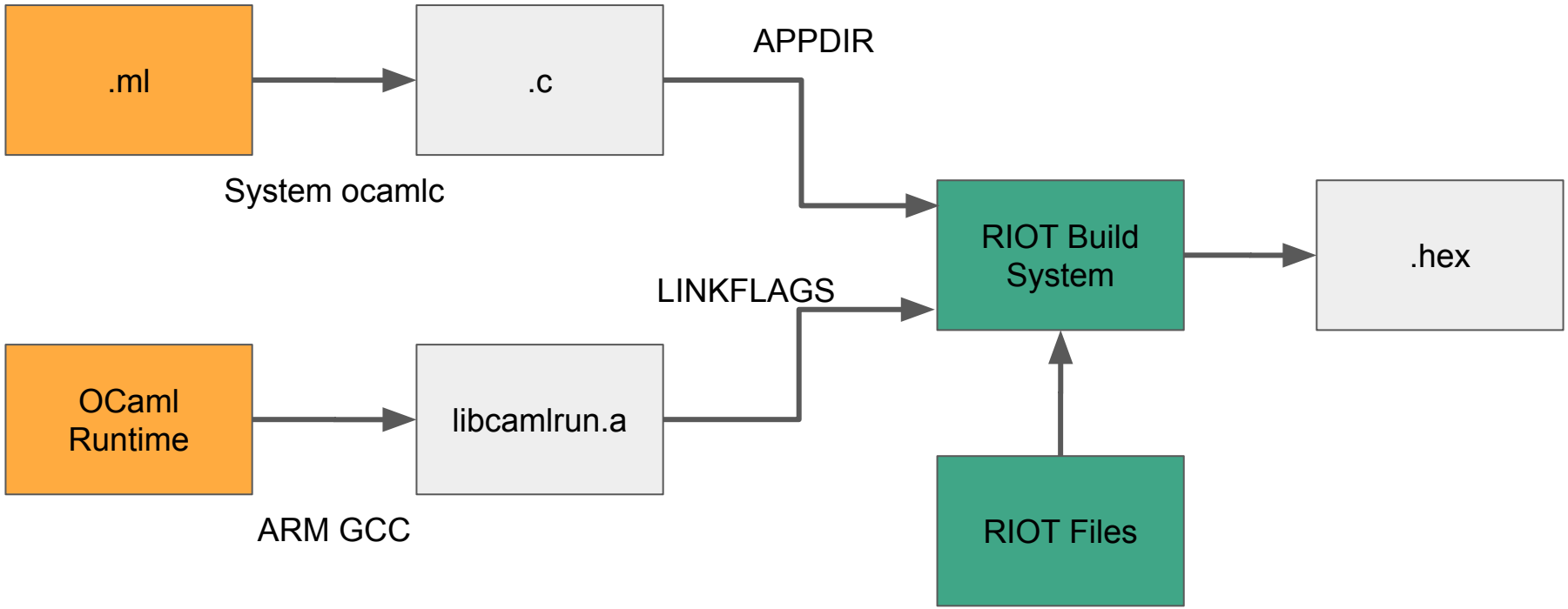
C File Structure

```
static int caml_code[] = { ... }
static char caml_data[] = { ... }
static char caml_sections[] = { ... }

caml_startup_code(
  caml_code, sizeof(caml_code),
  caml_data, sizeof(caml_data),
  caml_sections, sizeof(caml_sections),
  /* pooling */ 0,
  argv);
```

C File structure cont.

```
extern value caml_abs_float();
extern value caml_acos_float();
extern value caml_add_debug_info();
...
primitive caml_builtin_cprim[] = {
  caml_abs_float,
  caml_acos_float,
  caml_add_debug_info,
  ...
}
```

Building


.ml
.c
APPDIR
System ocamlc
RIOT Build System
.hex
LINKFLAGS
OCaml Runtime
libcamlrun.a
ARM GCC
RIOT Files

A Test Program

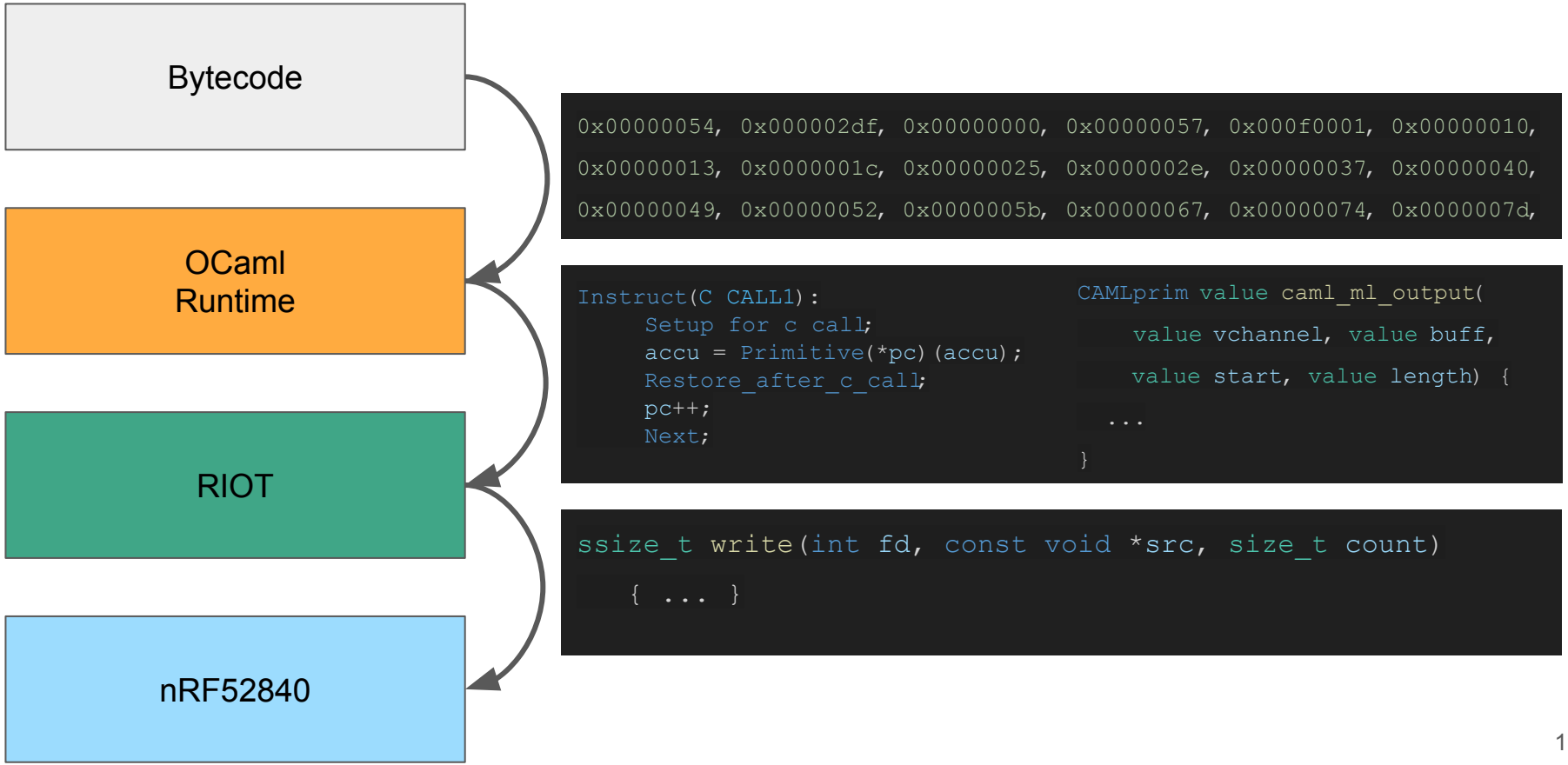
```
let () =
  print_string "Hello World!\n";
  flush stdout
```

It Works!

```
let () =
  print_string "Hello World!\n";
  flush stdout
```

2021-09-24 16:11:47,150 # main(): This is RIOT!
2021-09-24 16:11:47,159 # Hello World!

A Quick Recap - `print_string "Hello World!\n";`



Issues with using the standard OCaml runtime

Very large image produced, a sizeable program can easily exceed the flash memory.

The program from before links into an image 435Kb in size.

```
let () = Printf.printf "Hello World!"
```

This program produces an image 743Kb in size.

Remember there is only 1Mb of flash!

Module Instantiation

```
let () = Printf.printf "Hello World!"
```

2021-09-24 16:07:38,487 # main(): This is RIOT!
2021-09-24 16:07:38,541 # Fatal error: not enough memory
2021-09-24 16:07:38,543 # #! exit 1: powering off

OMicroB to the rescue

A project that allows you to run OCaml bytecode on extremely memory-constrained devices (2.5Kb of RAM!)

It does this with a **custom runtime**, as well as doing multiple static passes over the bytecode itself to reduce its size.

Targets the **PIC32** and **AVR** microcontrollers.

Two Nice Optimisations

1. The bytecode is cleaned of redundant information using **ocamlclean**. This includes C primitives.

2. The interpreter is stripped of unused bytecode instructions.

The Main Optimisation

OMicroB does a ‘simulation’ of the bytecode interpreter. It has a virtual stack, heap, accumulator, program counter, etc. and runs the bytecode on your computer.

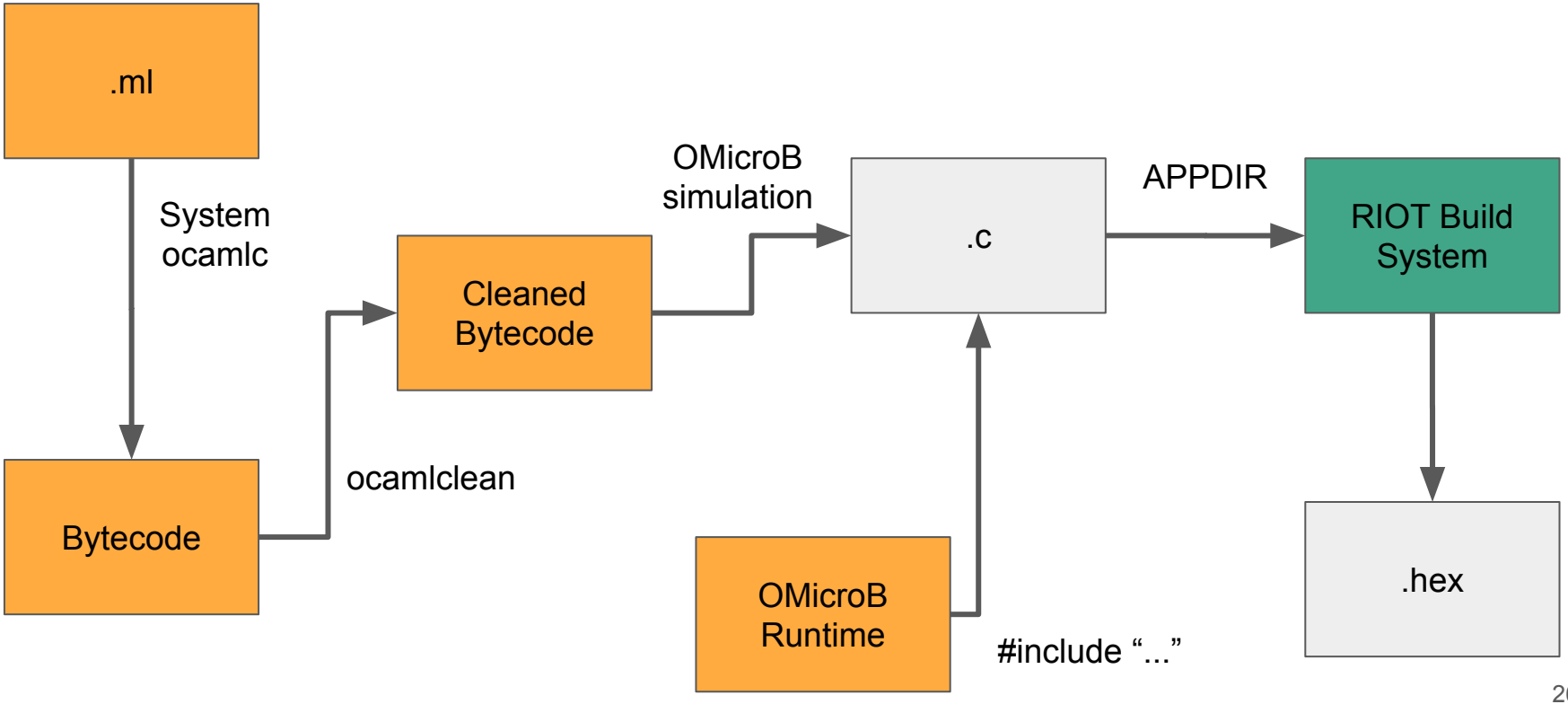
It does this up until the first IO, at which point the program actually has to be run on the device. It then dumps the state of the runtime in a sort of ‘screenshot’ into a C file, similarly to how ocamlc did before.

```
value ocaml_stack[OCAML_STACK_WOSIZE];
value ocaml_ram_heap[OCAML_STATIC_HEAP_WOSIZE + OCAML_DYNAMIC_HEAP_WOSIZE];
value ocaml_ram_global_data[OCAML_RAM_GLOBDATA_NUMBER];
PROGMEM value const ocaml_flash_heap[OCAML_FLASH_HEAP_WOSIZE] = { ... }
...
```

The Main Optimisation

```
PROGMEM opcode_t const ocaml_bytecode[OCAML_BYTECODE_BSIZE] = {
 /* 0 */  OCAML_STOP
};

#include "src/byterun/vm/runtime.c"

PROGMEM void * const ocaml_primitives[OCAML_PRIMITIVE_NUMBER] = {
};
```

Building with OMicroB



OMicroB's Limitations

OMicroB is almost too optimised. It does not have a lot of runtime functionality that you might want, such as pipe-based IO for terminal output.

You are able to add external C functions however, so these can be implemented yourself.

```
external length : 'a array -> int = "caml_array_length"

...

CAMLexport mlsize_t caml_array_length(value array) { ... }
```

The Point I Reached

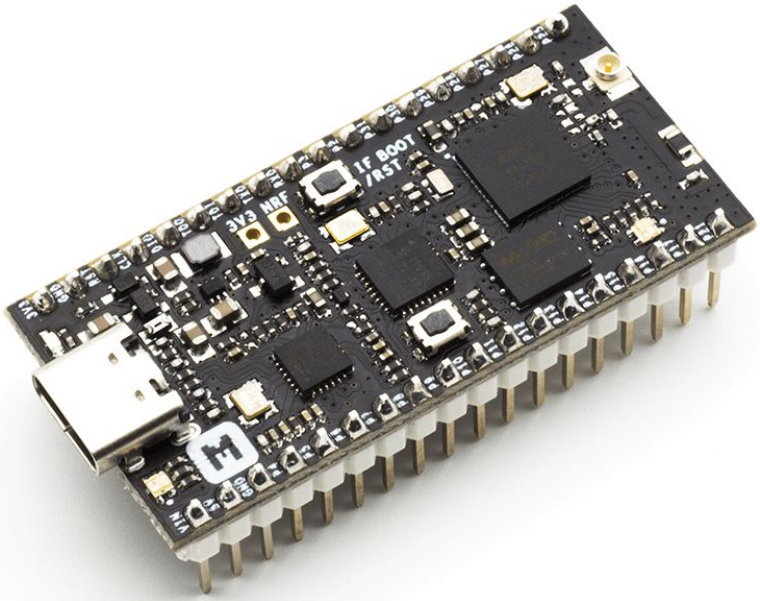
```
Error: File interp.ml, line 194, characters 4-10: Assertion failed.
```

Plans for the Future

```
external read_sensor : int -> float = "caml_read_sensor"

...

CAMLexport value caml_read_sensor(value i) { ... }
```

Plans for the Future

```
BOARD = nrf52840-mdk

USEMODULE += xtimer
```



Acknowledgements

- Magnus Skjegstad, Lucas Pluvinage, Romain Calascibetta
- The RIOT Team
- The OMicroB Team
- And everyone at Tarides

GitHub Project Link: https://github.com/benmandrew/omicrob-riot-nrf52