AtomVM

The Workshop

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!

← Me combining Hiking and Elixir by implementing Meshtastic protocol in Elixir

AtomVM

My Journey

- I started with C++ and low level programming by doing OS development and implementing a kernel in my free time
- Meanwhile working on KDE and Plasma taught me a lot about Qt (now a very popular framework for embedded UIs) and working in big distributed teams
- I eventually started contributing to Mer project (an embedded Linux distribution) after MeeGo death
- At Ispirata Srl (my former company) we created, HemeraOS, a Mer derived Linux OS focused on IoT and simple embedded development and deployment
- We created with Riccardo Binetti, the Astarte Platform (an IoT platform), and we started writing Elixir Code
- AtomVM was the natural consequence of those experiences

AtomVM

2026

- **Since 2026, I've been working full-time on AtomVM**
- Maintaining and developing AtomVM takes significant time
- NLnet is funding the next months of development
- Long-term sustainability requires additional support
- Sponsorship helps keep AtomVM actively maintained and high quality

→ ❤️ https://github.com/sponsors/atomvm ❤️

AtomVM

Introduction

TL;DR AtomVM

- AtomVM provides compatibility with Erlang/OTP by implementing a reduced set of APIs and features in order to run on constrained devices
  - Targets can have as little as 64 KiB of RAM or maybe less
  - No operating system (bare metal) or some kind of RTOS
- AtomVM runs unmodified BEAM files up to upcoming OTP-30
  - No translation: what you write is what you flash on device

Core difference: not all APIs/modules/functions are available, but a rich subset is

AtomVM

Supported features (September 2026)

- Support for Erlang, Elixir, Gleam and anything that compiles to a BEAM file
- All Erlang datatypes (except bitstrings and native records)
  - Binaries are supported (but non-byte-aligned bitstrings not yet, there is an open PR for this)
  - Integers are limited to 256 bit magnitude + sign
- Processes, Monitors, Supervisors, etc…
- Erlang Distribution (not yet production ready, but it can be tried)
- Optional JIT and AoT compilation
- ETS core features
- The VM has all the features required to run Erlang and Elixir compilers

AtomVM

Platforms (September 2026)

- generic_unix (tested on CI: Linux, macOS, FreeBSD)
- emscripten (wasm again but on the browser)
- ESP32
- RapsberryPi microcontrollers
- STM32

WIP: WASI (wasm system interface), Zephyr, RTEMS, etc...

AtomVM

Packed BEAM files

You are targeting devices that have a bare flash memory with no filesystem.

- All BEAM files are packed into .avm files
- .avm files can also serve as container for resources and assets (such as image)
- Some targets have a concept of flash partition:
  - Usually there is a partition called boot.avm for standard libraries coming from AtomVM (that include erlang standard library, Elixir core modules, etc…)
  - There is a partition called main.avm dedicated to the application itself

AtomVM

Releases

- **Nearly Stable Release: v0.7.0-alpha.1**: https://github.com/atomvm/AtomVM/releases
  - Up to OTP-30
  - Pre-built binaries available
  - Includes Erlang Distribution, JIT and big integers
  - Fairly well tested, not yet 100% stable
- Development branch: main
  - https://github.com/atomvm/AtomVM/tree/main
  - Moving target, still pretty high quality

The Hardware

ESP32-S3, The Badge Core

- CPU: 2× 32-bit Xtensa LX7, running at 240 MHz
- Memory: 512 KiB SRAM + 2 MiB of PSRAM
- Storage: 4 MiB of FLASH storage
- Connectivity: Wi-Fi 2.4 GHz 802.11 b/g/n + Bluetooth 5 LE
- GPIOs: 45
- Interfaces: SPI, I²C, UART, USB, etc...

AtomVM

The Espressif ESP32 Family

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

If you want to build your project do not buy ESP8266 and other ancient devices pre-ESP32, they are not supported.

AtomVM

Other Supported Platforms: RaspberryPi Pico

- Pico 1/1W (RP2040) → **264 KiB of RAM,** dual core @ 133 MHz+, 2 MiB+ flash
- Pico 2/W (RP2350) → **512 KiB of RAM,** dual core @ 150 Mhz, 4 MiB+ flash
- **The 'W' models include Wi-Fi + Bluetooth**
- What's cool? Peripherals & **Programmable I/O (PIO)**

Note: Raspberry Pi Pico ≠ "classic" Raspberry Pi or Raspberry Pi Zero

AtomVM

Other Supported Platforms: STM32

- This is a very vast family with a huge number of supported micros, some are barely supported (very constrained) some others are super powerful
- Not the easiest solution if you need something with integrated connectivity
- Support improved a lot during 2026
- We support: F2 family (4 models, e.g. STM32F205), F4 (16 models, e.g. STM32F407), F7 (14 models, e.g. STM32F746), G0 (2 models, e.g. STM32G071), G4 (6 models, e.g. STM32G474), H5 (1 model, e.g. STM32H562), H7 (1 model, e.g. STM32H743), L4 (16 models, e.g. STM32L476), L5 (2 models, e.g. STM32L552), U3 (2 models, e.g. STM32U385), U5 (1 model, e.g. STM32U585), WB (1 model, e.g. STM32WB55)

Useful Concepts Before Starting

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).

When controlling a LED with a GPIO this means it's either fully on or fully off. No fading.

AtomVM

Circuits Pro-tips

Learning with the badge is easier and safer!

Don’ts

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


1

#1: Console Hello World

Let's check everything is working with a simple `IO.puts "Hello World"`.

Goals:

- Starting with AtomVM
- Learning how to install AtomVM, create your project, flash and display debug messages

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

What you Need / Setup

- 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~~
- Make sure your user has permission to access the serial port (`dialout` group on Linux)
- Your favorite Elixir setup

AtomVM

Install AtomVM on the Target Device

- Add `{:exatomvm, github: "AtomVM/exatomvm", runtime: false}` to `mix.exs` deps
- `mix deps.get`
- `mix atomvm.esp32.install --image AtomVM-esp32s3-atomgl-ipv6-libsodium-psram-elixir-nightly-0.7`
- Mix task will prompt you for installing also the following dependencies:
  - {:atomvm, "~> 0.7.0-alpha.1", runtime: false} ← Tell exatomvm which version you target
  - {:pythonx, "~> 0.4.0", runtime: false} ← Thanks to pythonx no manual install of ESP32 tools
  - {:req, "~> 0.5.0", runtime: false} ← Automatic download of ESP32 firmware
- Make the changes and install AtomVM

AtomVM

Define an Entrypoint

```
def project do
  [
    app: :my_app,
    ...
    atomvm: [start: MyApp.Main]
  ]
end
```

`MyApp.Main` must define `start/0`, which AtomVM calls when the board boots:

```
defmodule MyApp.Main do
  def start do
    IO.puts("Hello")
  end
end
```

AtomVM

Build and Run Workflow

Write as usual your Elixir code and then:

- `$ mix atomvm.esp32.flash`

(

Lot of hidden and automatic steps here, that you can manually do if you like:

- `$ mix atomvm.check` ← make sure that used funcs are available on AtomVM
- `$ mix atomvm.packbeam` ← packs everything into a flashable .avm file

)

- `$ mix atomvm.esp32.monitor` ← show all the serial messages, including `IO.puts` output

AtomVM

Resources:

uninstall.it/goatmire-2026-workshop.html

GitHub (solution):
bettio/atomvm_workshop_examples
01_hello_world


2

#2: Temperature Sensor

Let's talk to our first real peripheral and **read the temperature from the badge**.

Goals:

- Learn the basics of **I²C communication**
- Understand **device addresses and register maps**
- Use AtomVM's I²C API to communicate with a peripheral
- **Read and decode** the TMP103 temperature register
- **Print the measured temperature** to the console

AtomVM

Peripherals!

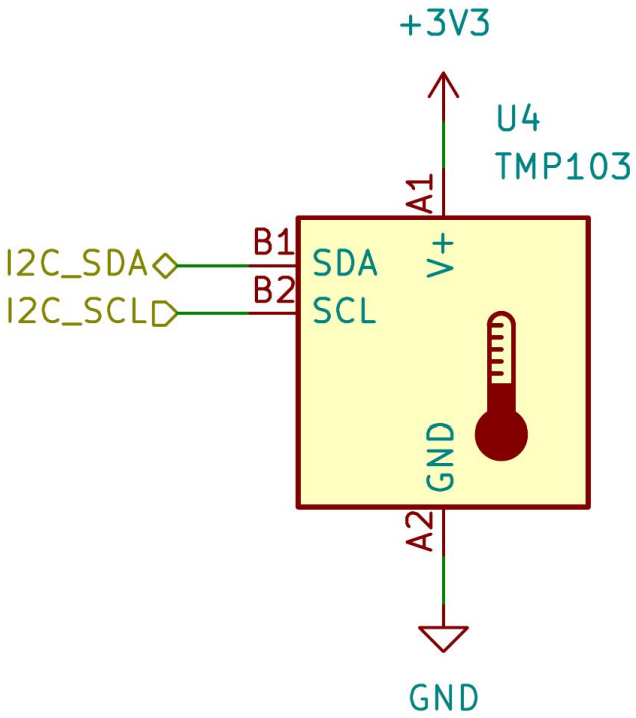
- Connecting to the outside world: peripherals!
- Usually, a *bus* lets you connect multiple devices to the same set of GPIO pins
- Many flavors
  - **I2C** → 2 wires, synchronous, slow-to-medium speeds
  - **SPI** → 4-wire bus (or more), synchronous, faster than I²C, good for displays
  - **UART** → 2-wire bus, point-to-point connection. TL;DR: a serial port
- Most sensors, displays, and other modules use one of these standard communication protocols (or "buses")

AtomVM

Physical I²C

- The bus uses 2 shared signal lines:
  - SDA: serial data
  - SCL: serial clock
- Multiple devices can share the same SDA and SCL lines
  - Each device has a 7-bit address (followed by an 8th bit indicating R/W)
  - The controller addresses one device before exchanging data
- SDA and SCL are open-drain
  - Both lines therefore require pull-up resistors
  - Pull-up resistance depends mainly on bus capacitance and clock speed
  - More devices / longer traces → more capacitance → typically stronger pull-ups

I2C address: 0x70



AtomVM

AtomVM I²C API

- i2c:open/1: configure/open bus
- i2c:read_bytes/3,4: read data / registers
- i2c:write_bytes/3,4: write data / registers
- i2c:begin_transmission/2: start explicit transaction
- i2c:write_byte/2: append byte to transaction
- i2c:end_transmission/1: send/finish transaction
- i2c:close/1: release bus

See also: https://doc.atomvm.org/v0.7.0-alpha.0/programmers-guide.html#i2c

AtomVM

TMP103 digital temperature sensor

- Connected to the badge's shared **I²C bus**
- Badge device address: **0x70**
- **8-bit temperature value**
  - resolution: **1 °C / LSB**
  - negative temperatures use **two's complement**
- Temperature range: **−40 °C to +125 °C**
- Default conversion rate: **0.25 Hz**
  - configurable: **0.25 / 1 / 4 / 8 Hz**
  - a conversion itself takes about **26 ms**
- Supports continuous, shutdown and one-shot operation.

AtomVM

Registers map

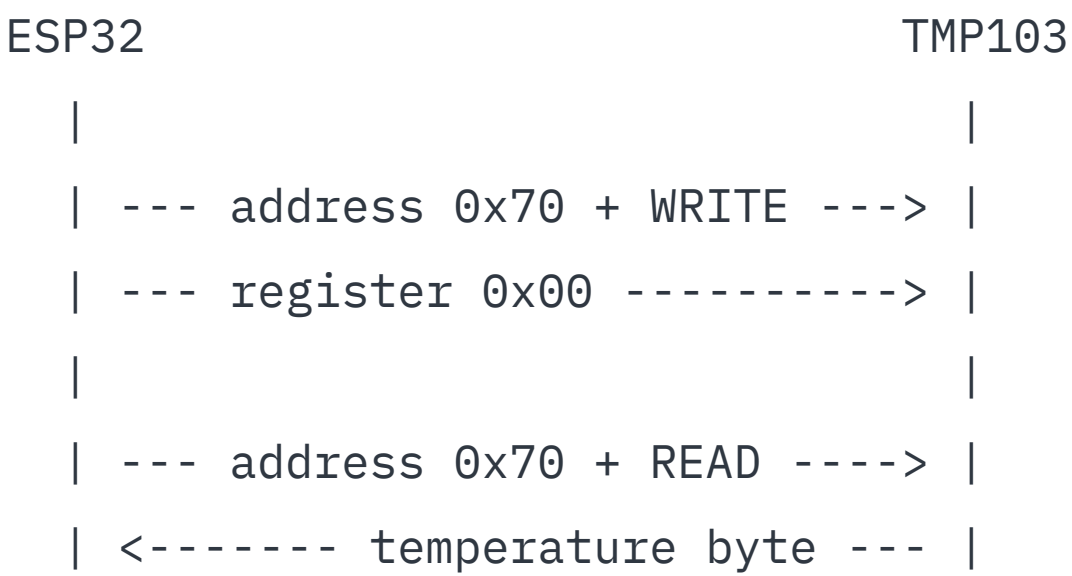
A lot of devices work according to a register model. Think about registers like table rows, and each row stores a byte.

Our sensor has the following registers:

0x00	Temperature read-only
0x01	Configuration read/write
0x02	TLOW read/write
0x03	THIGH read/write

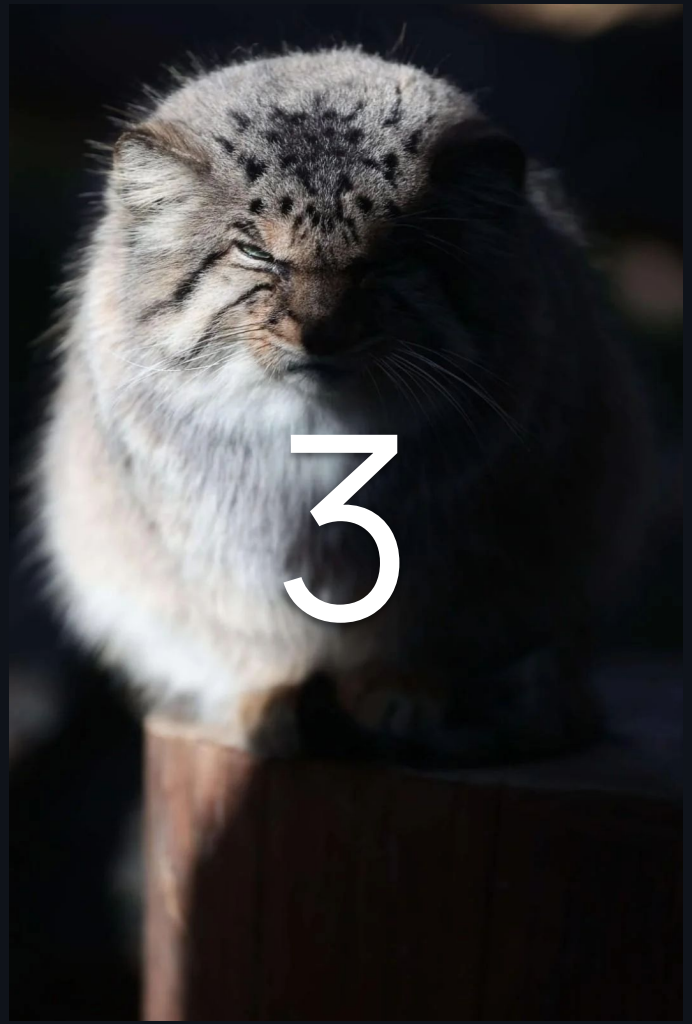
(You can find them on the datasheet)

Registers map



AtomVM

Resources: uninstall.it/goatmire-2026-workshop.html


3

#3: AtomGL

Let's put the temperature from the previous exercise on the badge display.

Goals:

- **Bring up the ST7789 display** with AtomGL
- **Render text** and shapes using a display list
- Keep acquisition and rendering separate
- Send temperature updates to an **avm_scene** process
- **Display the live temperature** on the badge

Displaying stuff: AtomGL

- A native component for rendering to a display
- Displays any list of items:

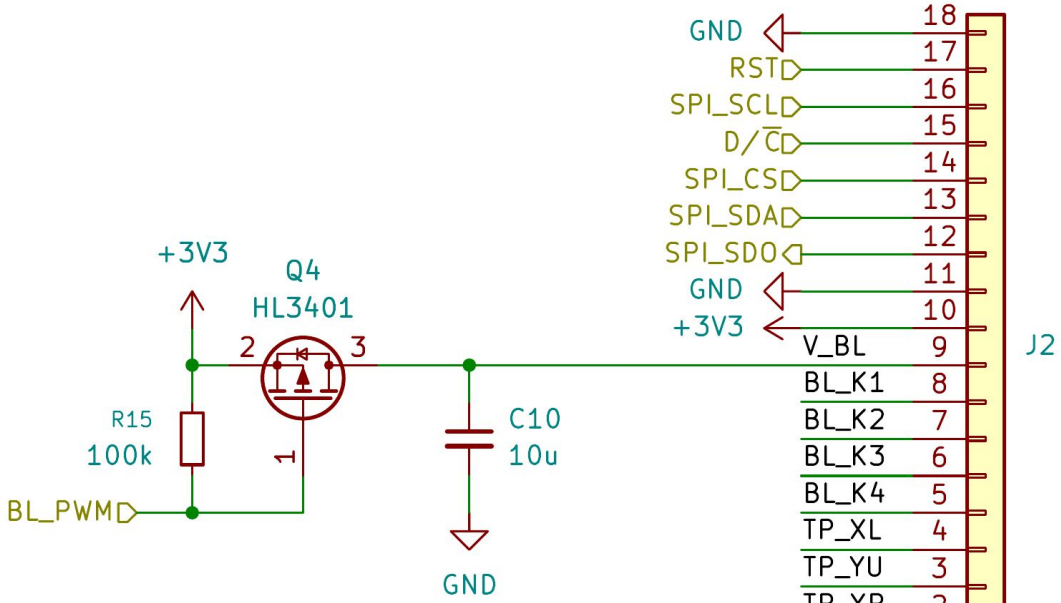
```
{:text, 16, 16, :default16px, 0xFFFFFF, 0x404040, title},
{:image, div(320 - 64, 2), div(240 - 64, 2), 0x404040, error_image}
```

- 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


+3V3
Q4
HL3401
2
3
1
R15
100k
BL_PWM
C10
10u
GND
GND
RST
SPI_SCL
D/C
SPI_CS
SPI_SDA
SPI_SDO
GND
+3V3
V_BL
BL_K1
BL_K2
BL_K3
BL_K4
TP_XL
TP_YU
TP_XR
TP_YD
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
J2

Displaying stuff: AtomGL

```
spi = :spi.open(%{
  bus_config: %{
    peripheral: "spi2",
    sclk: 5, mosi: 8, miso: 9
  },
  device_config: %{}
})
```

```
display = :erlang.open_port(
  {:spawn, "display"},
  [
    spi_host: spi,
    compatible: "sitronix,st7789",
    width: 320, height: 240, rotation: 3,
    cs: 7, dc: 4, reset: 6,
    enable_tft_invon: true, init_seq_type: "alt_gamma_2",
    backlight: 3, backlight_active: :low, backlight_enabled: true
  ]
)

HelloScene.start_link([], display_server: {:port, display})
```

AtomVM

Displaying stuff: AtomGL

```
defmodule HelloScene do
  def start_link(display_server: {:port, _display} = args) do
    :avm_scene.start_link(__MODULE__, [], args)
  end

  def init(_args) do
    send(self(), :render)
    {:ok, %{}}
  end

  def handle_info(:render, state) do
    scene = [
      {:text, 20, 40, :default16px, 0x000000, :transparent, "Hello World"},

      # Background goes last
      {:rect, 0, 0, 320, 240, 0xFFFFFF}
    ]

    {:noreply, state, [{:push, scene}]}
  end
end
```

AtomVM

AtomVM has some additions/differences

Compared to the regular BEAM, AtomVM has some extensions to the standard BEAM (usually prefixed with atomvm or avm), e.g.:

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

Idea: **we can use `:atomvm.read_priv/2` for reading raw images, so they are memory mapped from flash and we don't use additional RAM!**

AtomVM

Photo taken from: https://manulization.com/manuls/manul-16.html

Resources: uninstall.it/goatmire-2026-workshop.html

4

from: https://blog.jholko.com/2023/03/03/photograph-of-the-month-march-2023-pallas-cat-in-winter/

#4: WiFi and MQTT

Let's connect the badge to Wi-Fi and publish the temperature to an MQTT broker.

Goals:

- Connect the ESP32-S3 to a Wi-Fi network
- Connect to an MQTT 3.1.1 broker
- Handle the MQTT connection asynchronously through messages
- Reuse the TMP103 driver and publish temperature readings periodically
- Bonus: Give each badge its own topic using its factory MAC address

AtomVM

Setting up Wi-Fi

```
def connect_wifi(ssid, psk) do
  config = [
    {:ssid, ssid},
    {:psk, psk},
    {:dhcp_hostname, "atomvm-badge"}
  ]

  case :network.wait_for_sta(config, 15_000) do
    {:ok, {ip, _netmask, _gateway}} ->
      IO.inspect(ip, label: "WiFi connected")
      :ok

    {:error, reason} ->
      IO.inspect(reason, label: "WiFi failed")
      {:error, reason}
  end
end
```

AtomVM

TL;DR MQTT

- Protocol ideal for internet connected IoT devices
- Low bandwidth consumption
- Based on publish and subscribe pattern
  - Consumer (e.g. a backend) subscribes to topics e.g. `/+/temperature` (wildcard allowed!)
  - Producer (e.g. a sensor device) publishes data on different topics e.g. `/mysensor/temperature`
  - A broker such as VerneMQ, EMQX or mosquitto accepts connections from producers and consumers and dispatches messages
- You can subscribe (and test) with the following command:
  - `mosquitto_sub -h test.mosquitto.org -t 'goatmire/badge/+/temperature' -v`
  - https://mqttx.app/web-client/

AtomVM

Connect to MQTT

```
# required: {:amqtt_client, git: "https://github.com/atomvm/amqtt.git", branch: "main", sparse:
"amqtt_client"}

def start do
  :ok = connect_wifi(@ssid, @psk)

  {:ok, mqtt} =
    :amqtt_client.connect(%{
      host: @broker
    })

  receive do
    {:mqtt, ^mqtt, :connack, _info} -> :ok
  end

  loop(mqtt)
end
```

AtomVM

Publishing data on MQTT

```
defp loop(mqtt) do
  temperature = Temperature.read()

  TemperatureScene.show(temperature)

  :amqtt_client.publish(
    mqtt,
    "goatmire/badge/42/temperature",
    Integer.to_string(temperature),
    0
  )

  Process.sleep(5_000)
  loop(mqtt)
end
```

AtomVM

Photo taken from: https://manulization.com/zoos/armenia/yerevan-zoo.html

Resources: uninstall.it/goatmire-2026-workshop.html

5

#5: 🌈 on the RGB LEDs

Let's drive the badge's four SK6812 RGB LEDs and build a simple animation.

Goals:

- Understand how timing-sensitive addressable LEDs work
- Use the SPI peripheral as a hardware waveform generator
- Encode RGB values into the SK6812 wire format
- Animate all four LEDs with a continuously changing rainbow

AtomVM

Physical SPI

- SPI is a synchronous serial bus, and a typical SPI bus uses 4 signals:
  - SCLK — clock generated by the controller
  - MOSI — controller → peripheral data
  - MISO — peripheral → controller data
  - CS — selects which peripheral is active (**each device gets its own CS = chip-select line!**)
- **Multiple devices can share SCLK, MOSI and MISO**
- Data is shifted one bit per clock cycle (serialized and shifted out)
- Unlike I²C:
  - devices do not have bus addresses
  - does not require pull-up resistors like I²C
  - SPI can transfer data in both directions at the same time

AtomVM

I²C VS SPI

I²C	**SPI**
2 shared wires	usually 4 wires
addressed devices	one CS per device
half-duplex-ish	full duplex
slower	typically faster

AtomVM

AtomVM SPI API

- **spi:open/1** → configure the SPI bus and devices
- **spi:write/3** → write a binary transaction
- (spi:write_read/3 → write and read simultaneously)
- spi:close/1 → release the SPI bus

See also: https://doc.atomvm.org/v0.7.0-alpha.0/programmers-guide.html#spi

AtomVM

RGB LEDs / SK6812

The badge has four addressable RGB LEDs

- They are connected in a chain
- Each SK6812 contains:
  - a tiny controller
  - red, green and blue LED emitters
- The ESP32 sends one stream of bits to the first LED
- Each LED consumes its own color data and forwards the rest:
  - ESP32 GPIO14 → LED 0 → LED 1 → LED 2 → LED 3

SK6812: not a normal LED

A normal LED is basically ON or OFF: drive a GPIO HIGH or LOW.

An SK6812 is different: each LED contains its own tiny controller and expects a precisely timed digital data stream describing its RGB color.

- The data line alternates between HIGH and LOW
- A 0 and a 1 differ mainly by how long the signal stays HIGH:
  - 0 → HIGH for about 0.3 µs, then LOW for about 0.9 µs
  - 1 → HIGH for about 0.6 µs, then LOW for about 0.6 µs

AtomVM

Using SPI as a precise waveform generator

- Use MOSI as the LED data line (GPIO14 → DIN)
- MISO is unused + we don't route SCLK and CS to a pin (sclk: -1, cs: -1)
  - Internally, the SPI peripheral still uses its clock to shift MOSI at an exact rate →we'll use 3.2 MHz (1.25 µs / 4 = 312.5 ns → 3.2 MHz)
- Encoding:
  - 0 → 1000
  - 1 → 1100
  - [LED0: G R B][LED1: G R B][LED2: G R B][LED3: G R B]
  - after sending all pixels, hold the line LOW long enough to latch the frame

AtomVM

Photo taken from: https://manulization.com/manuls/oto-1/photos-and-videos/e8uSCMS1AQAjKHyp

Resources: uninstall.it/goatmire-2026-workshop.html

6

#6: Buttons and Interrupts

Let's turn button presses into Elixir messages without constantly polling GPIOs.

Goals:

- Configure GPIOs as digital inputs
- Understand why continuously polling while idle wastes CPU time and power
- Use GPIO interrupts to react when a button is pressed
- Receive hardware events as ordinary Elixir messages
- Handle press/release events inside a GenServer
- Extend the same idea to the badge's keyboard matrix

Reading a button

The badge's BOOT button is connected to GPIO0

- A pull-up[1] keeps GPIO0 HIGH while the button is released
- Pressing the button connects GPIO0 to GND
- Therefore the button is active-low:
  - HIGH → released
  - LOW → pressed

Configure the GPIO as an input, then read its digital level

[1] **A digital input needs a defined path to VCC or GND: use a pull-up or pull-down. Without one, the pin floats and its value is undefined.**

AtomVM

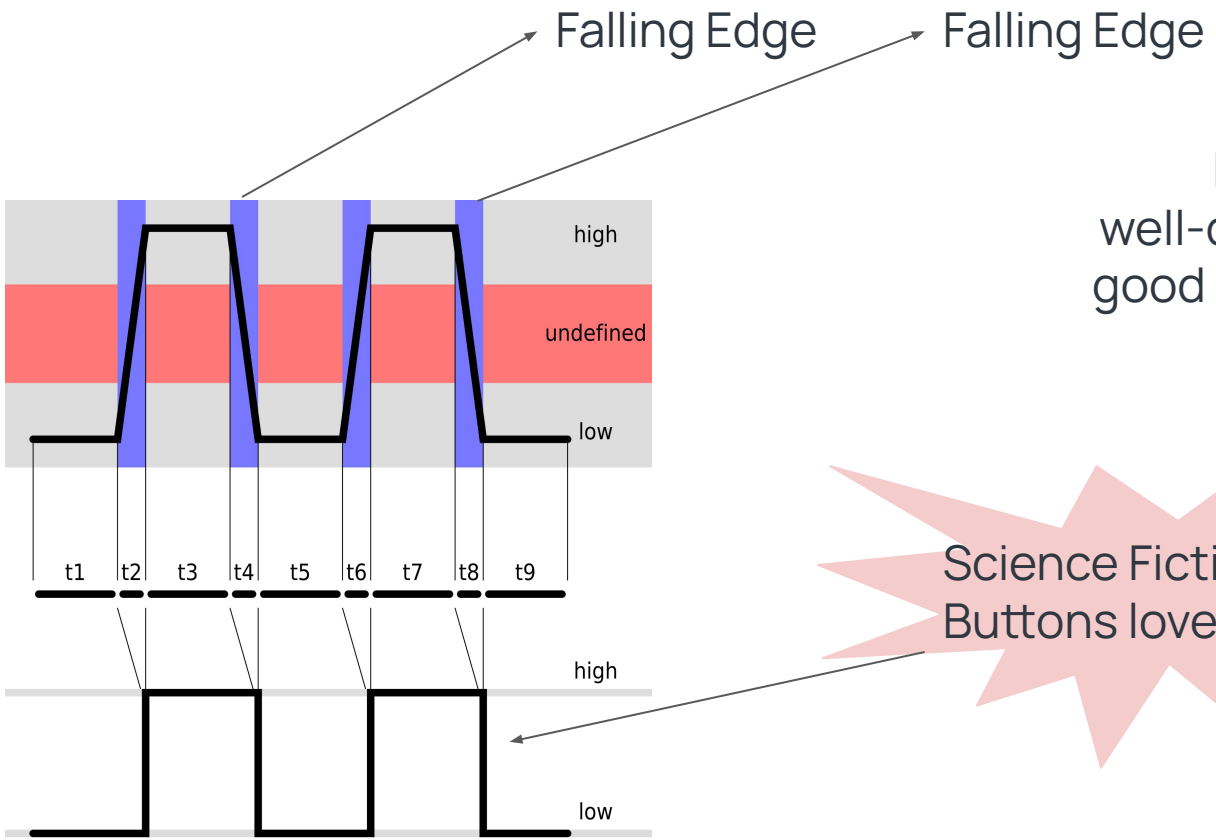
Waiting for a button press

We could repeatedly call `GPIO.digital_read/1` *BUT* **most of the time nothing happens.**

- Instead, the GPIO hardware can detect a signal transition
- Our active-low button produces a falling edge when pressed
- We arm an interrupt for that edge
- AtomVM delivers the hardware event as an ordinary Erlang message

AtomVM

Falling Edge



Falling Edge is a well-defined point in time, good for sending an Erlang Message.

Science Fiction Here: Buttons love bouncing!

AtomVM

GPIO interrupts in AtomVM

The API we can use:

- `GPIO.open/0` → open the GPIO interrupt driver
- `GPIO.set_pin_mode/2` → configure the pin as an input
- `GPIO.digital_read/1` → read its current level
- `GPIO.set_int/3` → arm an edge interrupt
- `GPIO.remove_int/2` → disarm the interrupt

Interrupts arrive as {:gpio_interrupt, pin}

AtomVM

One button is easy. What about a keyboard?

- One GPIO per key would require a lot of pins
- The badge keyboard is arranged as a 6 × 13 matrix
- That requires only 6 row + 13 column GPIOs
- Rows are open-drain outputs; columns are inputs with pull-ups
- Drive one row LOW
- Any pressed key in that row pulls its column LOW
- Read the columns, then move to the next row

Don’t scan when nobody is typing

- Idle: hold all rows LOW and arm interrupts on the 13 columns
- Any key press pulls a column LOW → falling-edge interrupt
- The interrupt only tells us something was pressed
- Disable interrupts and scan the matrix to find the actual key
- While keys are held, scan every 20 ms for press/release changes
- Once every key is released, re-arm interrupts and return to idle

TL;DR: one active-low button → edge interrupt → Erlang message → matrix scanning → interrupt-driven keyboard.

AtomVM

Photo taken from: https://manulization.com/manuls/nar/photos-and-videos/cC4TztS8fuBOBSOI

Resources: uninstall.it/goatmire-2026-workshop.html

7

#7: Analog Input / Battery Voltage

Let's measure a real voltage with the ESP32-S3 ADC.

Goals:

- Use AtomVM's ADC API to acquire and sample a channel
- Measure the badge's battery voltage and USB VBUS
- Understand why the board uses a voltage divider
- Use attenuation to select a useful ADC input range
- Average multiple samples to reduce ADC noise
- Convert the measured pin voltage back to the actual rail voltage

AtomVM

ADC: turning a voltage into a number

A normal digital GPIO does not measure voltage.

You need an ADC. ADC = **Analog-to-Digital Converter**

- A **digital GPIO only classifies the input as LOW or HIGH**
- An **ADC measures a voltage within a configured input range**
- The input is sampled at a specific instant
- The **sampled voltage is quantized into an integer value**
- That **integer must be scaled/calibrated to obtain a voltage**
- Values outside the measurable range saturate
- Real conversions include quantization error and noise

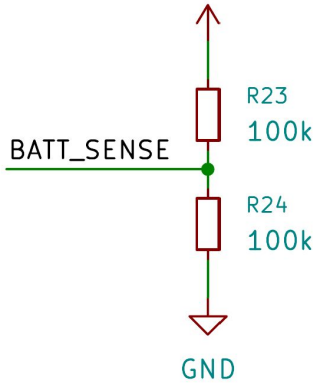
How to Measure “Any” Voltage

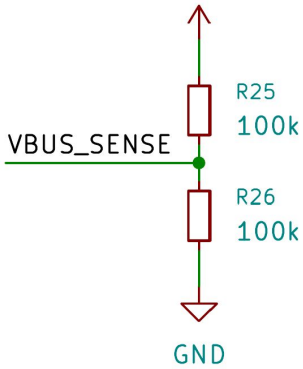
VBUS can reach 5 V, so it cannot be connected directly to this ADC input

Trick: use a voltage divider. In our case:

$$V_{out} = V_{in} \cdot \frac{R_2}{R_1 + R_2} \longrightarrow R_1 = R_2 \Rightarrow V_{out} = \frac{V_{in}}{2}$$

The ADC input draws negligible current. BATT_SENSE and VBUS_SENSE are “observation points”.





AtomVM

ADC range, resolution and attenuation

- After the voltage divider, the ADC sees at most about 2.5 V
- We use the ESP32-S3 ADC at 12-bit resolution → 4096 levels (0–4095)
- We need an input range that includes 0–2.5 V
- Per the datasheet, 12 dB attenuation gives a range up to about 2.9 V
- Therefore the exercise uses :bit_max + :db_12
- Ideal resolution is about 0.71 mV per ADC step
- With the external 1:2 divider, that is about 1.42 mV per step on the original rail voltage

AtomVM ADC API

- Init:

  `Esp.ADC.init/0` → initialize the ADC peripheral

- Choose the conversion resolution and attenuation:

  `Esp.ADC.acquire/4` → configure a GPIO as an ADC channel

- Read a Value:

  `Esp.ADC.sample/3` → perform one or more ADC conversions

  - `:raw` → return the integer ADC value
  - `:voltage` → return the calibrated value in mV
  - `{:samples, n}` → average multiple conversions to reduce noise

AtomVM

Join Us

https://atomvm.org/

Discord: https://discord.gg/QA7fNjm9Nw

Telegram: https://t.me/atomvm

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

AtomVM