Running Erlang and Elixir on microcontrollers with AtomVM

Davide Bettio <davide@uninstall.it>
https://github.com/bettio/AtomVM
#atomvm on https://elixir-lang.slack.com

Code BEAM Lite Italy 2019

About me

Uninstall on IRC/Slack/etc…

Software developer at Ispirata (Padova)

Working on Astarte → https://github.com/astarte-platform/astarte

- Astarte is an IoT platform written in Elixir

C/C++ developer for a while

- KDE developer since 2006
- Embedded software developer

AtomVM since 2017

What is an embedded system?

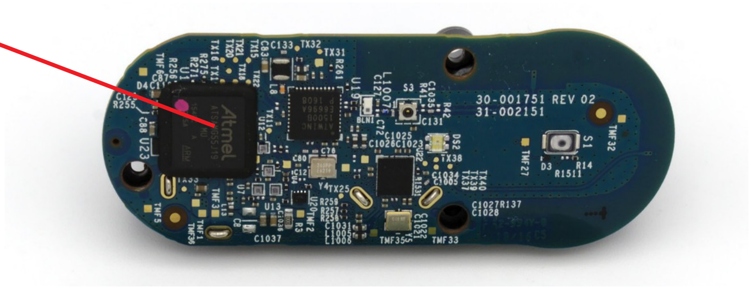
A system hidden inside a device

Compared to a PC it has constrained resources

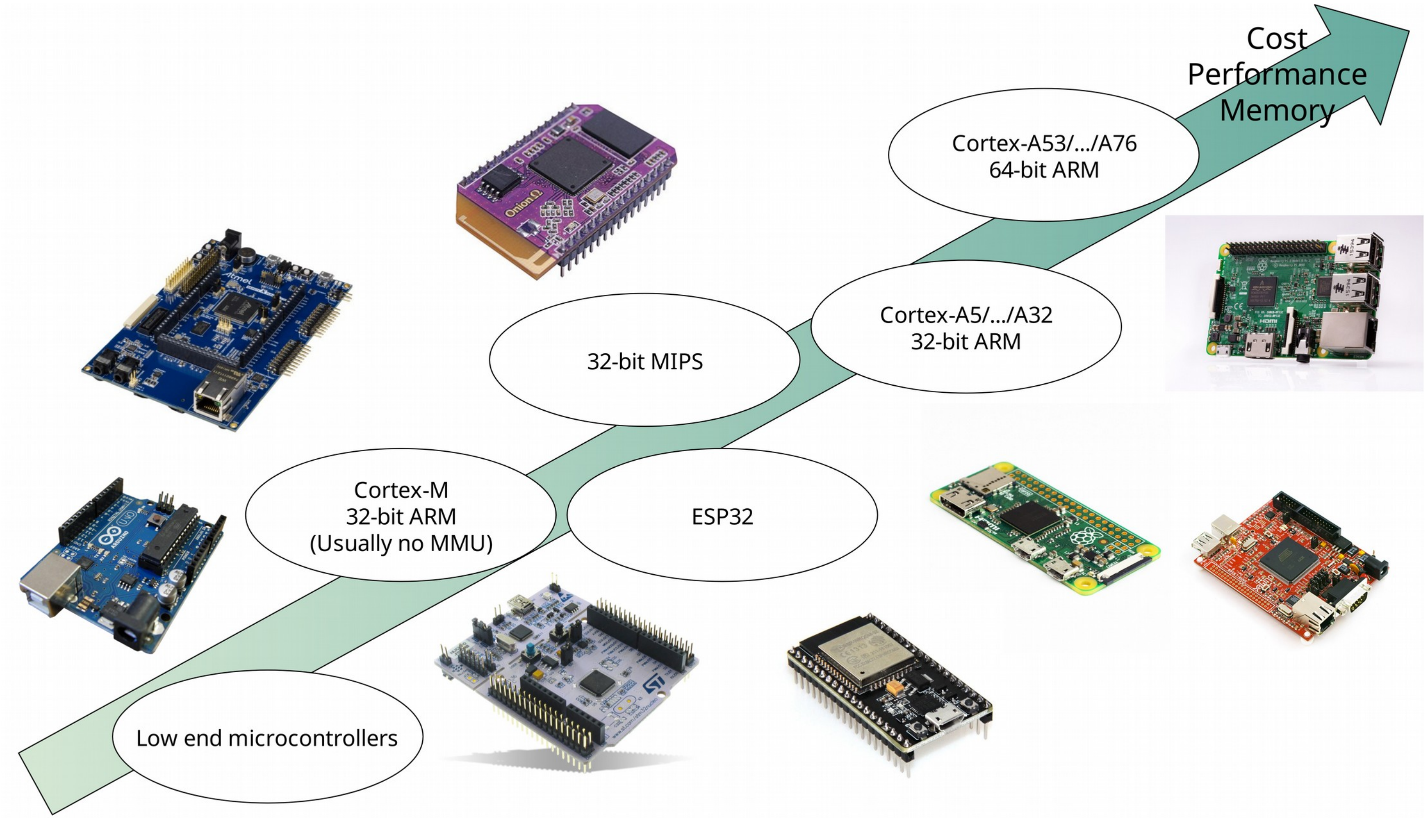
Frequently battery powered → sensors

Lot of different kinds of hardware and SDKs

Embedded system with
- CPU running at 120 MHz
- 176 KiB of RAM
- 512 KiB of FLASH
- WiFi connectivity



Embedded systems hierarchy



Embedded systems hierarchy

We can identify 2 bigger groups:

- High-end: CPUs with a MMU (and enough memory) so they can run an unmodified operating system such as Linux

- Low-end: CPUs with no-MMU or little memory, they need some custom software/OS on it such as uCLinux, FreeRTOS, Contiki, etc...

High-end systems

SoCs such as BCM2837 (RPi3), i.MX6, SAM9, Ath. AR9331 etc...

Usually > 16 MiB of RAM

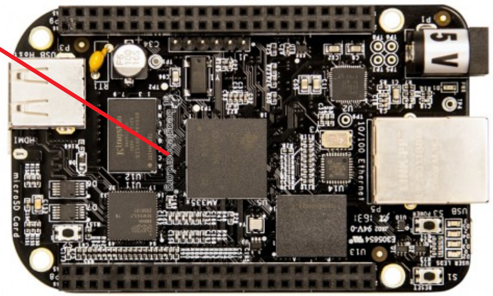
Usually > 16 MiB of FLASH

Capable of running an operating system such as Linux

Capable of running BEAM

→ https://nerves-project.org/

BeagleBone Black
Runs Linux and BEAM
512 MiB of RAM
AM335x 1 GHz ARM Cortex-A8



“Upper low-end” systems

MCUs such as ATSAMV7 (GRiSP board)...

Usually 16~64 MiB of RAM

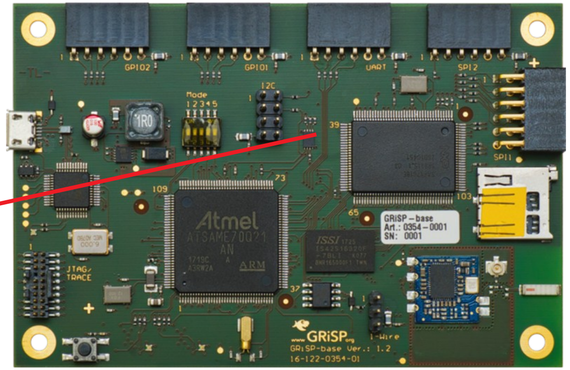
Usually 8~64 MiB of FLASH

Capable of running a RTOS (or uCLinux)

Capable of running a patched BEAM

→ https://www.grisp.org/

GRiSP board
Runs BEAM on RTEMS
ARM Cortex M7 (no MMU)
Runs at 300 MHz
64 MiB of RAM



Low-end systems

MCUs such as ESP32, STM32, etc...

Usually 128 KiB~16 MiB of RAM

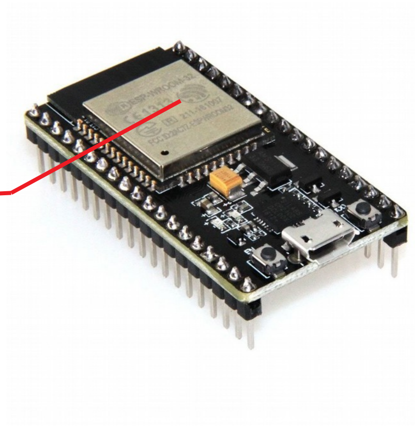
Usually 256 KiB~8 MiB of FLASH

Capable of running a RTOS (such as FreeRTOS)

BEAM does not run here

→ https://github.com/bettio/AtomVM

ESP32 board
Tensilica Xtensa LX6
Running at 240 MHz
No MMU
520 KiB of RAM



Very low-end systems

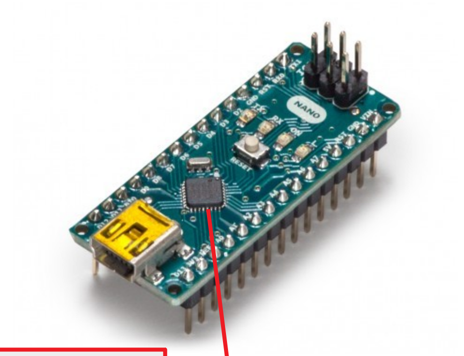
MCUs such as ATmega328p, PIC168F4

Usually 8-bit CPUs

Usually < 128 KiB of RAM

Usually < 256 KiB of FLASH

They might run a simple scheduler



Arduino Nano
ATmega328p
Runs at 16 MHz
2 KiB of RAM

No reasonable way to run Erlang, most of them can be programmed in C, some other only in assembly

Nerves

“Craft and deploy bulletproof embedded software in Elixir”

Regular Erlang/OTP based solution

Runs on top of Linux kernel

Several supported boards

mix tooling: `mix nerves.new`, `mix firmware.burn`

GRiSP

GRiSP is a board + a custom software

GRiSP-Base board has an ARM Cortex M7 with 64 MiB of RAM

GRiSP software is Erlang/OTP + custom patches + RTEMS

GRiSP 2 is under way

- The new hardware will have an ARM Cortex-A i.MX6
- The new board is quite similar to other boards that are used with Linux

AtomVM

- **Tiny Erlang VM written in C from scratch**
- **It runs on microcontrollers with less than 500 KiB of RAM**
  - Erlang and Elixir on 3 $ hardware
- **Easily portable to new hardware**
- **Easy to understand**
- **Runs .beam files**

BEWARE

Your .beam files will not work out of the box on AtomVM

Your code must be changed to work on a constrained environment

Some features will never be implemented

Why?

Makers

Makers are experimenting with alternatives to C/C++

- MicroPython/CircuitPython
- JerryScript/mJS
- eLua

They need rapid prototyping

IoT

Interaction with remote services

Need for simple error handling

Need to parse payloads

Binary protocols handling

Connected to a remote broker (usually MQTT)

New challenges: mesh networks, LoRA, etc…

Abstraction

Implementing IoT devices in C is painful

Writing code for an IoT device in plain C is a painful experience

Networking is even worse

Asynchronous operations are quite common but hard

Tasks are frequently needed

Turns out to be hard to test and debug

Takes a lot of time

Erlang and Elixir to the rescue

It is not C language

Processes

Easier to implement asynchronous processing

Easier to test and debug

Hardware independent

DSLs in Elixir

Lot of funs

Blinking a LED with Arduino

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

Blinking a LED with Erlang on AtomVM

```
-module(blink).
-export([start/0]).

start() ->
    GPIO = gpio:open(),
    gpio:set_direction(GPIO, 2, output),
    loop(GPIO, 0).

loop(GPIO, level) ->
    gpio:set_level(GPIO, 2, level),
    timer:sleep(1000),
    loop(GPIO, 1 - level).
```

Blinking a LED with Elixir on AtomVM

```
defmodule Blinker do
  def start(gpio, interval_ms) do
    gpio_driver = GPIO.open();
    GPIO.set_direction(gpio_driver, gpio, :output)

    loop(gpio_driver, gpio, interval_ms, 0)
  end

  def loop(gpio_driver, gpio, interval_ms, level) do
    GPIO.set_level(gpio_driver, gpio, level)

    :timer.sleep(interval_ms)

    loop(gpio_driver, gpio, interval_ms, 1 - level)
  end
end
```

Hello Arduino, can you do this?

```
defmodule Blink do
  def start do
    spawn(Blinker, :start, [{:d, 12}, 1000])
    spawn(Blinker, :start, [{:d, 13}, 500])
    spawn(Blinker, :start, [{:d, 14}, 1500])
    spawn(Blinker, :start, [{:d, 15}, 300])

    loop()
  end

  def loop do
    loop()
  end
end
```

Bringing up WiFi

```
-module(setup_network).
-export([start/0]).

start() ->
    NetworkConfig = [{sta, [
        {ssid, "mynetwokid"},
        {psk, "mypassword"}
    ]}],
    network:setup(NetworkConfig).
```

Flashing to the real hardware

Code must be compiled using erlc/elixirc

Microcontrollers have no filesystem on their flash

- .beam files must be packed together to an .avm file

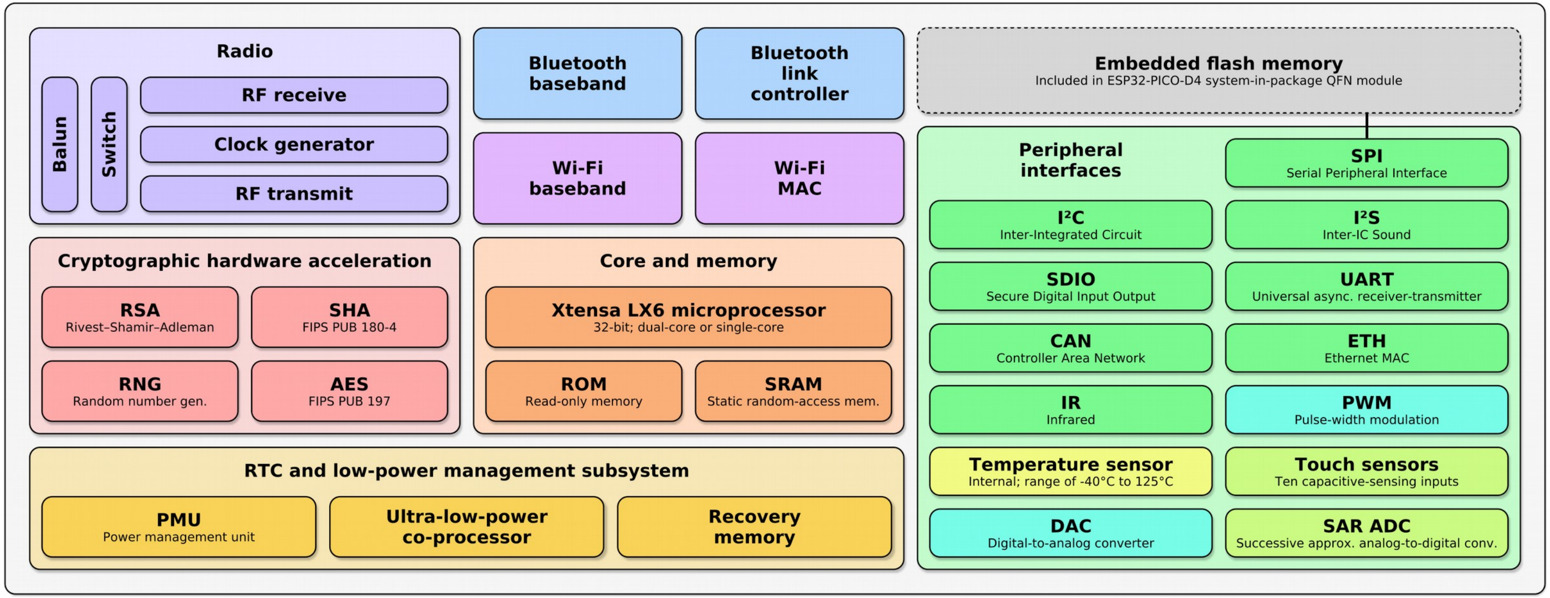
esp32 → `$IDF_PATH/components/esptool_py/esptool/esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 115200 --before default_reset --after hard_reset write_flash -u --flash_mode dio --flash_freq 40m --flash_size detect  0x110000 hello_world.avm`

stm32 → `st-flash --reset write packed.avm 0x8080000`

Supported hardware (ESP32)

1 or 2 cores, 520 KiB of RAM, WiFi, BLE, Ethernet, etc...

Espressif ESP32 Wi-Fi & Bluetooth Microcontroller — Function Block Diagram



Supported hardware (STM32)

ARM based hardware

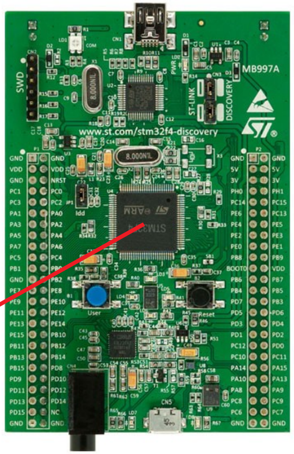
Wide choice of different models

Lot of dev boards with different peripherals

Well documented

Low power consumption

STM32 F4 Discovery
192 KiB of RAM
15-20 €



STM32 F7 Discovery
340 KiB + 128 MiB of RAM
~50 €

Supported hardware [Your favourite MCU/board here]

Just add needed code to `src/platforms`

STM32 port is < 500 lines of code

A port must provide code for:

- Loading/memory mapping modules from flash
- Waiting events and sleeping

Hardware specific features such as GPIOs are implemented as port drivers

How does it work?

- **A startup module is memory mapped → `src/main.c`**
- **.beam files are IFF files having some sections**
  - AT8U, CODE, EXPT, LOCT, IMPT, etc...
- **Code is parsed**
- **A label offsets table is built**
- **A startup function is searched in exported functions table**

How does it work?

Code is executed in place → No JIT, no threaded code

Execution in place does not require additional memory

Just one huge switch that keeps executing BEAM code

```
{label,4}.
  {allocate,1,2}.
  {move,{x,1},{y,0}}.
  {call,1,{f,6}}.
  {move,{x,0},{x,1}}.
  {move,{y,0},{x,0}}.
  {move,{x,1},{y,0}}.
  {call,1,{f,6}}.
  {gc_bif,'+',{f,0},1,[{y,0},{x,0}],{x,0}}.
  {deallocate,1}.
  return.
```

```
while (1) {
    switch (code[i]) {
        case OP_MOVE: {
            int next_off = 1; term src_value;
            DECODE_COMPACT_TERM(src_value, code, i,
                                next_off, next_off);
            int dreg; uint8_t dreg_type;
            DECODE_DEST_REGISTER(dreg, dreg_type, code, i,
                                 next_off, next_off);
            WRITE_REGISTER(dreg_type, dreg, src_value);

            NEXT_INSTRUCTION(next_off);
            break;
        }
```

How does it work?

Each process has:

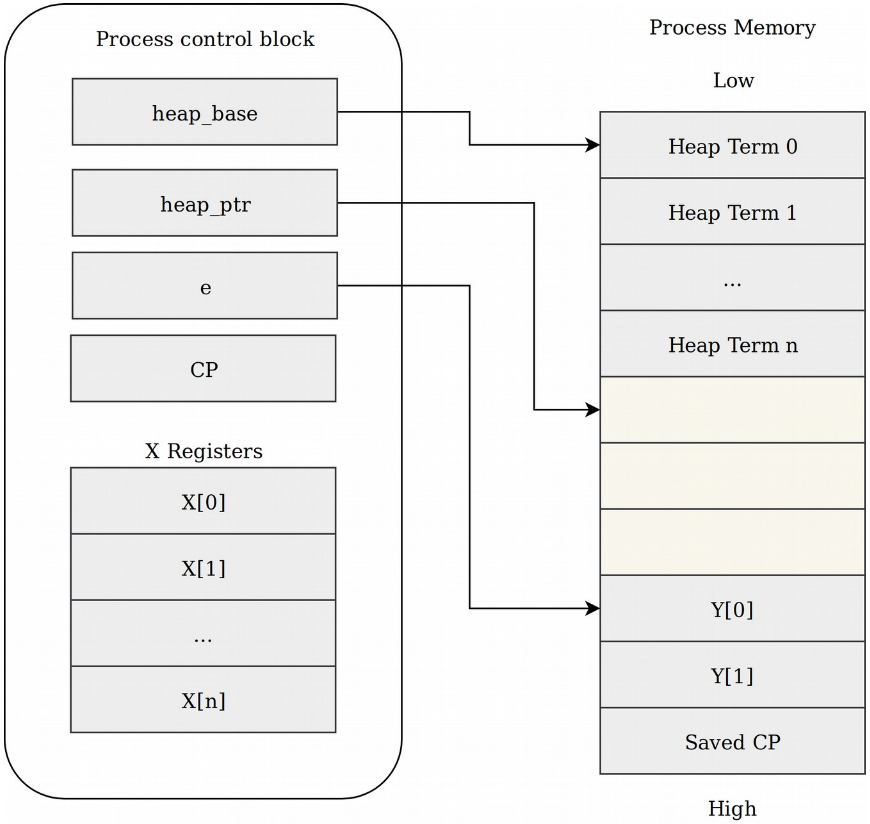
- A set of X registers
- A stack and a set of Y registers pointing to stack slots
- A heap

Function arguments are stored on X registers

BEAM assembly is not CPU assembly

- e.g. no add, sub, mul → BIFs are used instead

How does it work?



How does it work?

Simple copying garbage collector (Cheney's algorithm)

Garbage collection is triggered by some instructions

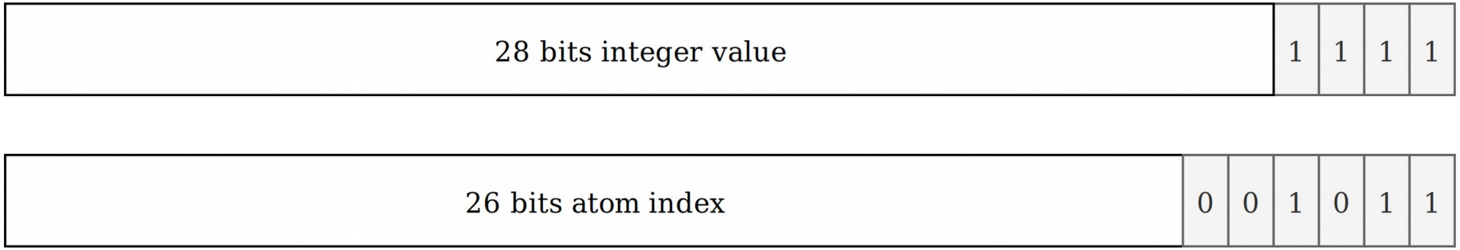
- allocate, allocate_heap, allocate_zero, allocate_heap_zero, test_heap, etc...

Same memory layout as the one used on BEAM

How does it work?

All values are tagged

On a 32-bit CPU values bigger than 134217728 are stored on the heap



Useful resources

https://happi.github.io/theBeamBook/

Some differences

Optional big integers support (at compile time) → overflow error

Optional floating point support (at compile time)

Easier to run out of memory → out_of_memory error

Some features are missing

Future developments

Better tooling, e.g. mix task

Bootloader

Remote shell

More documentation

Future developments

Complete support for binaries

Maps

Supervision trees

Floating point support

Big integer support

An improved standard library

Support for multiple cores

<Your contribution here>

Future developments

Ready to use port drivers for hardware integration

Modules for sensors support

Crazy ideas

WebAssembly port

Distributed Erlang

Secondary cores as port drivers

Conclusions

Running Elixir on a RaspberryPi (or similar hardware) → Nerves

Running Erlang/Elixir on a constrained system → AtomVM

Not all hardware is suitable

Your code needs to be “ported” to run on AtomVM

Thanks

https://github.com/bettio/AtomVM