Porting OCaml to nRF52

nRF52840

The board has an ARM Cortex-M4 processor, 1 MB of flash memory, and 256kb of RAM.

Specsheet

Memory Ranges

- Flash ranges from 0x00000000 to 0x10000000
- RAM ranges from 0x20000000 to 0x20004000

Setup

ARM Toolchain

You need to install the ARM embedded toolchain, linked here. This contains required tools for development such as a GCC compiler/linker, GDB debugger, etc.

The arch triplet is *arm-none-eabi*, with tools specified with this prefix, for example *arm-none-eabi-gcc* and *arm-none-eabi-gdb*. The toolchain's bin directory should be added to PATH.

pyOCD

This is a programmer for the board, and helps with debugging. A tutorial is linked here.

RIOT

What does RIOT do?

RIOT provides a base layer 'OS' to interact with the device, through a library-style interface. It also provides a build system to compile this base layer with your C application. This build system (using Make) does the heavy lifting of detecting required compilers, flags, and RIOT code needed to be included in the final image.

A large benefit is that the vast majority of the RIOT build system does not depend on your application, and instead on the device you're targeting. This gives some nice encapsulation between the stages of our eventual build system.

You do need to manually specify which RIOT 'modules' you want to include in the Makefile, which give you access to hardware such as timers or GPIO, but this likely won't be changed often in application development.

How is a binary image built?

1. The necessary compiler and linker flags are worked out. Include directories, ARM-specific flags, enabling and disabling errors, etc.

2. The required RIOT source files are compiled to object files. This will be core RIOT code, board-specific code, and cpu-specific code. For example in this case the board is the NRF52840-mdk and the CPU is the ARM Cortex-M4.

3. The source of any requested modules will be compiled.

4. All C files in the APPLICATION directory will be compiled to object files; this is your application.

5. The compiler will then be called to link together all of these object files into a single ELF file.

6. This ELF file will be translated into a HEX file.

In development I found the HEX file unnecessary, as all the board programmers I used did the conversion themselves when given the ELF file. But it’s probably useful for later.

OCaml Runtime

The standard OCaml runtime is written entirely in C, and is compiled to a static library called *libcamlrun.a*. Usually this is used in the OCaml build system to generate the *ocamlrun* tool for running OCaml bytecode, read from a file via STDIN, but it also provides an entrypoint for bytecode stored as simple C arrays, passing the pointers to the various required arrays through a function call.

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

These arrays can be generated by the ocamlc compiler with the call:

```
ocamlc -g -custom -output-obj -o main.c main.ml
```

Also generated is the table of pointers to caml primitives, C functions implementing lower-level functionality that can be called by the OCaml bytecode interpreter. The implementations of these functions are contained in the *libcamlrun.a* runtime. A large number of the problems I ran into in the project involved function implementations missing from this file.

Similarly to RIOT, the building of *libcamlrun.a* can be done independently of the rest of the build system. It does however need the compiler flags generated by RIOT's system, which leads to the strange layout of the main Makefile; I still don't really understand the control flow of Makefile evaluation and while it works at the moment, it breaks frequently.

Directory Layout

```
ocaml-riot-nrf52
|-> ocaml-nrf52
    |-> example
    |-> ocaml
        |-> runtime
    |-> bin
```

The `ocaml-riot-nrf52` directory contains all of the RIOT-specific things, as well as the `ocaml-nrf52` folder, which contains the root makefile we start from and the application directory (`example`). Once the Makefile is invoked, we create the `bin` folder as well as copying the complete OCaml source from the opam switch into `ocaml`.

It'd probably be nicer just to have the parts of `ocaml-src` that are needed in the repo, but this is a remnant of Lucas' OCaml ESP32 project.

Injection into RIOT's build system

RIOT very nicely compiles and links all of the C files in the application directory, but it is good to build static library separately. In this case there is no builtin way to link in the static library, so we have to delve into RIOT's internals a little.

In RIOT's *Makefile.include* we can add *libcamlrun.a* to the LINKFLAGS variable, also taking this opportunity to link in a maths library with *-lm*.

```
LINKFLAGS += $(OCAMLDIR)/runtime/libcamlrun.a -lm
```

Then when RIOT links all the other object files together with its fancy system, the OCaml runtime will be snuck in too!

Obviously then we must invoke OCaml's build system before RIOT's.

This is a remnant of OCaml's build system, in which the runtime is built as a static library to be reused multiple times. It would be better to simply put the C files in the application directory so that RIOT compiles them and pulls them into the linking step. This would not only remove the circular dependency, but remove all of the OCaml-specific build system entirely.

Ordering of build system invocations

We now have a bit of a circular dependency; to build the OCaml runtime we need the compiler flags generated by the RIOT build system, but to fully run the RIOT build system we need the OCaml runtime fully built as a static library.

As stated above, I don't understand the Makefile's control flow but I seem to have it organised so it works. However it is completely unreadable, even to me.

Problems with using OCaml's runtime

One of the first things the standard runtime does when running a program is import the required modules. The problem with this is that it is very memory intensive: while hundreds of kilobytes is nothing on a desktop, the microcontroller instantly runs out of memory and crashes.

As an example, this program crashes with an out of memory exception,

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

while this one does not, and functions as expected due to the functions used being builtin.

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

If even the simple *Printf* module cannot be used, writing non-trivial programs would be impossible.

OMicroB Runtime

We can alternatively use the runtime of OMicroB, a project specifically designed to get OCaml bytecode running on extremely low memory boards such as the AVR Atmega32u4, which has 2.5kb of ram. While it only targets the AVR and PIC32 architectures, if it can run on boards with literally 100 times less RAM than the NRF52840 then porting it to this board should be worth it.

Link to Paper

Directory Layout

```
omicrob-riot-nrf52
|-> omicrob
    |-> example
    |-> ocaml
    |-> bin
    |-> src
        |-> bc2c
        |-> byterun
        |-> omicrob
```

The `omicrob-riot-nrf52` directory contains all of the RIOT-specific things, as well as the `omicrob` folder, which contains the application directory (`example`), the ocaml runtime (`ocaml`), and the sources for OMicroB's subprograms (`src`)

OMicroB's differences to the OCaml runtime

OMicroB reduces memory usage through multiple techniques.

Bytecode cleaning

OMicroB uses the *ocamlclean* tool to remove redundant information from the bytecode.

Simulation

To get around the memory-intensive module instantion, OMicroB simulates the execution of the bytecode program in an interpreter on the host computer, complete with a simulated stack and heap where intermediate values are stored. It runs the program up until the first I/O, at which point the program cannot be run any further without the intended meaning being correct when run on the device.

At this point the bytecode and intermediate data structures (stack, heap, accumulator, etc.) are dumped into a C file, somewhat similarly to how it was done above with the standard OCaml runtime.

Interpreter cleaning

At this point OMicroB can see the bytecode, and knows which instructions will be used. It uses this information to strip out the implementations of instructions in the interpreter that will never be encountered, reducing the size of the runtime.

OMicroB's Dependency Issues

OMicroB depends on *ocamlclean* (as above) and *obytelib*, however the versions of these dependencies that opam provides are not natively compatible with OCaml 4.12. For example *ocamlclean* requires *ocaml* >= 4.07.0 && < 4.08.0 and *obytelib* requires *ocaml* >= 4.07.0.

I updated both projects as described below and used opam to pin the local repositories.

Updated ocamlclean and obytelib repo links. These DO NOT WORK CORRECTLY.

Updating *ocamlclean*

ocamlclean has a stupid magic number checking system that checks whether the magic number in the file is *explicitly equal* to the specified one. There is currently no way to have any sort of range of versions, you get one. Thus we change the magic string from Caml1999X026 to Caml1999X029.

There is an environment cleaning step in *ocamlclean* that does not work with 4.12 bytecode. While the memory saved by doing it is reasonable, I don't know enough about the internals of the language to fix it so I simply commented it out.

Updating *obytelib*

obytelib is simpler in that we only need to update the accepted magic number to take 4.12 bytecode files. It has a much better implementation than *ocamlclean*, pattern matching against the version number to change the behaviour according the the file structure. Thus we just need to add V029 to the Version type, and add this case to every pattern match in the source. This is very easily implemented by in most places simply falling through to the V023 case, which of course will cause issues if the bytecode structure differs between these versions. I haven't yet run into issues, but they could be there.

Would be great if there was some file format documentation for these files online.

C file structure

The structure of the C files outputted by OMicroB differs significantly from those of ocamlc, in that there are explicit flash and RAM data regions, as well as the stack and heap being dumped instead of being implicitly empty and initialised on device by the runtime. The accumulator and program counter are also included.

The OCaml runtime has no entrypoints that take this intermediate 'screenshot' and resume from it. As well as this the opcodes correspond to the stripped OMicroB interpreter described above, so would not be interpreted correctly by the OCaml interpreter. It also appears that the OMicroB interpreter has some custom instructions that the OCaml one does not, such as expanding the single jump instruction into a jump with a single byte offset and one with a two byte offset; another way it saves memory.

While the OMicroB runtime is split into multiple C files, the 'linking' is done at compile-time with preprocessor #includes to simply paste all of the files into a single C file. This approach is interesting to say the least, and not very flexible. It caused me a couple of issues.

Closure Environments in 4.07 and 4.12

There is an incompatibility between the bytecodes of OCaml 4.07 and 4.12, specifically in the structure of closure environments. Unfortunately as stated before OMicroB, *ocamlclean*, and *obytelib* are written for 4.07 and so giving then 4.12 bytecode does not work. This is the current main issue with getting bytecode running on the board, as it prevents any and all functions from being called.

The Standard Library

When OMicroB invokes *ocamlc* to compile OCaml to bytecode, it specifies a CAMLLIB directory in which the standard library we're using is contained. This is to prevent the calling of a function that the bytecode thinks exists in the actual OCaml standard library but that does not exist in our limited OMicroB standard library.

However I don't really understand how this works, and the 4.07 to 4.12 issue prevented me from making progress with it.

Debugging on Device

Running the code

In separate terminals, after plugging in the device:

```
## List available serial ports
ls /dev/tty.*
## Open a terminal to device IO
./dist/tools/pyterm/pyterm -p '/dev/tty.usbmodem11102' -b '115200'
```

pyterm is a tool included with RIOT to open a terminal to the device. This call is made from RIOT's root directory.

The serial port (the *-p* argument) may be different in your case; choose the correct one according to the *ls* command above.

```
## Erase the flash of the chip
pyocd erase -t nrf52 --chip
## Flash the image onto the chip
pyocd flash -t nrf52 <file>.elf
```

If you don't erase the flash memory between flashes, the image will not be written successfully. The program will start to run immediately on flashing, so make sure to have the terminal open before it finishes!

Makefiles

The structure of the main Makefile is slightly weird, in that RIOT generates the compiler flags, jumps back out to compile ocaml and OMicroB, then jumps back in to RIOT to do the rest of the compilation and linking. This can cause the CFLAGS variable to duplicate itself, which causes problems (specifically with --specs=nosys.specs). Thus near the top of the file I deduplicate the flags.

Debugging the code

Again in separate terminals:

```
## starting the on-device debugger
pyocd gdbserver -t nrf52
```

This gives you the address of the device for debugging, in my case *localhost:3333*.

```
## start the ARM GDB
arm-none-eabi-gdb
## Set the target to the on-device GDB server
## Make sure the address matches the above
target remote localhost:3333
## Load the wanted ELF file into the local GDB
file <file>.elf
## Flash to the device
load
```

I'm not sure if it's required, but erasing the flash between debugging sessions may be useful.

I had an issue where GDB wasn't detecting the RAM memory region as addressable; this can be explicitly specified with this command in GDB:

```
mem 0x20000000 0x20040000 32 rw
```

It can be useful to move commands made for every GDB session to the *.gdbinit* file, which is executed on GDB startup.

Next Steps

Updating OMicroB and Dependencies

As described above, OMicroB, ocamlclean, and OByteLib need to be updated to OCaml 4.12.

RIOT Modules in OCaml

RIOT provides a module system to allow interfacing with peripheral devices like timers, GPIO, sensors, etc. We can connect these directly to corresponding OCaml modules so that the functionality is usable from OCaml.

```
external read_sensor : int -> float = "caml_read_sensor"
...
CAMLexport value caml_read_sensor(value i) {
   ...
   ret = read_sensor();
   ...
}
```

Here we have an OCaml module with an externally defined function 'read_sensor', implemented in C as an OCaml compatible function (only taking and returning OCaml 'value' types). This wrapper function converts the arguments into C types, calls the right RIOT function, then returns the result converted back into an OCaml value.

The first functionality that should be implemented should likely be pipe-based output to a terminal, which can be accessed via the RIOT *write* or *printf* function. I have a little bit of this in the *example* directory in *io.c* and *test.ml*.

Porting to Other Boards

Because RIOT takes care of all the flags, files, and scripts in the background, we simply specify the board in the makefile like so:

```
BOARD=nrf52840-mdk
```

This makes it very easy to target other boards, just changing one line. You must also check that the modules that you have selected are also supported by the board. This can be checked on the RIOT documentation website (https://doc.riot-os.org).

```
MODULES=xtimer
```