
AtomVM

About me (Davide Bettio)

https://github.com/bettio/ | davide@uninstall.it | https://uninstall.it/

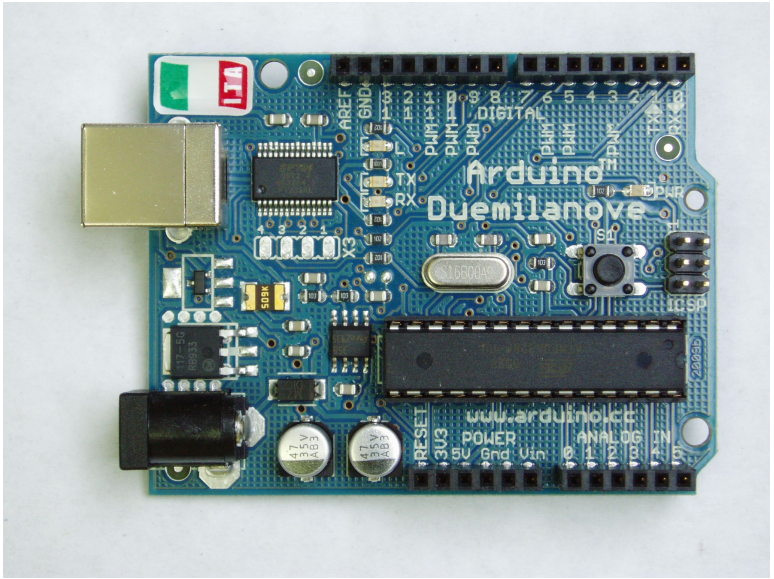
- Tinker with hardware and embedded systems since 2004.
- Long-time open-source dev (since ~2005 contributed to KDE Plasma and others).
- Fell in love with Elixir in 2017, while creating Astarte Platform.
- Started the AtomVM project in 2017
- I love hiking!

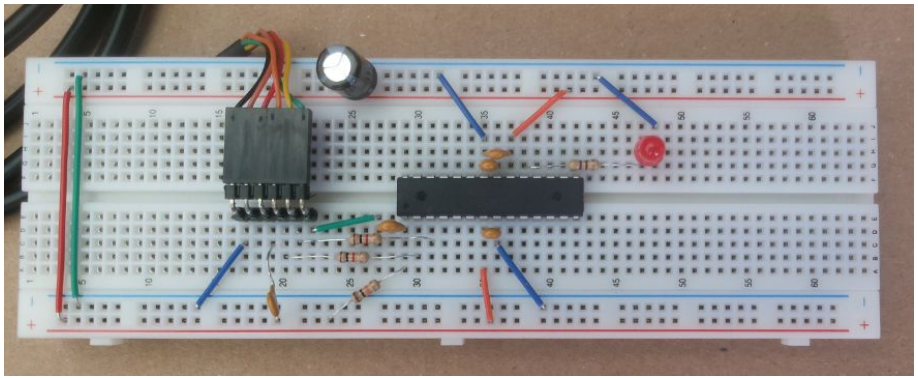
AtomVM

AtomVM

Once Upon a Time, the Arduino

- The pioneer of physical computing devices
- Simple to assemble and develop
- Cheap (arduino ~20 €, IC: < 2 €)
- Very limited, yet so powerful

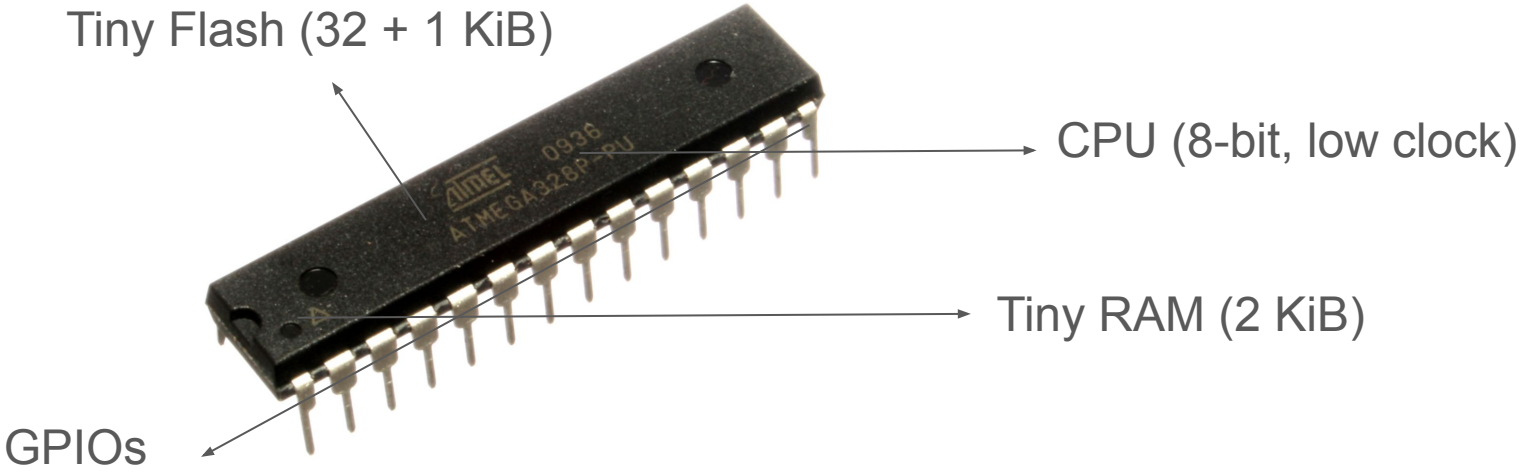




AtomVM

Classic MCU Anatomy (e.g., ATMega328P)

A small Computer on a Chip:

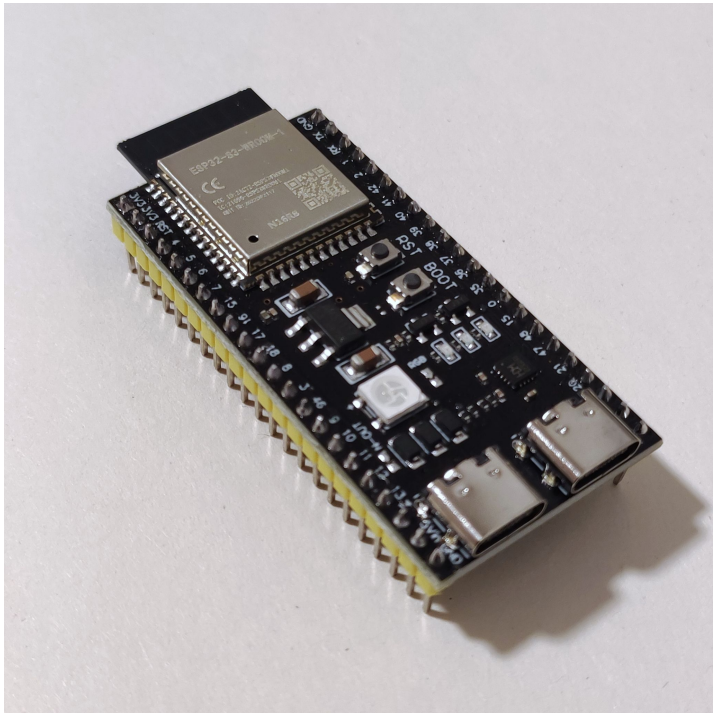


AtomVM

Modern MCU: ESP32 Example

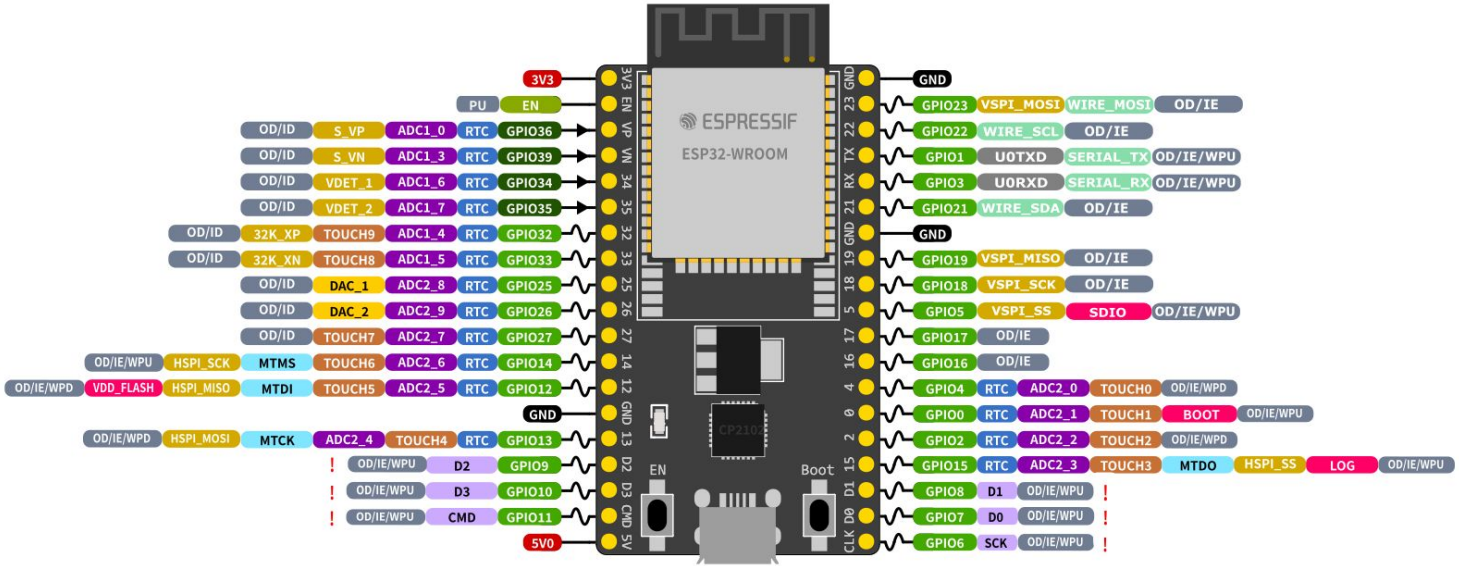
ESP32:

- Cost < 5 €
- Dual Core @ 240MHz
- RAM: ~500KB - 8MB
- Flash: 4MB - 16MB
- Connectivity: WiFi, Bluetooth, etc.
- Lot of GPIOs & integrated peripherals
- Low Power / Battery-friendly



AtomVM

ESP32-DevKitC



ESP32 Specs

32-bit Xtensa® dual-core @240 MHz
Wi-Fi IEEE 802.11b/g/n 2.4 GHz
Bluetooth 4.2 BR/EDR and BLE
520 KB SRAM (16 KB for cache)
448 KB ROM
34 GPIOs, 4x SPI, 3x UART, 2x I2C
2x I2S, RMT, LED PWM, 1 host SD/eMMC/SDIO
1 slave SDIO/SPI, TWAI®, 12-bit ADC, Ethernet

PWM Capable Pin
GPIOX — GPIO Input Only
GPIOX — GPIO Input and Output
DAC_X — Digital-to-Analog Converter
DEBUG — JTAG for Debugging
FLASH — External Flash Memory (SPI)
ADCX_CH — Analog-to-Digital Converter
TOUCHX — Touch Sensor Input Channel
OTHER — Other Related Functions
SERIAL — Serial for Debug/Programming
ARDUINO — Arduino Related Functions
STRAP — Strapping Pin Functions

RTC — RTC Power Domain (VDD3P3_RTC)
GND — Ground
PWD — Power Rails (3V3 and 5V)
! — Pin Shared with the Flash Memory Can't be used as regular GPIO

GPIO STATE

WPU: Weak Pull-up (Internal)
WPD: Weak Pull-down (Internal)
PU: Pull-up (External)
IE: Input Enable (After Reset)
ID: Input Disabled (After Reset)
OE: Output Enable (After Reset)
OD: Output Disabled (After Reset)

Modern MCU: RP2040 (Pi Pico) Example

Raspberry Pi Pico (RP2040):

- Cost < 5 €
- Dual Core @ 133MHz+
- RAM: 264KiB+
- Flash: 2MB+ (via QSPI)
- GPIOs, Periph. & Programmable I/O (PIO)
- WiFi option
- Low power

AtomVM

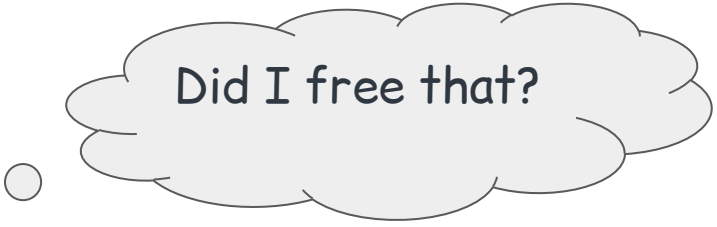
Powerful, But Still...Different

- Massive leap from classic MCUs
- Still resource-constrained vs. PC/Servers
  - KB/MB RAM, not GB
  - No OS (or RTOS)
  - Development: Mostly C / C++

AtomVM

The C/C++ Experience on MCUs

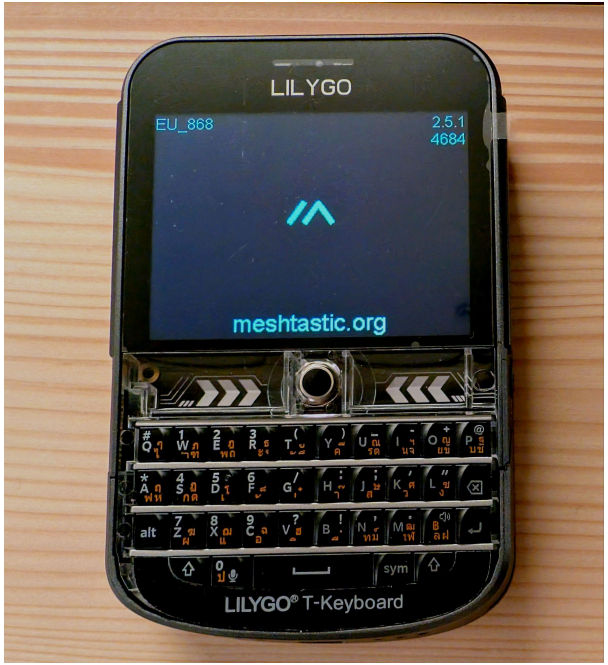
- Concurrency? Manual, tricky.
- Binary parsing? Boring & dangerous.
- Async? Callback hell, anyone?
- Memory?



AtomVM

The Intricacies of Embedded Communication: LoRa

- LoRa: Long-Range radio, raw bytes to CPU
- Need to implement: routing, security, mesh
- Meshtastic parses them in C++
    - C++: One wrong move... 💣



AtomVM

Clarity in Complexity for LoRa Packets

```
def parse(
    <<dest::little-unsigned-32, src::little-unsigned-32, pkt_id::little-unsigned-32,
      hop_start::size(3), via_mqtt::size(1), want_ack::size(1),
      hop_limit::size(3), channel_hash::8, _padding::16, encrypted_data::binary>>) do
  {:ok, %{dest: dest, src: src, packet_id: pkt_id,
          hop_start: hop_start, via_mqtt: int_to_bool(via_mqtt), want_ack: int_to_bool(want_ack),
          hop_limit: hop_limit, channel_hash: channel_hash, encrypted_data: encrypted_data}}
end

def parse(_), do: {:error, :failed_meshtastic_parse}

def decrypt(%{src: src, packet_id: pkt_id, encrypted_data: enc_data} = packet, key) do
    iv = <<pkt_id::little-unsigned-64, src::little-unsigned-32, 0::32>>

    decrypted = :crypto.crypto_one_time(:aes_128_ctr, key, iv, enc_data, false)
    packet
    |> Map.put(:data, decrypted)
    |> Map.delete(:encrypted_data)
end

defp int_to_bool(0), do: false
defp int_to_bool(1), do: true
```

AtomVM

Projects like Meshtastic couldn't leverage these advantages on such microcontrollers. **The standard BEAM VM wasn't designed for environments with only ~500 KiB of available RAM**.

What if we could bring *somehow* the safety, concurrency, and productivity of the BEAM ecosystem to these tiny devices?

AtomVM

To the Rescue

AtomVM, A lightweight virtual machine designed to run compiled Erlang and Elixir code on microcontrollers with limited resources.

- Key Trade-offs:
  - Memory First: RAM & Flash are precious
  - Portability: New targets in hours, not days
  - Flexible Requirements: Adaptable core

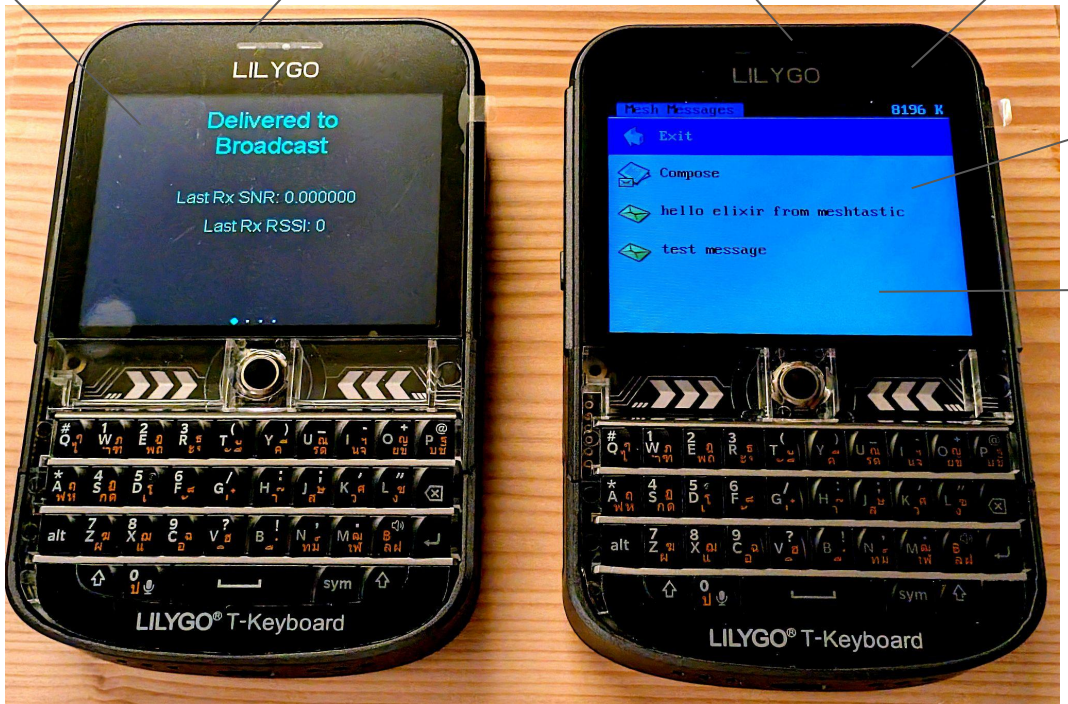
AtomVM

To the Rescue

Original firmware in C++

Battery powered
CPU: ESP32-S3
Radio: LoRa

Runs AtomVM



Display managed using https://github.com/atomvm/**AtomGL** component.
UI is made with Elixir

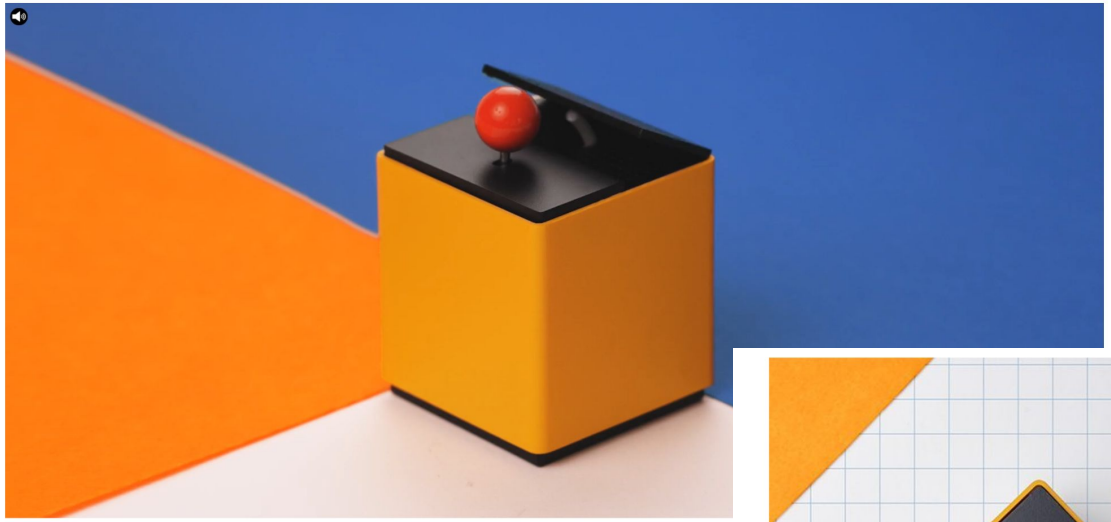
Runs an Elixir app, no native parts are used (github.com/bettio/pocketOS/)

AtomVM

la machine

by multiplié

The Useless Box : Reloaded



- AtomVM powered
- ESP32-C3
- 32-bit RISC V single core @ 160 MHz 400 KB of SRAM
- 5µA in deep sleep !

- la machine code is in Erlang
- uses atomvm_esp_adf component for playing audio from Erlang code (thanks Paul)

KEY FEATURES

- Over 500 unique sound effects & reactions
- Unlimited choreography combinations - never the same twice
- Fully modular design for easy repairs & customization
- Powered by ESP32 architecture
- Completely open source software - hack it, modify it, make it yours
- Eco-friendly construction from 100% recycled materials
- Exceptional battery life: three months on a single charge

Popcorn

Popcorn is a library that allows you to run client-side Elixir in browsers, with JavaScript interoperability.

https://popcorn.swmansion.com/

AtomVM

Popcorn: How Does it Work?

- Applications on AtomVM, compiled to WebAssembly (emscripten platform)
- Small footprint: the VM is ~200 KiB gzipped
- Popcorn gives you:
  - Tooling: `mix popcorn …`
  - An easy-to-use library for JavaScript interoperability
  - The *full* Elixir standard library, not the reduced version used on MCUs

TL;DR: It's still AtomVM, just with batteries included for the browser

AtomVM

Wise Manul:

Simple as:

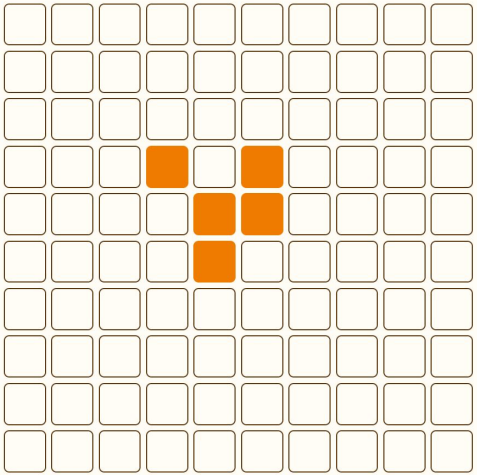
```
def deps do
  [
    {:popcorn, "~> 0.1.0"}
  ]
end
```

AtomVM

Game of life demo

A cellular automaton simulation written in Elixir. The entire UI is controlled by Elixir with no additional JavaScript. Every cell is distinct Elixir process. Click cells to toggle them, use the glider preset, or start the simulation.

Start simulation | Reset | Use glider preset



AtomVM

This live demo showcases Elixir's IEx – running right here in the browser. Write your own code and make it happen, or click on the buttons to run examples that we've prepared for you.

See the repo

Read the docs

```
Interactive Elixir (1.17.3) - press Ctrl+C to exit (type h() E
NTER for help)
avm_iex(1)> :ok

:ok
avm_iex(2)> hi = fn -> IO.inspect(self()) end

#Function<0.23/1 in :erl_eval.avmo_expr/6>
avm_iex(3)> Enum.each(0..3, fn _ -> spawn(hi) end)

#PID<0.39.0>
#PID<0.40.0>
#PID<0.41.0>
#PID<0.42.0>
:ok
avm_iex(4)>
```

Example: Sort

Example: Processes

AtomVM

Moving to Real Hardware

What you Need / Compatible Hardware

Option 1: **Espressif**

- **ESP32 / ESP32-S2, ESP32-S3 DevKit C** → **Wifi, Bluetooth**, up to **8 MiB** of RAM
- **ESP32-C2/C3** → Wifi, Bluetooth, up to ~512 KiB or RAM, RISC-V CPU
- **ESP32-C6** → Wifi, Bluetooth, **Thread, ZigBee**, RISC-V CPU
- ESP32-H2, ESP32-C5, ESP32-P4 misc models with different features

Disclaimer: Do not buy ESP8266 and other ancient devices pre-ESP32

What you Need / Compatible Hardware

Option 2: **RaspberryPi**

- Pico 1/1W (RP2040) → **264 KiB of RAM** (optional Wifi and Bluetooth: W model)
- Pico 2/W (RP2350) → **512 KiB of RAM** (optional Wifi and Bluetooth: W model)

AtomVM

What you Need / Compatible Hardware

Option 3: **STM32**

- Lot of boards, I'm not going to mention them

Disclaimer: Make sure to use a model with enough flash and RAM

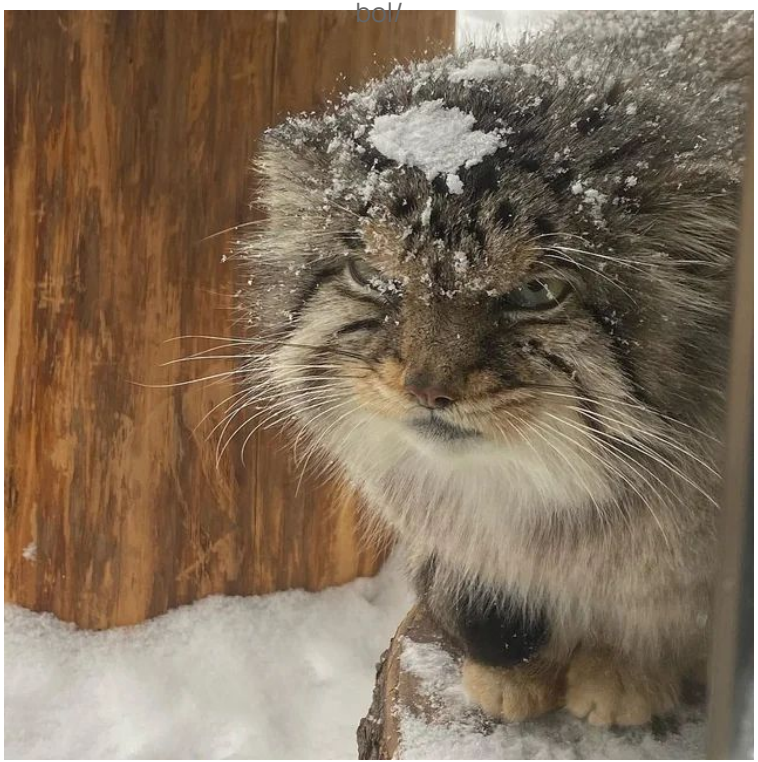
Disclaimer 2: I will not further talk about this target, it is not yet "golden"

AtomVM

Incompatible Hardware

- Classic Arduinos like the Uno aren't supported
  - But there are some great Arduino boards based on the ESP32 that work perfectly!
- As a rule of thumb, you'll want at least 128 KiB of RAM for most projects
- Support for Nordic nRF chips is on our wish list, but not there yet

Source: https://www.reddit.com/r/PallasCats/comments/1d8j3jd/hol/



AtomVM

What you Need / Accessories

- Minimal hardware setup: just a USB cable (that's it)
- A serial terminal app (like `minicom` on Linux/macOS or PuTTY on Windows)
  - This is how you'll see all the debug, error, and info messages from your device
- A working Elixir install

AtomVM

Big Disclaimer

Source: https://manulization.com/manuls/magellan.html

- Heads up: AtomVM is still pre-v1.0, which means APIs are not yet stable
- We will break APIs, but the core concepts will remain the same
- The code here might not work forever, but we keep the official documentation and examples up-to-date

See also: https://doc.atomvm.org/latest/UPDATING.html

AtomVM

Next Step: exatomvm

- Add `{:exatomvm, github: "AtomVM/exatomvm", runtime: false}` to `mix.exs`
- It provides you a number of mix tasks to build your AtomVM project and flash it

See also: https://github.com/atomvm/exatomvm

AtomVM

Do I Need a C Toolchain or SDK?

- Raspberry Pico: never been an issue: just flash the uf2 file as you have been used to
  - exatomvm handles the uf2 creation

- ESP32:
  - For ESP32, not anymore! The `exatomvm` installer handles it
  - If you add `{:pythonx, "~> 0.4.0", runtime: false}`, to your deps, you can just flash your app without any additional burden
  - No need to download AtomVM, just do **`mix atomvm.esp32.install`**

AtomVM

Configuring mix.exs

Just add an atomvm section to mix.exs project function:

```
def project do
    [...]
    atomvm: [
      start: Blink, # the module with our start/0 entry point function
      flash_offset: 0x210000
    ]
end
```

AtomVM

The Physical Computing Hello World

```
defmodule Blink do
  @pin 2
  def start() do
      GPIO.set_pin_mode(@pin, :output)
      loop(:high)
  end
  defp loop(level) do
      GPIO.digital_write(@pin, level)
      Process.sleep(200)
      loop(toggle(level))
  end
  defp toggle(:high), do: :low
  defp toggle(:low), do: :high
end
```

See also:
https://github.com/atomvm/AtomVM/tree/main/examples

AtomVM

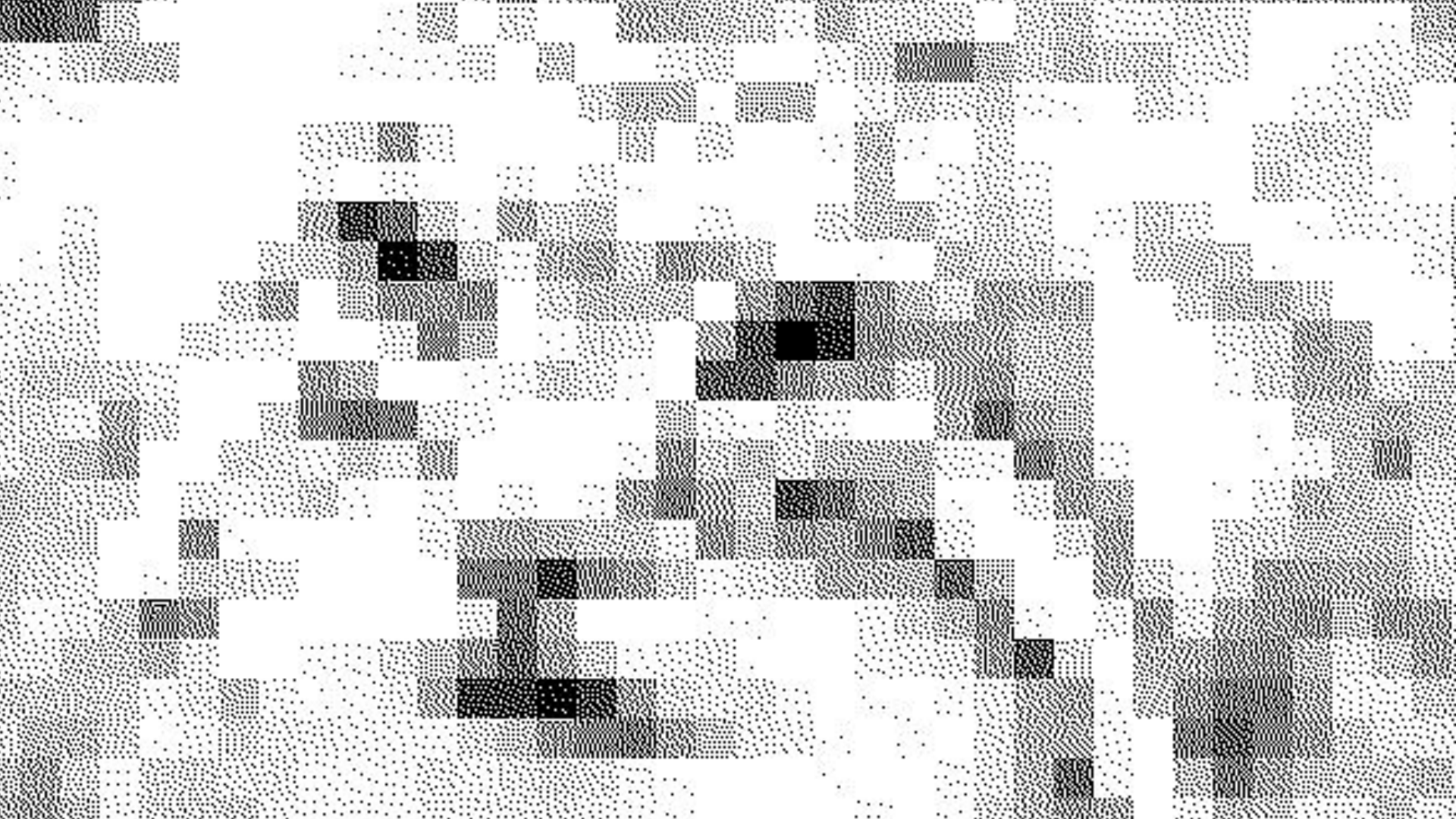
What's a GPIO?

- **GPIO** stands for **G**eneral **P**urpose **I**nput/**O**utput.
- Think of them as simple digital pins that can be either an input or an output
- They can be set to high (e.g., `3.3V`) or low (`0V` / Ground).

For our LED, this means it's either fully on or fully off. No fading.

AtomVM

AtomVM



The AtomVM Workflow

- Add `{:exatomvm, github: "AtomVM/exatomvm", runtime: false}` to `mix.exs` ✓
- Write Elixir/Erlang (like always!) ✓
- (Behind the scenes: compile, like always!)
- (Behind the scenes: pack, `mix.atomvm.packbeam → myapp.avm`)
- **Flash, run one command: (e.g. `mix atomvm.esp32.flash`)**

Remember: AtomVM runs unmodified BEAM file, so any language that runs on the BEAM, will run also on AtomVM.

AtomVM

Demo

HONESTLY I WILL NOT DO A BLINKING LED DEMO. TRUST ME IT WORKS. I WILL NOT EVEN TRY SHOWING A MICROSCOPIC LED TO THE AUDIENCE. IF I BRING BOARDS AND ELECTRONICS ON PUBLIC TRANSPORTS I MIGHT BE MISIDENTIFIED AS A TERRORIST. DEMOS ALWAYS FAIL

AtomVM

Time for `minicom -D /dev/ttyACM0`

- As soon as the device is flashed use minicom for reading `IO.puts` and `IO.inspect` output
- Do not try using `minicom` while flashing the device
- It may require some configuration

AtomVM

Circuits Pro-tips

Do not

1. **Never mess with "GND"** (the ground): if you connect GND pin to something that is not ground/0V you are likely going to fry your device
2. **Respect polarity:** components like LEDs and some capacitors have positive and negative sides: connecting them backward = ☠️
3. **Don't mix voltage levels**: sending 5V into a 3.3V pin can permanently damage the chip unless the pin is explicitly *'5V tolerant'*
4. **Always connect an antenna**: before powering on a radio. Without it, the transmitter can be damaged

Do: **Double check all your connections before powering up your device!**

AtomVM

Handling a Button Press

Goal: We want to know when a button is pressed

- The naive way is "polling": constantly looping to check the button's GPIO pin
  - The problem? This keeps the CPU busy doing nothing and drains the battery. This is called "busy-waiting"

- A better way: Interrupts
  - A hardware interrupt tells the CPU to pause its current task and handle something important *right now*
  - In AtomVM, we translate these hardware interrupts into standard Elixir messages

AtomVM

Interrupts in Elixir

First, we configure the GPIO as an input and tell it to trigger an interrupt on a 'falling edge' (when the button is pressed):

```
:gpio.set_direction(gpio, gpio_num, :input)
:gpio.set_int(gpio, gpio_num, :falling)
```

The hardware event is safely delivered to your process's mailbox as a message. No callbacks, no polling—just the actor model you already know and love. e.g. let's add to our GenServer:

```
def handle_info({:gpio_interrupt, gpio_num}, state) do
    IO.puts "Button pressed"
    {:noreply, State}
end
```

AtomVM

Source: https://www.flickr.com/photos/tambako/31556104335/in/photostream/

Great, I'm blinking an LED and reading a button. Now what?

AtomVM

Peripherals!

- Connecting to the outside world: Peripherals!
  - I2C
  - SPI
  - UART
- Most sensors, displays, and other modules you can buy use one of these standard communication protocols (or "buses")
- Usually, a *bus* lets you connect multiple devices to the same set of GPIO pins

AtomVM

A Quick Guide to Peripheral Buses

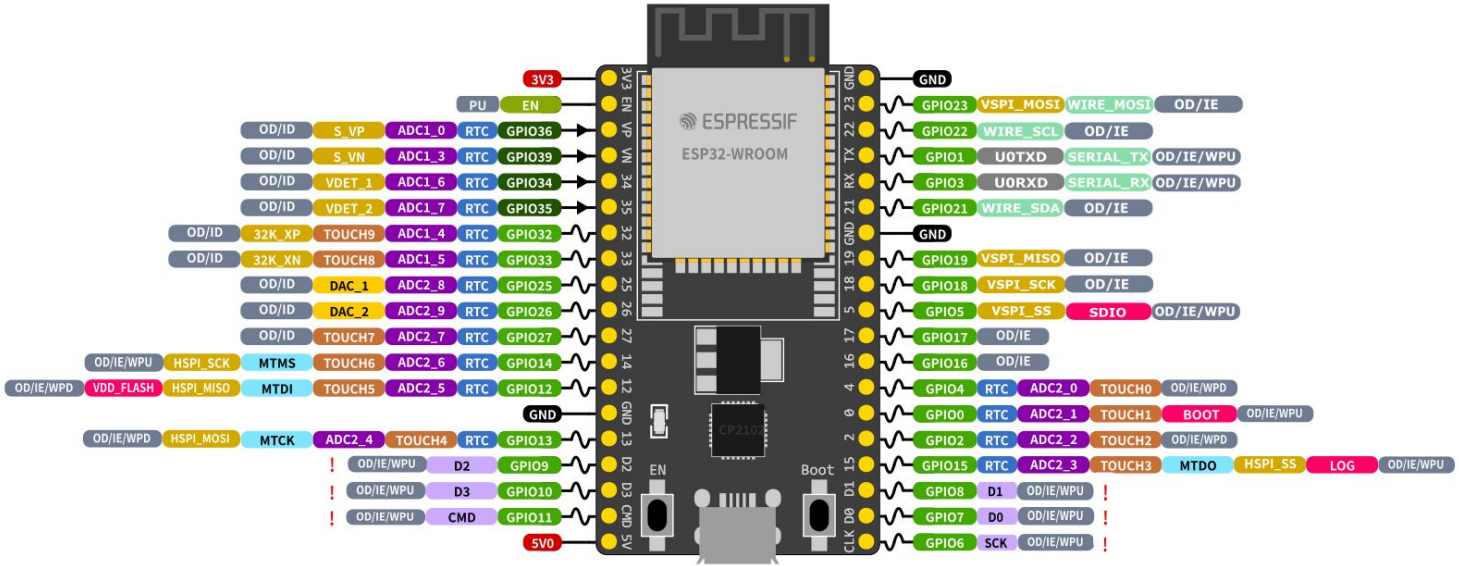
- **I²C**: A 2-wire bus. Great for connecting many different devices (each has a unique address) at slow-to-medium speeds. It's synchronous.
    - Pay attention to pull-up resistors when buying breakout boards (is included or not?)

- **SPI**: A 4-wire bus (or more). Faster than I²C, great for things like displays and SD cards. It's also synchronous.
    - Requires an additional select wire for each peripheral

- **UART**: A simple 2-wire, point-to-point connection. Think of it as a simple serial port. It's asynchronous.

AtomVM

ESP32-DevKitC



ESP32 Specs

32-bit Xtensa® dual-core @240 MHz
Wi-Fi IEEE 802.11b/g/n 2.4 GHz
Bluetooth 4.2 BR/EDR and BLE
520 KB SRAM (16 KB for cache)
448 KB ROM
34 GPIOs, 4x SPI, 3x UART, 2x I2C
2x I2S, RMT, LED PWM, 1 host SD/eMMC/SDIO
1 slave SDIO/SPI, TWAI®, 12-bit ADC, Ethernet

PWM Capable Pin
GPIOX — GPIO Input Only
GPIOX — GPIO Input and Output
DAC_X — Digital-to-Analog Converter
DEBUG — JTAG for Debugging
FLASH — External Flash Memory (SPI)
ADCX_CH — Analog-to-Digital Converter
TOUCHX — Touch Sensor Input Channel
OTHER — Other Related Functions
SERIAL — Serial for Debug/Programming
ARDUINO — Arduino Related Functions
STRAP — Strapping Pin Functions

RTC — RTC Power Domain (VDD3P3_RTC)
GND — Ground
PWD — Power Rails (3V3 and 5V)
! — Pin Shared with the Flash Memory Can't be used as regular GPIO

GPIO STATE

WPU: Weak Pull-up (Internal)
WPD: Weak Pull-down (Internal)
PU: Pull-up (External)
IE: Input Enable (After Reset)
ID: Input Disabled (After Reset)
OE: Output Enable (After Reset)
OD: Output Disabled (After Reset)

Native vs Elixir Components

- AtomVM APIs such as spi, i2c, gpio and other related modules, allow building libraries for quite complex peripherals
- For high-performance or very low-level hardware access, you might need more speed than pure Elixir can provide
- For this, AtomVM supports native components (NIFs and Ports), just like the main BEAM VM
- The catch: using native components requires a custom AtomVM build using the device's SDK (like the ESP-IDF)

AtomVM

AtomGL

- AtomGL is the component for displaying stuff on screen
- Displays any list of items:

```
{:text, 16, 16, :default16px, 0xFFFFFF, 0x404040, title},
{:image, div(320 - 64, 2), div(240 - 64, 2), 0x404040, error_image}
```

- There is an additional `avm_scene` library that provides some scaffolding for managing displayed scene, using a `gen_server` like approach:

```
def handle_info(:show_foo, %{width: width, height: height} = state) do
    {:noreply, state, [{:push, items}]}
end
```

See also: https://github.com/atomvm/atomgl

AtomVM

Audio? atomvm_esp_adf

- Provides building blocks for building simple audio pipelines, e.g.
- Leverage existing esp-idf library for decoding audio formats such as mp3

See also https://github.com/pguyot/atomvm_esp_adf/

AtomVM

What about I in IoT?

- There is a handy `network` module
- AtomVM has support for `gen_tcp`, `gen_udp` and `socket`
- `http_server` and `ahttp_client` modules
- `mdns` for finding your device on the network

See also: https://doc.atomvm.org/latest/network-programming-guide.html

AtomVM

Setting up Wi-Fi

```
self_pid = self()
config = [
  sta: [
    ssid: :esp.nvs_get_binary(:atomvm, :sta_ssid, "myssid"),
    psk: :esp.nvs_get_binary(:atomvm, :sta_psk, "mypsk"),
    connected: fn -> send(self_pid, :connected) end,
    got_ip: fn ip_info -> send(self_pid, {:ok, ip_info}) end,
    disconnected: fn -> send(self_pid, :disconnected) end
  ]
]
:network.start(config)
```

AtomVM

Handling HTTP Requests

The built-in http_server module makes it easy to spin up a web server on your device:

```
router = [
  {"*", __MODULE__, []}
]
:http_server.start_server(8080, router)
[...]
def handle_req("GET", [], conn) do
  body =
    "<html>\n" <> [...]
  :http_server.reply(200, body, conn)
end
```

AtomVM

Notes / Differences

Quick Stats & Nerves Comparison

AtomVM:

- AtomVM, hello world footprint: 512 KiB of flash, 32 KiB of RAM
- Targets smaller MCUs (no Linux / no OS at all)

Nerves:

- Awesome on capable devices (RPi, etc.), such as those running Linux

AtomVM

Big Caveat

- Some features, standard modules or functions are missing (e.g. digraph module)
- But `exatomvm` will do its best to tell you if you are using any missing feature, so you can quickly iterate before flashing your application

AtomVM

`.avm` files

- They are designed to be written directly to flash memory, no filesystem needed
- Instead of several .beam files, everything is packed together with an `.avm` file bundles all your compiled `.beam` files and any assets (like images or config files) into a single package
- They all start with `#!/usr/bin/env AtomVM`
  - (Possible idea: making CLI tools with AtomVM)
- Executable .avm have a startup module

AtomVM

Extensions

AtomVM implements some extensions (most of them are prefixed with atomvm or avm):

- `:atomvm.read_priv/2` →reads a binary file stored in a loaded .avm file
- `:atomvm.posix_*` → posix functions, they mimic unistd.h ones

Closing Words on AtomVM

What's Next

New: Erlang Distribution (thanks Paul) and other 40+ additions

Soon:

- Big Integers (WIP, limited to 256-bit)
- Bitstrings (next release!)
- JIT & Ahead of Time

Future:

- More devices & peripherals (Zephyr devices, Bluetooth, Zigbee/Thread, etc…)
- Even better tooling & DevX
- Stable APIs (path to 1.0)

Releases

- **Stable Release: v0.6.6**: https://github.com/atomvm/AtomVM/releases
  - Up to OTP-27, OTP-28 support has not been backported yet
  - Pre-build binaries available
  - Well tested, focused on stability
- Development branch: main
  - https://github.com/atomvm/AtomVM/tree/main
  - Moving target, still pretty high quality

AtomVM

Contributing

- Any kind of contribution is welcome
  - Artists included, we are looking for artworks for our site and documentation :D
- Feedbacks and issues are valuable
  - Is AtomVM behaving somehow in a different way than the BEAM?
  - Are you missing any feature?
  - Is documentation or tooling usage unclear?
  - Did you notice any crash?
- Code contributions in C, Erlang, Elixir and Gleam are appreciated
  - Improvements to our documentation as well

AtomVM

Join Us

https://atomvm.org/

Discord: https://discord.gg/QA7fNjm9Nw

Telegram: https://t.me/atomvm

Documentation: https://doc.atomvm.org/

AtomVM

Thanks