OCaml on the ESP32 chip: Well Typed Lightbulbs Await

Lucas Pluvinage, Sadiq Jaffer and Anil Madhavapeddy

1 Introduction

The ESP32[1] is a relatively new low-cost and energy efficient system on a chip with builtin WiFi and Bluetooth. It is a family of chips that all have a single- or dual-core Tensilica Xtensa LX6 microprocessor with a clock rate of up to 240 MHz. The ESP32 is aimed at embedded hardware usage and is thus equipped with builtin antenna switches, power amplifiers, low-noise receive amplifiers, filters and power management modules. The ESP32 achieves ultra-low energy consumption through power saving features including fine resolution clock gating, multiple power modes, and dynamic power scaling. It also includs an ultra low power coprocessor that allows ADC conversions, computation, and level thresholds while in deep sleep, thus permitting long battery lifetimes.

The ESP32 relies on a new instruction set architecture that requires a new code generation backend. The current SDK is a patched gcc, and LLVM has not yet been ported. In this talk, we will report on our successful port of OCaml and MirageOS to run on this new processor. We have successfully:

- Ported the bytecode OCaml runtime to boot and execute OCaml program.
- Written a new ESP32 native code backend that passes the “embedded” parts of the OCaml test suite.
- Booted MirageOS with its OCaml console, HTTP client and framebuffer on a ESP32 with a display module, in full native code.
- Written bindings to the Wifi APIs, permitting wireless connectivity from OCaml code to the outside world.

In this talk, we will outline some of the challenges related to porting OCaml to a new hardware architecture, some of our usecases behind the ESP32 family towards building safer embedded environments, and finally some approaches towards submitting our port upstream into the mainline OCaml distribution.

[1] `http://esp32.net`

2 Challenges

2.1 Bootstrap

Since the ESP32 chipsets are typically in embedded hardware with limited resources, they do not run a general purpose operating system like Linux. Instead they use a library operating system geared towards real time operation, such as FreeRTOS. This is a single-address space OS that exposes hardware functionality directly as libraries, with a libc-like layer bolted on top to facilitate compatibility with some applications.

Bootstrapping on this environment requires cross-compiling from a host general purpose OS, since an OCaml compiler will not run directly on the (rather slow) ESP32 environment. We did this in a few steps: firstly by porting the bytecode OCaml runtime, and then building a native code compiler.

Porting the bytecode runtime first is a useful exercise since it gets us familiar with the (often quirky) embedded toolchains. The ESP32 toolchain includes the newlib libc which we disabled, since our eventual goal was to run the MirageOS environment instead of a pseudo-libc. We then wrapped up the build steps in a Docker environment to make the boot process reproducible and documented[2].

Once the bytecode ran, we forked the OCaml codebase to add a new native architecture target to the ocamlopt compiler[3]. This native code backend directly emits ESP32 opcodes that can be linked along with the libasmrun OCaml native runtime into an object file that interfaces with the FreeRTOS hardware drivers. Eventually, these C hardware drivers could also be replaced with OCaml equivalents, as has been done for other MirageOS backends such as Xen. We first tested this using a qemu software emulator, and then progressed to booting it on the embedded hardware.

Finally, the hardcoded C entrypoint was replaced with the MirageOS console and network infrastructure by hooking in the relevant C functions to the `CONSOLE` and `NETIF` module signatures in MirageOS. The Wifi module in the ESP32 is a refreshingly clean C interface to initialise and retrieve Ethernet frames from the wireless

[2] `https://github.com/sadiqj/ocaml-esp32-docker`
[3] `https://github.com/TheLortex/ocaml-esp32`

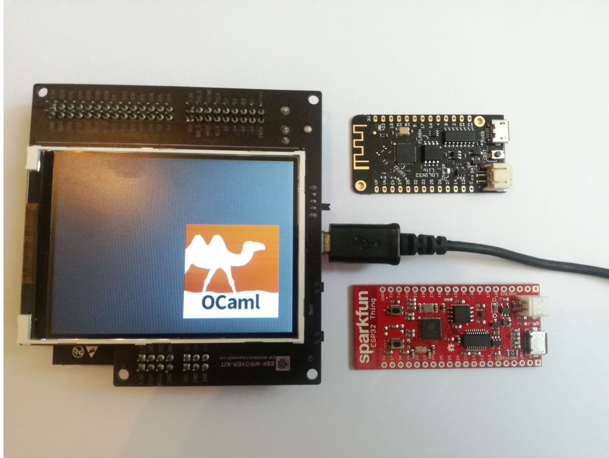

Figure 1: Three of the ESP32 hardware platforms that we have ported OCaml and MirageOS to run on. They are also capable of running on battery power for embedded operation.

interface, and we soon had network packets flowing over our local network! At this point, the rest of MirageOS was the standard OCaml networking stack (tcpip, cohttp and so on). We also have access to some ESP32 hardware that has a display interface that is mapped as a framebuffer accessible from OCaml, which we will demonstrate during our talk.

2.2 Memory Usage

The most difficult limitation to workaround in the ESP32 is the limited on-board memory. The chip has 520kB of SRAM for data and instructions, and some hardware bundles have larger 4-16MB of QSPI external flash. In practical terms, this means ensuring very good linking hygiene to ensure that unnecessary packages are not part of the OCaml dependency cone.

The first thing we noticed is that code size could be significantly improved by "sealing" the compiled executable by removing dynamic linking, and reducing the amount of module time initialisation work that happens. We did this manually for some packages, and we could find good use for the Link Time Optimisation[4] pass to be considered for inclusion into OCaml to add more principled dead-code elimination to the OCaml ecosystem.

2.3 Cross Compilation

The other activity we did to improve the memory usage situation was to submit several fixes to upstream Opam packages to remove unnecessary dependencies at the opam level.

Since (unlike other architectures) everything is cross-compiled in the ESP32, pulling in the full package dependency cone is problematic. During the course of our work, the developers of Jbuilder/Dune added cross-compilation support to that build system, which made the package management significantly more consistent than dealing with a myriad of individual opam packages.

As a result, we can now automatically synthesise opam packages that are cross-compiled to their ESP32 variants, within the limits of only working with a set of build systems. This is sufficient to (e.g) compile a large portion of the MirageOS package ecosystem for ESP32.

3 Usecases and Hardware

Our primary usecase for this work is to deploy safe, reliable MirageOS unikernels written in pure OCaml into embedded environments. The ESP32 chipset is showing up in a number of interesting hardware devices, but is currently usually programmed in C. The combination of OCaml's runtime efficiency and good support for metaprogramming makes it an attractive language to base IoT infrastructure on.

In terms of hardware, Figure 1 shows some of the boards we are working with. They range from the tiny Lolin32 chip (2-4 dollars each), to the ESP32 IDF reference board that includes more RAM and a builtin LED display. We have also ported to the Matrix Voice, which includes high quality microphones in order to build open source voice assistants – this board also includes a Spartan FPGA in addition to the ESP32 processors, which has prompted us to begin investigating the use of the HardCaml[5] hardware design library.

3.1 Upstreaming

The native code port of ESP32 is in pretty good shape now, and so we are considering the best route to upstreaming this so that it is tracked by mainline OCaml. This requires some consideration of whether upstream OCaml can accept a cross-compilation only architecture, and the extra testing burden this would impose. The native code patch itself is self-contained and does not disturb other targets such as x86 or ARM, which is a good sign.

For the package ecosystem, the main work we need to do is to integrate the build flags into Jbuilder/Dune so that package cross-compilation can happen semi-automatically. This is steadily happening via the work ongoing in the OCaml Platform.

For MirageOS, we still need to integrate the ESP32 toolchain into the `mirage` CLI tool. This depends on some of the core MirageOS packages being ported to

[4] `https://github.com/ocaml/ocaml/pull/608`

[5] `https://github.com/janestreet/hardcaml`

Dune, but after that we do not anticipate any particular difficulties fitting it in alongside the other existing unikernel targets.

3.2 Acknowledgements

We would like to thank Mark Shinwell, David Allsopp, KC Sivaramakrishnan, Stephen Dolan and Gemma Gordon for their help and advice on making this work possible, and the MirageOS team lead by Thomas Gazagnaire for timely library fixes. The Dune team lead by Jeremie Dimino and Rudi Grinberg were extremely helpful with adding cross-compilation features that were needed. This hardware for this project was supported by a mini-projects award from the Centre for Digital Built Britain, under InnovateUK grant number 90066.