AtomVM documentation

Release 0.7.0-dev+git.10429ba9

github.com/atomvm

June 12, 2025

Welcome to AtomVM, the Erlang virtual machine for IoT devices!

AtomVM is a lightweight implementation of the the Bogdan Erlang Abstract Machine (_aka_, the BEAM), a virtual machine that can execute byte-code instructions compiled from Erlang or Elixir source code. AtomVM supports a limited but functional subset of the BEAM opcodes, and also includes a small subset of the Erlang/OTP standard libraries, all optimized to run on tiny micro-controllers. With AtomVM, you can write your IoT applications in a functional programming language, using a modern actor-based concurrency model, making them vastly easier to write and understand!

AtomVM includes many advanced features, including process spawning, monitoring, message passing, pre-emptive scheduling, and efficient garbage collection. It can also interface directly with peripherals and protocols supported on micro-controllers, such as GPIO, I2C, SPI, and UART. It also supports WiFi networking on devices that support it, such as the Espressif ESP32. All of this on a device that can cost as little as $2!

Warning AtomVM is currently in v0.x stage. Software may contain bugs and should not be used for mission-critical applications. Application Programming Interfaces may change without warning.

1 Welcome to AtomVM!

Welcome to AtomVM, the Erlang virtual machine for IoT devices!

1.1 What is AtomVM?

AtomVM is a ground-up implementation of the Bogdan Erlang Abstract Machine (a.k.a the BEAM) and is designed specifically to run on small systems, such as the Espressif ESP32 and ST Microelectronics STM32 micro-controllers. It allows developers to implement IoT applications in the Erlang or Elixir programming languages and to deploy those applications onto tiny devices. (Users may also target their applications for fully-fledged operating systems, such as Linux, FreeBSD, and MacOS, though in most cases deployment to traditional computers is done for development and testing purposes, only.)

AtomVM features include:

- An Erlang runtime, capable of executing bytecode instructions in compiled BEAM files;
- Support for all the major Erlang and Elixir types, including integers, strings, lists, maps, binaries, Enums, and more;
- A memory-managed environment, with efficient garbage collection and shared data, where permissible;
- Support for truly functional programming languages, making your programs easier to understand and debug;
- A concurrency-oriented platform, allowing users to spawn, monitor, and communicate with lightweight processes, making it easy for your IoT devices to perform tasks simultaneously;
- Support for symmetric multi-processing (SMP); leverage all available cores on platforms that support it (e.g., ESP32) without any code changes;
- A rich set of networking APIs, for writing robust IoT applications that communicate over IP networks;
- A rich set of APIs for interfacing with standard device protocols, such as GPIO, I2C, SPI, and UART;
- And more!

1.2 Why Erlang/Elixir?

The environments on which AtomVM applications are deployed are significantly more constrained than typical programming environments. For example, the typical ESP32 ships with 520K of RAM and 4MB of flash storage, roughly the specs of a mid 1980s desktop computer. Moreover, most micro-controller environments do not support native POSIX APIs for interfacing with an operating system, and in many cases, common operating system abstractions, such as processes, threads, or files, are simply unavailable.

However, because the BEAM is provides a pre-emptive multitasking environment for your applications, many of the common operating system abstractions, particularly involving threading and concurrency, are simply not needed. As concurrently-oriented languages, Erlang and Elixir support lightweight "processes", with message passing as the mechanism for inter-(erlang)process communication, pre-emptive multi-tasking, and per-process heap allocation and garbage collection.

In many ways, the programming model for Erlang and Elixir is closer to that of an operating system and multiple concurrent processes running on it, where operating system processes are single execution units, communicate through message passing (signals), and don't share any state with one another. Contrast that with most popular programming languages today (C, C++, Java, Python, etc), which use threading abstractions to achieve concurrency within a single memory space, and which subsequently require close attention to cases in which multiple CPUs operate on a shared region of memory, requiring threads, locks, semaphores, and so forth.

As an implementation of the BEAM, AtomVM provides a modern, memory managed, and concurrency-oriented environment for developing applications on small devices. This makes writing concurrent code for micro-controllers (e.g., and application that reads sensor data, services HTTP requests, and updates the system clock, all at the same time) incredibly simple and natural – far easier writing programs that use concurrency than C, C++, or even, for example, Micropython.

In addition, because it is targeted for micro-controller environments, AtomVM provides interfaces for integrating with features commonly seen on micro-controllers, such as GPIO pins, analog-to-digital conversion, and common industry peripheral interfaces, such as I2C, SPI, and UART, making AtomVM a rich platform for developing IoT applications.

Finally, one of the exciting aspects about modern micro-controllers, such as the ESP32, is their integration with modern networking technologies, such as WiFi and Bluetooth. AtomVM leverages Erlang and Elixir's natural affinity with telecommunications technologies to open up further possibilities for developing networked and wireless IoT devices.

We think you will agree that AtomVM provides a compelling environment not only for Erlang and Elixir development, but also as a home for interesting and fun IoT projects.

1.3 Design Philosophy

AtomVM is designed from the start to run on small, cheap embedded devices, where system resources (memory, cpu, storage) are tightly constrained. The smallest environment in which AtomVM runs has around 512k of addressable RAM, some of which is used by the underlying runtime (FreeRTOS), and some of which is used by the AtomVM virtual machine, itself, leaving even less RAM for your own applications. Where there is a tradeoff between memory consumption and performance, minimizing memory consumption (and heap fragmentation) always wins.

From the developer's point of view, AtomVM is designed to make use of the existing tool chain from the Erlang and Elixir ecosystems. This includes the Erlang and Elixir compilers, which will compile Erlang and Elixir source code to BEAM bytecode. Where possible, AtomVM makes use of existing tool chains to reduce the amount of unnecessary features in AtomVM, thus reducing complexity, as well as the amount of system resources in use by the runtime. AtomVM is designed to be as small and lean as possible, providing as many resources to user applications, as possible.

1.4 Licensing

AtomVM is licensed under the terms of the Apache2 and LGPLv2 licenses.

1.5 Source Code

The AtomVM Github Repository contains the AtomVM source code, including the AtomVM virtual machine and core libraries. The AtomVM Build Instructions contains instructions for building AtomVM for Generic UNIX, ESP32, and STM32 platforms.

1.6 Contributing

The AtomVM community welcomes contributions to the AtomVM code base and upstream and downstream projects. Please see the contributing guidelines for information about how to contribute.

AtomVM developers can be reached on the #AtomVM discord server (rarely used) or on Telegram at AtomVM - Erlang and Elixir on Microcontrollers (this is where we are most active).

1.7 Where to go from here

The following guides provide more detailed information about getting started with the AtomVM virtual machine, how to develop and deploy applications, and implementation information, for anyone interested in getting more involved:

- Getting Started Guide
- Programmers Guide
- Example Programs
- Build Instructions

Chapter 2

Release Notes

Welcome to AtomVM `0.7.0-dev+git.10429ba9`

These release notes provide version information about the current release of AtomVM.

See also

For a detailed list of changes since the last release consult the Changelog.

2.1 Required Software

The following software is required to develop Erlang or Elixir applications on AtomVM:

- An Erlang/OTP compiler (`erlc`)
- The Elixir runtime, if developing Elixir applications.
- (recommended) For Erlang programs, `rebar3`
- (recommended) For Elixir programs, `mix`, which ships with the Elixir runtime.

AtomVM will run BEAM files that have been compiled using the following Erlang and Elixir versions:

Erlang Version	Elixir Version
?	?
?	?
?	?
?	?
?	?
?	?
?	?
?	?

Note Versions of Elixir that are compatible with a particular OTP version may work. This table reflects the versions that are tested.

Not all BEAM instructions are supported for every Erlang and Elixir compiler. For details about which instructions are supported, see the `src/libAtomVM/opcodes.h` header file in the AtomVM github repository corresponding to the current release.

For detailed information about features and bug fixes in the current release, see the AtomVM Change Log. For information about how to update from previous versions of AtomVM, see the AtomVM Updating page.

2.1.1 ESP32 Support

AtomVM supports deployment on the Espressif ESP32 family of architectures.

To run applications built for AtomVM on the ESP32 platform you will need:

- The `esptool` program, for flashing the AtomVM image and AtomVM programs to ESP32 MCUs.
- A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.

AtomVM currently supports the following Espressif ESP SoCs:

Espressif SoCs	AtomVM support
ESP32	?
ESP32c2	?
ESP32c3	?
ESP32c6	?
ESP32h2	?
ESP32s2	?
ESP32s3	?
ESP32p4 Datasheet Pending	?

AtomVM currently supports the following versions of ESP-IDF:

IDF SDK supported versions	AtomVM support
ESP-IDF v5.0	?
ESP-IDF v5.1	?
ESP-IDF v5.2	?
ESP-IDF v5.3	?
ESP-IDF v5.4	?

Building the AtomVM virtual machine for ESP32 is optional. In most cases, you can simply download a release image from the AtomVM release repository. If you wish to work on development of the VM or use one on the additional drivers that are available in the AtomVM repositories you will to build AtomVM from source. See the Build Instructions for information about how to build AtomVM from source code. We recommend you to use the latest subminor (patch) versions for source builds. You can check the current version used for testing in the esp32-build.yaml workflow.

2.1.2 STM32 Support

AtomVM supports deployment on the STMicroelectronics STM32 architecture.

AtomVM has been tested on the following development boards:

STM32 Development Boards	AtomVM support
Nucleo-F429ZI	?
STM32F4Discovery	?
BlackPill V2.0	?

Due to the proliferation of boards for the STMicroelectronics STM32 platform, AtomVM does not currently support pre-build binaries for STM32. In order to deploy AtomVM to the STM32 platform, you will need to build AtomVM for STM32 from source. See the Build Instructions for information about how to build AtomVM from source code.

Note AtomVM tests this build on the latest Ubuntu github runner.

2.1.3 Raspberry Pi Pico Support

AtomVM supports deployment on the Raspberry Pico RP2 architecture.

AtomVM currently supports the following Raspberry Pico development boards:

Development Board	AtomVM support
Raspberry Pico and Pico H	?
Raspberry Pico W and Pico WH	?
Raspberry Pico 2 and Pico 2H	?

Building the AtomVM virtual machine for Raspberry Pico is optional. In most cases, you can simply download a release image from the AtomVM release repository. If you wish to work on development of the VM or use one on the additional drivers that are available in the AtomVM repositories you will to build AtomVM from source. See the Build Instructions for information about how to build AtomVM from source code.

3 Getting Started Guide

Welcome to the AtomVM Getting Started Guide. This document is intended to get you started so that you can run Erlang or Elixir programs on the AtomVM platform as quickly as possible.

In order to do so, you will need to provision your device (depending on the device type) with the AtomVM virtual machine. Typically, you only need to do this once (or at least once per release of the VM you would like to use). Once the VM is provisioned on the device, you can then deploy your application onto the device, and we expect this process to your typical "deploy, test, debug" development lifecycle. The subsequent chapter on AtomVM Tooling will help you understand that process.

The getting started is broken up into the following sections:

- Getting Started on the ESP32 platform
- Getting Started on the STM32 platform
- Getting Started on the Raspberry Pi RP2
- Getting Started on the Generic UNIX platform
- Getting Started with AtomVM WebAssembly

Please use the appropriate section for the device type you intend to use.

3.1 Getting Started on the ESP32 platform

The AtomVM virtual machine is supported on the Espressif ESP32 platform, allowing users to write Erlang and Elixir programs and deploy them to the ESP32 micro-controller. For specific information about which ESP32 boards and chip-sets are supported, please refer to the AtomVM Release Notes.

These instructions cover how to provision the AtomVM virtual machine flashed to your ESP32 device.

For most applications, you should only need to install the VM once (or at least once per desired AtomVM release). Once the VM is uploaded, you can then begin development of Erlang or Elixir applications, which can then be flashed as part of your routine development cycle.

3.1.1 ESP32 Requirements

Deployment of AtomVM on the ESP32 platform requires the following components:

- A computer running MacOS or Linux (Windows support is not currently supported);
- An ESP32 (including ESP32-S2, ESP32-S3, ESP32-C2, ESP32-C3, ESP32-C6, ESP32-H2) module with a USB/UART connector (typically part of an ESP32 development board);
- A USB cable capable of connecting the ESP32 module or board to your development machine (laptop or PC);
- The `esptool` program, for flashing the AtomVM image and AtomVM programs;
- An Erlang/OTP;
- A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.
- (recommended) For Erlang programs, `rebar3`;
- (recommended) For Elixir programs, `mix`, which ships with the Elixir runtime;

See also

For information about specific versions of required software, see the AtomVM Release Notes.

3.1.2 ESP32 Deployment Overview

The ES32 AtomVM virtual machine is an IDF application that runs on the ESP32 platform. As an IDF application, it provides the object code to boot the ESP device and execute the AtomVM virtual machine code, which in turn is responsible for execution of an Erlang/Elixir application.

The AtomVM virtual machine is implemented in C, and the AtomVM binary image contains the binary object code compiled from C source files, as well as the ESP boot loader and partition map, which tells the ESP32 how the flash module is laid out.

AtomVM developers will typically write their applications in Erlang or Elixir. These source files are compiled into BEAM bytecode, which is then assembled into AtomVM "packbeam" (`.avm`) files. This packbeam file is flashed onto the ESP32 device, starting at the data partition address `0x210000`. When AtomVM starts, it will look in this partition for the first occurrence of a BEAM module that exports a `start/0` function. Once that module is located, execution of the BEAM bytecode will commence at that point.

The following diagram provides a simplified overview of the layout of the AtomVM virtual machine and Erlang/Elixir applications on the ESP32 flash module.

```
|                |
+----------------+  ----------- 0x0 | 0x1000 | 0x2000 (varies by esp32 flavor)
| boot loader    |           ^
+----------------+           |
| partition map  |           | AtomVM
+----------------+           | binary
|                |           | image
|   AtomVM       |           |
|   Virtual      |           |
|   Machine      |           |
|                |           v
+----------------+  ----------- 0x210000 for thin images or
|                |           ^  0x250000 for images with Elixir modules
|                |           |
|     data       |           | Erlang/Elixir
|   partition    |           | Application
|                |           |
|                |           v
+----------------+  ----------- end
```

Deploying an AtomVM application to an ESP32 device typically involved two steps:

1. Connecting the ESP32 device;
2. Deploying the AtomVM virtual machine;
3. Deploying an AtomVM application (typically an iterative process)

These steps are described in more detail below.

3.1.3 Connecting the ESP32 device

Connect the ESP32 to your development machine (e.g., laptop or PC) via a USB cable.

```
+----------------+
| laptop or PC   |
|                |           +-------+
|                | USB       |       |
|                x-----------x       |
|                |           |       |
|                |           +-------+
+----------------+           ESP32
```

Important There are a wide variety of ESP32 modules, ranging from home-made breadboard solutions to all-in-one development boards. For simplicity, we assume a development board that can both be powered by a USB cable and which can be simultaneously flashed using the same cable, e.g., the Espressif ESP32 DevKit.

Consult your local development board documentation for instructions about how to connect your device to your development machine.

3.1.4 Deploying the ESP32 AtomVM virtual machine

The following methods can be used to deploy the AtomVM virtual machine to an ESP32 device:

1. Flashing a binary image;
2. Building from source.

Flashing a binary image to ESP32

Flashing the ESP32 using a pre-built binary image is by far the easiest path to getting started with development on the ESP32. Binary images contain the virtual machine image and all of the necessary components to run your application.

We recommend first erasing any existing applications on the ESP32 device. E.g.,

```
$ esptool.py --chip auto --port /dev/ttyUSB0 --baud 921600 erase_flash
```

Note Specify the device port and baud settings and AtomVM image name to suit your particular environment. A baud rate of 921600 works well for most ESP32 devices, some can work reliably at higher rates of 1500000, or even 2000000, but some devices (especially those with a 26Mhz crystal frequency, rather than the more common 40 Mhz crystal) may need to use a slower baud rate such as 115200.

Download the latest release image for ESP32.

This image will generally take the form:

`Atomvm-<esp32-soc>-<atomvm-version>.img`

For example:

`Atomvm-esp32-v0.6.0.img`

You will also find the sha256 hash for this file, which you should verify using the `sha256sum` command on your local operating system.

Warning Alpha and Beta images may be unstable and may result in unpredictable behavior. You can help solve these bugs by opening a detailed issue on GitHub, if you encounter such problems.

Finally, use the `esptool.py` command to flash the image to the bootloader start address `0x1000` on the ESP32. E.g.,

```
$ esptool.py \
--chip auto \
--port /dev/ttyUSB0 --baud 921600 \
--before default_reset --after hard_reset \
write_flash -u \
--flash_mode dio --flash_freq 40m --flash_size detect \
0x1000 \
/path/to/Atomvm-esp32-v0.6.0.img
```

Attention! A baud rate of 921600 works well for most ESP32 devices, some can work reliably at higher rates of 1500000, or even 2000000, but some devices (especially those with a 26Mhz crystal frequency, rather than the more common 40 Mhz crystal) may need to use a slower baud rate such as 115200.

The chart below lists the bootloader offset for the various ESP32 family of chips:

Chipset	Bootloader offset
ESP32	`0x1000`
ESP32-S2	`0x1000`
ESP32-S3	`0x0`
ESP32-C2	`0x0`
ESP32-C3	`0x0`
ESP32-C6	`0x0`
ESP32-H2	`0x0`
ESP32-P4	`0x2000`

Once completed, your ESP32 device is ready to run Erlang or Elixir programs targeted for AtomVM.

Building for ESP32 from source

You may optionally build AtomVM from source and deploy the AtomVM virtual machine to your ESP32 device manually. Building AtomVM from source is slightly more involved, as it requires the installation of the Espressif IDF SDK and tool chain and is typically recommended only for users who are doing development on the AtomVM virtual machine, or for developers implementing custom Nifs or ports.

Instructions for building AtomVM from source are covered in the AtomVM Build Instructions

3.1.5 Deploying an AtomVM application for ESP32

An AtomVM application is a collection of BEAM files, which have been compiled using the Erlang or Elixir compiler. These BEAM files are assembled into an AtomVM "packbeam" (`.avm`) file, which in turn is flashed to the `main` data partition on the ESP32 flash module, starting at address `0x210000` if you are using a thin image, or `0x250000` for images with Elixir support.

When the AtomVM virtual machine starts, it will search for the first module that contains an exported `start/0` function in this partition, and it will begin execution of the BEAM bytecode at that function.

AtomVM applications can be written in Erlang or Elixir, or a combination of both. The AtomVM community has provided tooling for both platforms, making deployment of AtomVM applications as seamless as possible.

For information about how to flash your application to your ESP32, see the AtomVM Tooling chapter.

3.2 Getting Started on the STM32 platform

AtomVM can run on a wide variety of STM32 chip-sets available from STMicroelectronics. The support is not nearly as mature as for the ESP32 platform, but work is ongoing, and pull requests are always welcome. At this time AtomVM will work on any board with a minimum of around 128KB ram and 512KB (1M recommended) flash. Simple applications and tests have been successfully run on a stm32f411ceu6 (A.K.A. Black Pill V2). These minimum requirements may need to be raised as platform support matures.

3.2.1 STM32 Requirements

Deployment of AtomVM on the STM32 platform requires the following components:

- A computer running MacOS or Linux (Windows is not currently supported);
- An stm32 board and a USB/UART connector (these are built into some boards such as the Nucleo product line) and a minimum of 512k (1M recommended) of flash and a recommended minimum of 100k RAM;
- A USB cable capable of connecting the STM32 module or board to your development machine (laptop or PC);
- `st-flash` via stlink, to flash both AtomVM and your packed AVM applications. Make sure to follow its installation procedure before proceeding further.
- A st-link v2 or st-link v3 device (typically already included on Nucleo and Discovery boards), is needed for flashing and optional jtag debugging.
- A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.
- (recommended) For Erlang programs, `rebar3`;
- (recommended) For Elixir programs, `mix`, which ships with the Elixir runtime;

3.2.2 Deploying the STM32 AtomVM virtual machine

The following methods can be used to deploy the AtomVM virtual machine to an STM32 device:

1. Building from source.

Attention! Due to the very large number of supported chip-sets and the wide variety of board configurations, and the code changes required to support them, pre-built binaries for the stm32 platform are not currently available.

Consult the STM32 Build Instructions to create a binary compatible with your board.

Flashing a binary image to STM32

Once you have created an STM32 binary image, you can flash the image to your STM32 device using the `st-flash` application.

To flash your image, use the following command:

```
$ st-flash --reset write AtomVM-stm32f407vgt6.bin 0x8000000
```

Congratulations! You have now flashed the AtomVM VM image onto your STM32 device!

Important AtomVM expects to find the AVM at the address 0x8080000. On a STM32 Discovery board this means that the 1MB of flash will be split in 512KB available for the program and 512KB available for the packed AVM. For devices with only 512KB of flash the application address is 0x8060000, leaving 128KB of application flash available.

Console Printing

By default, stdout and stderr are printed on USART2. On the STM32F4Discovery board, you can see them using a TTL-USB with the TX pin connected to board's pin PA2 (USART2 RX). Baudrate is 115200 and serial transmission is 8N1 with no flow control.

For Nucleo boards the on board USB-COM to USART may be used by configuring your build with a BOARD parameter, see the STM32 Build Instructions for Configuring the Console.

3.2.3 Deploying an AtomVM application for STM32

An AtomVM application is a collection of BEAM files, which have been compiled using the Erlang or Elixir compiler. These BEAM files are assembled into an AtomVM "packbeam" (`.avm`) file, which in turn is flashed to the `main` data partition on the STM32 flash module, starting at address `0x8080000`, for boards with 512KB of flash the address is `0x8060000`.

When the AtomVM virtual machine starts, it will search for the first module that contains an exported `start/0` function in this partition, and it will begin execution of the BEAM bytecode at that function.

AtomVM applications can be written in Erlang or Elixir, or a combination of both. The AtomVM community has provided tooling for both platforms, making deployment of AtomVM applications as seamless as possible.

For information about how to flash your application to your STM32, see the AtomVM Tooling chapter.

3.3 Getting Started on the Raspberry Pi RP2

AtomVM supports deployment of the VM and applications onto boards based on Raspberry Pi RP2 socs, including RP2040 and RP2350. This includes Raspberry Pi Pico and Pico W, as well as Raspberry Pi Pico 2 and Pico 2 W boards. For information about supported boards, please refer to the AtomVM Release Notes.

The following instructions show you how to install the AtomVM onto one of the Raspberry Pi Pico boards.

3.3.1 Pico and Pico 2 Requirements

Deployment of AtomVM on the Raspberry Pi Pico and Pico 2 platform requires the following components:

- A computer running MacOS or Linux (Windows support is not currently supported);
- A Raspberry Pi Pico or Pico 2 board;
- A USB cable capable of connecting the board to your development machine (laptop or PC);
- A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.
- (recommended) For Erlang programs, `rebar3`;
- (recommended) For Elixir programs, `mix`, which ships with the Elixir runtime;
- (recommended) `picotool`, useful for resetting the device into `BOOTSEL` or `application` mode, optionally used by the `atomvm_rebar3_plugin` (if available in env $PATH) for disconnecting active `screen` sessions when attempting to flash when still connected.

3.3.2 Deploying the Pico or Pico 2 AtomVM virtual machine

The following methods can be used to deploy the AtomVM virtual machine to a Raspberry Pico device:

1. Flashing a binary image;
2. Building from source.

Flashing a binary image to Pico

Flashing the Raspberry Pi Pico or Pico 2 using a pre-built binary image is by far the easiest path to getting started with development on this platform. Binary images contain the virtual machine image and all of the necessary components to run your application.

Download the latest release image for Raspberry Pi Pico or Raspberry Pi Pico 2.

This image will generally take the form:

```
Atomvm-<raspberry-pico-board>-<atomvm-version>.uf2
```

For example:

```
Atomvm-pico-v0.7.0.uf2
```

or

```
Atomvm-pico2-v0.7.0.uf2
```

You will also find the sha256 hash for this file, which you should verify using the `sha256sum` command on your local operating system.

You will also need a copy of the AtomVM core libraries, which include all of the compiled Erlang and Elixir needed to run parts of the VM.

This library will generally take the form:

```
atomvmlib-<raspberry-pico-board>-<atomvm-version>.uf2
```

For example:

```
atomvmlib-pico-v0.7.0.uf2
```

You will also find the sha256 hash for this file, which you should verify using the `sha256sum` command on your local operating system.

To flash your Raspberry Pico, you will need to undertake a few steps that interact with your operating file system.

Important It is important that you downloads the `.uf2` versions of these files for the Raspberry Pi Pico or Pico 2 platform. It is also important to download the Pico or the Pico2 files depending on your board.

For each of the above files, you will start your Raspberry Pi Pico or Pico 2 in bootloader mode by pressing the `BOOTSEL` button on the board, while powering on the device. Doing so will automatically boot the device and mount the Raspberry Pi Pico on to your file system as a USB device.

You can then use normal operating system commands (such as `cp`, or even drag-and-drop) to copy the above files to the mounted USB volume.

Note, however, that in general the USB device will auto-unmount after each file has been copied, so you will need to repeat the procedure for each of the above two files.

On most Linux systems, the Raspberry Pi Pico will be mounted at `/run/media/${USER}/RPI-RP2` and the Raspberry Pi Pico 2 will be mounted at `/run/media/${USER}/RP2350`.

On macOS system, the Raspberry Pi Pico will be mounted at `/Volumes/RPI-RP2` and the Raspberry Pi Pico 2 will be mounted at `/Volumes/RP2350`.

For example:

Power on Raspberry Pico with BOOTSEL button pressed

```
$ ls -l /Volumes/RPI-RP2
total 16
-rwxrwxrwx  1 joe  staff  241 Sep  5  2008 INDEX.HTM*
-rwxrwxrwx  1 joe  staff   62 Sep  5  2008 INFO_UF2.TXT*

$ cp ~/Downloads/AtomVM-pico-v0.7.0.uf2 /Volumes/RPI-RP2/.
```

… at this point, the device will auto-unmount.

And again for the AtomVM core library (note that previously flashed `.uf2` files have disappeared):

Power on Raspberry Pico with BOOTSEL button pressed

```
$ ls -l /Volumes/RPI-RP2
total 16
-rwxrwxrwx  1 joe  staff  241 Sep  5  2008 INDEX.HTM*
-rwxrwxrwx  1 joe  staff   62 Sep  5  2008 INFO_UF2.TXT*

$ cp ~/Downloads/atomvmlib-pico-v0.7.0.uf2 /Volumes/RPI-RP2/.
```

… and again, at this point, the device will auto-unmount.

3.3.3 Potential Issues with macOS

There are known issues copying files to the Pico using macOS, and a lot of literature online. Usually it's best to use the Terminal rather than the Finder because the errors are more explicit. Copying may also fail with UF2 files downloaded from the Internet, typically AtomVM release binaries.

```
$ cp ~/Downloads/AtomVM-pico_w-v0.6.0.uf2 /Volumes/RPI-RP2/.
cp: /Volumes/RPI-RP2/AtomVM-pico-v0.6.0.uf2: fcopyfile failed: Operation not
 permitted
cp: /Users/joe/Downloads/AtomVM-pico-v0.6.0.uf2: could not copy extended
 attributes to
/Volumes/RPI-RP2/AtomVM-pico-v0.6.0.uf2: Operation not permitted
```

Two issues appear here: one is macOS tries to copy extended attributes and this fails (but this error is not a blocker), and the other is the "Operation not permitted" because the file is quarantined, having been downloaded from the web.

First issue can be solved with `cp -x` if you don't tolerate the error message and second with `xattr -d`.

```
$ xattr -d com.apple.quarantine ~/Downloads/AtomVM-pico_w-v0.6.0.uf2
$ cp -x ~/Downloads/AtomVM-pico_w-v0.6.0.uf2 /Volumes/RPI-RP2/.
$
```

3.3.4 Deploying an AtomVM application for Generic Unix

An AtomVM application is a collection of BEAM files, which have been compiled using the Erlang or Elixir compiler. These BEAM files are assembled into an AtomVM "packbeam" (`.avm`) file, which in turn can be provided to the `atomvm` executable on the command line.

When the AtomVM virtual machine starts, it will search for the first module that contains an exported `start/0` function in this partition, and it will begin execution of the BEAM bytecode at that function.

AtomVM applications can be written in Erlang or Elixir, or a combination of both. The AtomVM community has provided tooling for both platforms, making deployment of AtomVM applications as seamless as possible.

For information about how to flash your application to your Raspberry Pi Pico, see the AtomVM Tooling chapter.

3.4 Getting Started on the Generic UNIX platform

The AtomVM virtual machine is supported a wide variety of Generic UNIX platforms, including many Linux kernels and target architectures, FreeBSD, and MacOS, allowing users to write Erlang and Elixir programs and run them on a local development machine. For specific information about which Generic UNIX versions and architectures are supported, please refer to the AtomVM Release Notes.

These instructions cover how to provision the AtomVM virtual machine onto your development machine. Running applications locally can sometimes be a useful exercise in debugging.

Caution! Not all programming interfaces are supported on all platforms. See the AtomVM

Programmers Guide for more information.

For most applications, you should only need to install the VM once (or at least once per desired AtomVM release). Once the VM is installed, you can then begin development of Erlang or Elixir applications, which can then be flashed as part of your routine development cycle.

3.4.1 Generic UNIX Requirements

Deployment of AtomVM on the Generic UNIX platform requires the following components:

- A computer running MacOS or Linux (Windows support is not currently supported);
- An Erlang/OTP and compatible Elixir runtime;
- (recommended) For Erlang programs, `rebar3`;
- (recommended) For Elixir programs, `mix`, which ships with the Elixir runtime;

For information about specific versions of required software, see the AtomVM Release Notes.

3.4.2 Installing the AtomVM virtual machine

The following methods can be used to install the AtomVM virtual machine on the Generic UNIX platform:

1. Download Linux Binaries
2. (MacOS only) Installing via macports or Homebrew;
3. Building from source.

Installation on Linux Platforms

Downloading a pre-built binary image for Linux is by far the easiest path to getting started with development on a Linux development machine. Binary images contain the virtual machine.

Download the latest release image for Linux.

This image will generally take the form:

```
Atomvm-linux-<arch>-<atomvm-version>
```

where `<arch>` is the target architecture.

For example:

```
Atomvm-linux-x86_64-v0.6.0
```

You will also find the sha256 hash for this file, which you should verify using the `sha256sum` command on your local operating system.

You will also need a copy of the AtomVM core libraries, which include all of the compiled Erlang and Elixir needed to run parts of the VM.

This library will generally take the form:

```
atomvmlib-<atomvm-version>.avm
```

For example:

```
atomvmlib-v0.6.0.avm
```

You will also find the sha256 hash for this file, which you should verify using the `sha256sum` command on your local operating system.

See also

See below for instructions about how to run the `AtomVM` binary, together with the AtomVM core libraries on the command line.

Installation on MacOS

You can install AtomVM for Generic UNIX using macports or Homebrew. This instructions assume you are familiar with these package managers.

To install via macports:

```
$ sudo port install atomvm
```

Once installed, the `atomvm` executable should be available in your `$PATH` environment variable.

```
$ which atomvm
/opt/local/bin/atomvm
```

To install via Homebrew, you will first need to install the `atomvm` Homebrew Tap:

```
$ brew tap atomvm/atomvm
```

This command will make the `atomvm` Homebrew formula available to you.

```
$ brew install atomvm
```

Once installed, the `atomvm` executable should be available in your `$PATH` environment variable.

```
$ which atomvm
/usr/local/bin/atomvm
```

Building on MacOS from source

You may optionally build AtomVM from source and install the AtomVM virtual machine to your development machine. Building AtomVM from source is slightly more involved, as it requires the installation of third party libraries and is typically recommended only for users who are doing development on the AtomVM virtual machine, or for developers implementing custom Nifs or ports.

Instructions for building AtomVM from source are covered in the AtomVM Build Instructions.

3.4.3 Running applications on the Generic UNIX platform

AtomVM may be run on UNIX-like platforms using the `atomvm` command.

You may specify one or more AVM files on the command line when running the `atomvm` command. BEAM modules defined in earlier AVM modules on the command line take higher precedence that BEAM modules included in AVM files later in the argument list.

```
$ atomvm /path/to/myapp.avm
```

To get the current version of AtomVM, use the `-v` option, e.g.:

```
$ atomvm -v
0.6.0
```

Use the `-h` option to get command line help:

```
$ atomvm -h

Syntax:

    /usr/local/lib/atomvm/AtomVM [-h] [-v] <path-to-avm-file>+

Options:

    -h          Print this help and exit.
    -v          Print the AtomVM version and exit.
```

```
Supply one or more AtomVM packbeam (.avm) files to start your application.

Example:

    $ /usr/local/lib/atomvm/AtomVM /path/to/my/application.avm /path/to
/atomvmlib.avm
```

3.5 Getting Started with AtomVM WebAssembly

You can run AtomVM for WebAssembly with NodeJS or within common browsers (Safari, Chrome and Chrome-based, Firefox).

3.5.1 Getting Started with AtomVM WebAssembly port for NodeJS

Download the latest release image for Node.

This image will generally take the form:

    Atomvm-node-<atomvm-version>.js

For example:

    Atomvm-node-v0.6.0.js

You will also find the sha256 hash for this file, which you should verify using the `sha256sum` command on your local operating system.

AtomVM's WebAssembly port for NodeJS may be run using `node` command and AtomVM.js, AtomVM.worker.js and AtomVM.wasm files.

```
$ node /path/to/Atomvm-node-v0.6.0.js /path/to/myapp.avm
```

3.5.2 Getting Started with AtomVM WebAssembly port for browsers

AtomVM may also be run in modern browsers (Safari, Chrome and Chrome-based, Firefox) using AtomVM.js, AtomVM.worker.js and AtomVM.wasm files.

Please note that these files are different from the NodeJS ones.

Because AtomVM uses SharedArrayBuffer, to be executed by a browser, these files need to be served:

- on localhost or over HTTPS
- by a web server that also sends `Cross-Origin-Opener-Policy` and `Cross-Origin-Embedder-Policy` headers. These headers are also called COOP and COEP headers.

These security requirements are documented in Mozilla's documentation.

Trying locally from AtomVM source tree

If you compile AtomVM for Unix as well as for Node as explained in the build instructions, you can use an AtomVM-based toy webserver to serve the WebAssembly examples with:

```
$ ./src/AtomVM examples/emscripten/wasm_webserver.avm
```

This web server serves HTML files from `examples/emscripten/`. It works without HTTPS because files are served on localhost.

Using a hosting service with a `_headers` file

You can also host the three files on a hosting service such as Netlify that uses `_headers` files.

The file could have the following content:

```
/*
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

Using web server such as Nginx

You can also host the three files on web server such as Nginx or Apache.

The configuration for Nginx would be:

```
server {
    add_header Cross-Origin-Opener-Policy "same-origin";
    add_header Cross-Origin-Embedder-Policy "require-corp";
    location / {
        ...
    }
}
```

Using Javascript service worker trick

If you have no possibility to modify the headers, for example with GitHub pages, you can still get AtomVM to run in the browser using a Javascript service worker trick.

We did successfully use coi-serviceworker.

3.6 Where to go from here

The following resources may be useful for understanding how to develop Erlang or Elixir applications for the AtomVM platform:

- AtomVM Tooling
- Example Programs
- Programmers Guide

4 AtomVM Tooling

AtomVM provides an implementation of the Erlang virtual machine, and as such it provides runtime support for applications targeted for the platform.

However, developers will typically make use of downstream tooling that simplifies the development and provisioning of applications onto devices that are running the on the virtual machine.

This chapter presents an overview of these tools and how they can be used to make you more productive as an AtomVM developer.

Two tools are supported, one for Erlang developers, and one for Elixir developers:

- For Erlang developers: `atomvm_rebar3_plugin`
- For Elixir developers: `ExAtomVM`

4.1 `atomvm_rebar3_plugin`

The `atomvm_rebar3_plugin` is a rebar3 plugin that can be used to create and flash Erlang applications that run over AtomVM. Using this plugin greatly simplifies the process of building Erlang applications that run over AtomVM, and is strongly encouraged for all users.

4.1.1 Prerequisites for `atomvm_rebar3_plugin`

To use the `atomvm_rebar3_plugin`, you will need the following software on your development machine:

- A suitable version of the Erlang/OTP distribution. See the Release Notes for information about supported Erlang/OTP versions.
- A recent version of the rebar3 command-line tool.
- (optional) The git command line tool, to follow examples in this chapter.
- For flashing to ESP32, the `esptool` program.
- For flashing to STM32, `st-flash` via stlink
- (optional) A serial console program such as minicom or screen, to view console output from a device.
- (recommended) For rp2040 and rp2350, `picotool` for software resets on Raspberry Pi Pico. (optionally used if found in PATH to disconnect active `screen` sessions, which normally prevent flashing)

4.1.2 Erlang Example Program

To see this plugin in action, we will clone the `atomvm_examples` Github repository, and build and run the most simple

```
$ git clone https://github.com/atomvm/atomvm_examples
...
$ cd atomvm_examples/erlang/hello_world
```

From this directory we will run various rebar3 targets in the steps below.

4.1.3 Creating an AtomVM AVM file with `rebar3`

To create an AtomVM packbeam file (ending in `.avm`), use the `packbeam` target in the `atomvm` namespace:

```
$ rebar3 atomvm packbeam
...
===> AVM file written to .../hello_world/_build/default/lib/hello_world.avm
```

See also

See the atomvm_rebar3_plugin page for more detailed instructions about how to use the packbeam target.

4.1.4 Running applications on `generic_unix`

If you have installed AtomVM on a generic UNIX platform, you and run the example program directly using the atomvm command:

```
$ atomvm _build/default/lib/hello_world.avm
Hello World
Return value: ok
```

For instructions about how to install AtomVM on the generic_unix platform, see the Getting Started Guide

4.1.5 Flashing your application with `rebar3`

The atomvm_rebar3_plugin supports flash targets for various device types. These targets are described in more detail below.

ESP32

To flash AtomVM AVM file to an ESP32 device, use the esp32_flash target in the atomvm namespace. Users will typically specify the device port and baud rate as command-line options to this target.

Important In order to use the esp32_flash target, you will need to install the esptool program.

For example:

```
$ rebar3 atomvm esp32_flash --port /dev/ttyUSB0 --baud 921600
...
===> esptool.py --chip auto --port /dev/ttyUSB0 --baud 921600 --before
 default_reset
    --after hard_reset write_flash -u --flash_mode keep --flash_freq keep
 --flash_size detect
    0x210000 atomvm_examples/erlang/hello_world/_build/default/lib/hello_world.avm
```

Tip A baud rate of 921600 works well for most ESP32 devices, some can work reliably at higher rates of 1500000, or even 2000000, but some devices (especially those with a 26Mhz crystal frequency, rather than the more common 40 Mhz crystal) may need to use a slower baud rate such as 115200.

> Note. A baud rate of 921600 works well for most ESP32 devices, some can work reliably at higher rates of 1500000, or even 2000000, but some devices (especially those with a 26Mhz crystal frequency, rather than the more common 40 Mhz crystal) may need to use a slower baud rate such as 115200.

See the atomvm_rebar3_plugin page for more detailed instructions about how to use the esp32_flash target.

You can now use a serial console program such as minicom or screen to view console output from a device.

```
    ###########################################################
```

```


       ###    ######### #######  ##     ## ##     ## ##     ##
      ## ##      ##    ##     ## ###   ### ##     ## ###   ###
     ##   ##     ##    ##     ## #### #### ##     ## #### ####
    ##     ##    ##    ##     ## ## ### ## ##     ## ## ### ##
    #########    ##    ##     ## ##     ##  ##   ##  ##     ##
    ##     ##    ##    ##     ## ##     ##   ## ##   ##     ##
    ##     ##    ##     #######  ##     ##    ###    ##     ##

    ###########################################################

I (852) AtomVM: Starting AtomVM revision 0.6.0-alpha.1
I (862) sys: Loaded BEAM partition boot.avm at address 0x1d0000 (size=262144
 bytes)
I (882) network_driver: Initialized network interface
I (882) network_driver: Created default event loop
I (902) AtomVM: Found startup beam esp32init.beam
I (922) AtomVM: Starting esp32init.beam...
---
AtomVM init.
I (932) sys: Loaded BEAM partition main.avm at address 0x210000 (size=1048576
bytes)
Starting application...
Hello World
AtomVM finished with return value: ok
I (972) AtomVM: AtomVM application terminated.  Going to sleep forever ...
```

STM32

To flash AtomVM AVM file to an STM32 device, use the `stm32_flash` target in the `atomvm` namespace.

Important In order to use the `stm32_flash` target, you will need to install the `st-flash` tool from the open source (bsd-3 licensed)stlink suite of stm32 utilities.

Since the AtomVM core libraries are not flashed to an STM32 device, you will need to include is library in your application. As part of the build process for the STM32, you will have built the AtomVM core libraries into a file named `atomvmlib.avm`

!DANGER! It is critical that the version of the AtomVM core libraries match the version of the AtomVM virtual machine you built as part of the STM32 build. Be sure to use the version of this library (written to `build/lib/atomvmlib.avm` during the build process). For more information about how to build AtomVM for the STM32 platform, see the AtomVM Build Instructions.

In general, it is also a good idea to use the prune option when creating your application's AVM file. This way, only the modules that are needed for your application will be included, which will decrease the size of your application's AVM file, leading to faster development times.

Edit the `rebar.config` so that it includes the following `atomvm_rebar3_plugin` stanza, if it does not already.

```
{atomvm_rebar3_plugin, [
    {packbeam, [prune]}
]}.
```

This stanza will guarantee that the generated packbeam file will be pruned when created.

You will need to first build a packbeam file that includes the AtomVM core libraries. Use the `packbeam` task in the `atomvm` namespace, and specify the path to the `atomvmlib.avm` file you created as part of the build.

```
$ rebar3 atomvm packbeam -e /path/to/atomvmlib.avm
```

You may now flash your application to your STM32 device:

```
$ rebar3 atomvm stm32_flash
...
===> st-flash --reset write _build/default/lib/hello_world.avm 0x8080000
```

For devices with only 512KB of flash the application address is different and must be specified:

```
$ rebar3 atomvm stm32_flash -o 0x8060000
...
===> st-flash --reset write _build/default/lib/hello_world.avm 0x8060000
```

See the atomvm_rebar3_plugin page for more detailed instructions about how to use the `stm32_flash` target.

You can now use a serial console program such as minicom or screen to view console output from a device.

```
    ###########################################################

       ###    ########  #######  ##     ## ##     ## ##     ##
      ## ##      ##    ##     ## ###   ### ##     ## ###   ###
     ##   ##     ##    ##     ## #### #### ##     ## #### ####
    ##     ##    ##    ##     ## ## ### ## ##     ## ## ### ##
    #########    ##    ##     ## ##     ##  ##   ##  ##     ##
    ##     ##    ##    ##     ## ##     ##   ## ##   ##     ##
    ##     ##    ##     #######  ##     ##    ###    ##     ##

    ###########################################################

INFO [51] AtomVM: Starting AtomVM revision 0.6.0-alpha.2+git.59e25c34
INFO [58] AtomVM: Booting file mapped at: 0x8080000, size: 444
INFO [64] AtomVM: Starting: hello_world.beam...

---
Hello World
INFO [74] AtomVM: Exited with return: ok
INFO [78] AtomVM: AtomVM application terminated.  Going to sleep forever ...
```

Raspberry Pi RP2

To generate a Raspberry Pi RP2 uf2 file from an AtomVM AVM file and flash it to an RP2 device, use the `pico_flash` target in the `atomvm` namespace.

For example:

```
$ rebar3 atomvm pico_flash
...
===> AVM file written to _build/default/lib/hello_world.avm
===> Resetting device at path /dev/ttyACM0
===> Waiting for the device at path /run/media/${USER}/RPI-RP2 to settle and
 mount...
===> Copying atomvm_examples/erlang/hello_world/_build/default/lib/hello.uf2 to
     /run/media/${USER}/RPI-RP2...
```

See the atomvm_rebar3_plugin page for more detailed instructions about how to use the `pico_flash` target. By default it will generate a universal UF2 that works on both RP2040 and RP2350 but you can optionally target only one of socs.

You can now use a serial console program such as minicom or screen to view console output from a device. The default build will wait 20 seconds for a serial connection to be established before starting the application.

```
    ###########################################################

       ###    ########  #######  ##     ## ##     ## ##     ##
      ## ##      ##    ##     ## ###   ### ##     ## ###   ###
     ##   ##     ##    ##     ## #### #### ##     ## #### ####
    ##     ##    ##    ##     ## ## ### ## ##     ## ## ### ##
    #########    ##    ##     ## ##     ##  ##   ##  ##     ##
    ##     ##    ##    ##     ## ##     ##   ## ##   ##     ##
    ##     ##    ##     #######  ##     ##    ###    ##     ##

    ###########################################################

Starting AtomVM revision 0.6.0-alpha.2+git.59e25c34
Found startup beam hello_world.beam
Starting hello_world.beam...
---
Hello World
AtomVM finished with return value: ok
AtomVM application terminated.  Going to sleep forever ...
```

If no connection is made before the timeout is reached the application will start, but the uart console will not be available. At this point you can use `picotool` to reboot the device into `application` mode.

Example:

```
$ picotool reboot -f
The device was asked to reboot into application mode.

$
```

This will again give you 20 seconds to establish a serial monitor connection. For information about changing this timeout, or locking down the device so that software resets no longer work (requiring that the device be power cycled and the `BOOTSEL` button help when powering on to flash) consult the rp2040 section of the Build Instructions.

4.2 `ExAtomVM`

The `ExAtomVM` tool is a mix plugin that can be used to create and flash Elixir applications that run over AtomVM. Using this plugin greatly simplifies the process of building Elixir applications that run over AtomVM, and is strongly encouraged for new users.

4.2.1 Prerequisites for `ExAtomVM`

To use the `ExAtomVM` tool, you will need the following software on your development machine:

- A suitable version of the Erlang/OTP distribution. See the Release Notes for information about supported Erlang/OTP versions.
- A suitable version of the Elixir distribution. See the Release Notes for information about supported Elixir versions.
- (optional) The git command line tool, to follow examples in this chapter.
- For flashing to ESP32, the `esptool` program.
- (optional) A serial console program such as minicom or screen, to view console output from a device.

4.2.2 Elixir Example Program

To see this plugin in action, we will clone the `atomvm_examples` Github repository, and build and run the most simple

```
$ git clone https://github.com/atomvm/atomvm_examples
...
$ cd atomvm_examples/elixir/HelloWorld
```

From this directory we will run various `mix` targets in the steps below.

4.2.3 Creating an AtomVM AVM file with `mix`

To create an AtomVM packbeam file (ending in `.avm`), first use the `mix.deps` target to `mix` in order to download any dependencies:

```
$ mix deps.get
* Updating exatomvm (https://github.com/atomvm/ExAtomVM/)
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 17 (delta 6), reused 16 (delta 6), pack-reused 0
origin/HEAD set to main
```

You can now use the `mix atomvm.packbeam` task to create a packbeam (ending in `.avm`) file:

```
$ mix atomvm.packbeam
==> exatomvm
Compiling 5 files (.ex)
Generated exatomvm app
==> HelloWorld
Compiling 1 file (.ex)
Generated HelloWorld app
No avm_deps directory found.
This message can be safely ignored when standard libraries are already flashed to
 lib
partition.
```

The `HelloWorld.avm` is located in the top level directory:

```
$ ls -l HelloWorld.avm
-rw-rw-r--  1 user  wheel  19120 Oct 13 14:06 HelloWorld.avm
```

See also

See the `ExAtomVM` page for more detailed instructions about how to use the `mix atomvm.packbeam` task.

4.2.4 Running on the `generic_unix` platform

If you have installed AtomVM on a generic UNIX platform, you and run the example program directly using the `atomvm` command:

```
$ atomvm HelloWorld.avm
Hello World
Return value: ok
```

For instructions about how to install AtomVM on the `generic_unix` platform, see the Getting Started Guide

4.2.5 Flashing your application with `mix`

The `ExAtomVM` plugin supports flash targets for various device types. These targets are described in more detail below.

ESP32 flash task

To flash AtomVM packbeam file to an ESP32 device, use the `mix atomvm.esp32.flash` task. Users will typically specify the device port and baud rate as command-line options to this target.

Important In order to use the `mix atomvm.esp32.flash` task, you will need to install the `esptool` program.

For example:

```
$ mix atomvm.esp32.flash --port /dev/ttyUSB0 --baud 921600
```

Tip A baud rate of 921600 works well for most ESP32 devices, some can work reliably at higher rates of 1500000, or even 2000000, but some devices (especially those with a 26Mhz crystal frequency, rather than the more common 40 Mhz crystal) may need to use a slower baud rate such as 115200.

See the `ExAtomVM` page for more detailed instructions about how to use the `mix atomvm.esp32.flash` task.

You can now use a serial console program such as minicom or screen to view console output from a device.

```
    ###########################################################

       ###    ########  #######  ##     ## ##     ## ##     ##
      ## ##      ##    ##     ## ###   ### ##     ## ###   ###
     ##   ##     ##    ##     ## #### #### ##     ## #### ####
    ##     ##    ##    ##     ## ## ### ## ##     ## ## ### ##
    #########    ##    ##     ## ##     ##  ##   ##  ##     ##
    ##     ##    ##    ##     ## ##     ##   ## ##   ##     ##
    ##     ##    ##     #######  ##     ##    ###    ##     ##

    ###########################################################

I (852) AtomVM: Starting AtomVM revision 0.6.0-alpha.1
I (862) sys: Loaded BEAM partition boot.avm at address 0x1d0000 (size=262144
 bytes)
I (882) network_driver: Initialized network interface
I (882) network_driver: Created default event loop
I (902) AtomVM: Found startup beam esp32init.beam
W (902) sys: AVM partition not found for lib.avm
I (902) AtomVM: Unable to mount lib.avm partition. Hopefully the AtomVM core
 libraries
are included in your application.
I (922) AtomVM: Starting esp32init.beam...
---
AtomVM init.
I (932) sys: Loaded BEAM partition main.avm at address 0x250000 (size=1048576
 bytes)
Starting application...
Hello World
AtomVM finished with return value: ok
I (972) AtomVM: AtomVM application terminated.  Going to sleep forever ...
```

STM32 flash task

To flash AtomVM packbeam file to an STM32 device, use the `mix atomvm.stm32.flash` task.

Important In order to use the `mix atomvm.stm32.flash` task, you will need to install the `st-flash` tool from the open source (bsd-3 licensed) stlink suite of stm32 utilities.

For example:

```
$ mix atomvm.stm32.flash
```

Most devices do not need to enter the default application offset `0x8080000`, but devices with only 512KiB of flash storage need to use `--flash_offset=0x8060000` parameter setting to upload the application to the correct flash location.

BlackPill V2 example:

```
$ mix atomvm.stm32.flash --flash_offset=0x8060000
```

If the `st-flash` tool is not in environment PATH, the full path to the `st-flash` tool should be exported to the environment variable `ATOMVM_MIX_PLUGIN_STFLASH`, for example:

```
$ export ATOMVM_MIX_PLUGIN_STFLASH=/opt/stlink/bin/st-flash
```

4.3 `atomvm_packbeam`

The atomvm_packbeam tool is a simple command-line utility that allows you to create, inspect, and manipulate AtomVM PackBEAM files. By convention, PackBEAM files end in the `.avm` suffix and are referred to as "AVM" files, in the remainder of this section.

Tip Users generally do not have a need to use the `packbeam` tool directly. Instead, the functionality of this tool is embedded in the `atomvm_rebar3_plugin`.

4.3.1 Installation

Consult the atomvm_packbeam Github page for instructions about how to install the `atomvm_packbeam` utility. Once installed, you should have the `packbeam` command line tool available in your `PATH`.

4.3.2 Usage

The `packbeam` command supports the following sub-commands:

- `create` Create an AVM file from a collection of files.
- `list` List the contents of an AVM file.
- `extract` Extract elements from an AVM file.
- `delete` Delete elements from an AVM file.

These sub-commands are described in more detail below.

See also

These notes provide only a high-level view of this `packbeam` utility. For more detailed information, see the atomvm_packbeam Github repository.

Creating AVM files

To create an AVM from a list of existing files (typically `.beam` files), use the `create` subcommand. Specify the output `.avm` first, followed by a list of files you would like to include in the output file. E.g.,

```
$ packbeam create output.avm foo.beam bar.beam
```

Tip Creation of AVM files is more typically done via the atomvm_rebar3_plugin; however, the `packbeam` command can be used to inspect and/or manipulate AVM files after they have been created by this plugin. This isn't typically required, but in some instances it can be useful.

Note that you can supply a previously created AVM file as an input to another creation, which will result in including all the files in the source AVM file in the destination.

```
$ packbeam create new_output.avm tapas.beam output.avm
```

You can also embed non-BEAM files in an AVM file. These files are accessible programmatically withing atomvm via the `atomvm:read_priv/2` function, described in the AtomVM Programmer's Guide.

For example, if you wanted to add a file `my_app/priv/my_file.txt` to a new file, you could use the following command:

```
$ packbeam create my_app.beam my_app/priv/my_file.txt my_lib.avm
```

Important There are conventions for embedding non-BEAM files in AVM files that need to be followed in order to be able to load these files programmatically within AtomVM. Generally, these files must obey the path `<module-name>/priv/<path-to-file>`, where `<module-name>` is the name of a module, and `<path-to-file>` is a path to the embedded file. (This path may include embedded / separators). Example: `my_app/priv/bubbles/sample.txt`

Start Flags

An AtomVM application must contain a start entrypoint, i.e., a module that exports the `start/0` function. You can specify the name of this module via the `--start` flag. E.g.,

```
$ packbeam create --start main my_app.avm foo.beam bar.beam main.beam
```

Use of this flag will ensure that the `main.beam` module will be found first in the search order when the AtomVM virtual machine starts your application.

Pruning

Pruning an AVM file is a useful mechanism for making your AVM files smaller, and thus faster to flash and including less data than necessary. You can prune an AVM using the

```
$ packbeam create --start main --prune my_app.beam foo.beam bar.beam main.beam
 a.beam \
b.beam c.beam
```

Any BEAM files that contain no transitive references from the start module are removed from the output AVM file, making them smaller and less bloated.

Important You can only use the `--prune` option if you specify a `--start` module.

Listing AVM file contents

You can list the contents of an AVM file via the `list` sub-command.

```
$ packbeam list myapp.avm
myapp.beam * [384]
myapp/priv/application.bin [220]
```

Any BEAM files with an exported `start/0` function are listed with an asterisk (`*`). In general, if you want your application to start from a designated entrypoint, that BEAM file should occur first in the list.

The size (in bytes) of the entries are listed in square brackets (`[]`).

Extracting AVM file contents

You can extract elements of an AVM file, writing them to the file system, using the `extract` sub-command.

Specify the directory location into which you would like to extract the files using the `-out` flag, followed by the path to the input AVM file, and a list of paths from the input AVM you would like to extract.

```
$ mkdir mydir
$ packbeam extract -out mydir myapp.avm myapp/priv/application.bin
Writing to mydir ...
x myapp/priv/application.bin
```

Deleting AVM file contents

You can delete elements of an AVM file using the `delete` sub-command.

Specify the AVM file you would like to write as output (which can be the same as the input AVM file) using the `-out` flag, followed by the path to the input AVM file, and a list of paths from the input AVM you would like to delete.

```
$ packbeam delete -out myapp2.avm myapp.avm myapp/priv/application.bin

shell$ packbeam list myapp2.avm
myapp.beam * [384]
```

4.3.3 Help

To get help about `packbeam` syntax, use the `help` subcommand:

```
$ packbeam help

packbeam version 0.7.0

Syntax:
    packbeam <sub-command> <options> <args>

The following sub-commands are supported:

    create <options> <output-avm-file> [<input-file>]+
        where:
        <output-avm-file> is the output AVM file,
        [<input-file>]+ is a list of one or more input files,
        and <options> are among the following:
            [--prune|-p]           Prune dependencies
            [--start|-s <module>]  Start module
            [--remove_lines|-r]    Remove line number information from AVM files

    list <options> <avm-file>
        where:
        <avm-file> is an AVM file,
        and <options> are among the following:
            [--format|-f csv|bare|default]  Format output

    extract <options> <avm-file> [<element>]*
        where:
        <avm-file> is an AVM file,
        [<element>]+ is a list of one or more elements to extract
            (if empty, then extract all elements)
        and <options> are among the following:
            [--out|-o <output-directory>]   Output directory into which to write
 elements
```

```
            (if unspecified, use the current working directory)

    delete <options> <avm-file> [<element>]+
        where:
        <avm-file> is an AVM file,
        [<element>]+ is a list of one or more elements to delete,
        and <options> are among the following:
            [--out|-o <output-avm-file>]    Output AVM file

    version
        Print version and exit

    help
        Print this help
```

See also

For more detailed information about the `atomvm_packbeam` utility, see the atomvm_packbeam documentation Github page.

4.4 Where to go from here

With knowledge of AtomVM tooling, you can more easily follow the AtomVM Example Programs, or read the Programmers Guide and start writing your own applications.

Chapter 5

Programmers Guide

This guide is intended for programmers who develop applications targeted for AtomVM.

As an implementation of the Erlang virtual machine, AtomVM is designed to execute unmodified byte-code instructions compiled into BEAM files, either by the Erlang or Elixir compilers. This allow developers to write programs in their BEAM programming language of choice, and to use the common Erlang community tool-chains specific to their language platform, and to then deploy those applications onto the various devices that AtomVM supports.

This document describes the development workflow when writing AtomVM applications, as well as a high-level overview of the various APIs that are supported by AtomVM. With an understanding of this guide, you should be able to design, implement, and deploy applications onto a device running the AtomVM virtual machine.

5.1 AtomVM Features

Currently, AtomVM implements a strict subset of the BEAM instruction set.

A high level overview of the supported language features include:

- All the major Erlang types, including
  - integers (with size limits)
  - floats
  - tuples
  - lists
  - binaries
  - maps
- support for many Erlang BIFs and guard expressions to support the above types
- pattern matching (case statements, function clause heads, etc)
- `try ... catch ... finally` constructs
- anonymous functions
- process `spawn` and `spawn_link`
- send (`!`) and `receive` messages
- bit syntax (with some restrictions)
- reference counted binaries
- stacktraces
- symmetric multi-processing (SMP)

In addition, several features are supported specifically for integration with micro-controllers, including:

- Wifi networking (`network`)
- UDP and TCP/IP support (`inet`, `gen_tcp` and `gen_udp`)
- Peripheral and system support on micro-controllers, including
  - GPIO, including pins reads, writes, and interrupts
  - I2C interface
  - SPI interface
  - UART interface
  - LEDC (PWM)
  - non-volatile storage (NVS)
  - RTC storage
  - deep sleep

5.1.1 Limitations

While the list of supported features is long and growing, the currently unsupported Erlang/OTP and BEAM features include (but are not limited to):

- Atoms. Atoms larger than 255 bytes (255 ascii characters) are not supported.
- Bignums. Integer values are restricted to 64-bit values.
- Bit Syntax. While packing and unpacking of arbitrary (but less than 64-bit) bit values is supported, packing and unpacking of integer values at the start or end of a binary, or bordering binary packing or extraction must align on 8-bit boundaries. Arbitrary bit length binaries are not currently supported.
- The `epmd` and the `disterl` protocols are not supported.
- There is no support for code hot swapping.
- There is no support for a Read-Eval-Print-Loop. (REPL)
- Numerous modules and functions from Erlang/OTP standard libraries (`kernel`, `stdlib`, `sasl`, etc) are not implemented.

AtomVM bit syntax is restricted to alignment on 8-bit boundaries. Little-endian and signed insertion and extraction of integer values is restricted to 8, 16, and 32-bit values. Only unsigned big and little endian 64-bit values can be inserted into or extracted from binaries.

It is highly unlikely that an existing Erlang program targeted for Erlang/OTP will run unmodified on AtomVM. And indeed, even as AtomVM matures and additional features are added, it is more likely than not that Erlang applications will need to targeted specifically for the AtomVM platform. The intended target environment (small, cheap micro-controllers) differs enough from desktop or server-class systems in both scale and APIs that special care and attention is needed to target applications for such embedded environments.

That being said, many of the features of the BEAM are supported and provide a rich and compelling development environment for embedded devices, which Erlang and Elixir developers will find natural and productive.

5.2 AtomVM Development

This section describes the typical development environment and workflow most AtomVM developers are most likely to use.

5.2.1 Development Environment

In general, for most development purposes, you should be able to get away with an Erlang/OTP development environment, and for Elixir developers, and Elixir development environment. For specific version requirements, see the Release Notes.

We assume most development will take place on some UNIX-like environment (e.g., Linux, FreeBSD, or MacOS). Consult your local package manager for installation of these development environments.

Developers will want to make use of common Erlang or Elixir development tools, such as `rebar3` for Erlang developers or `mix` for Elixir developers.

Developers will need to make use of some AtomVM tooling. Fortunately, there are several choices for developers to use:

1. AtomVM `PackBEAM` executable (described below)
2. `atomvm_rebar3_plugin`, for Erlang development using `rebar3`.
3. `ExAtomVM` Mix plugin, Elixir development using `Mix`.

Some testing can be performed on UNIX-like systems, using the `AtomVM` executable that is suitable for your development environment. AtomVM applications that do not make use of platform-specific APIs are suitable for such tests.

Deployment and testing on micro-controllers is slightly more involved, as these platforms require additional hardware and software, described below.

ESP32 Deployment Requirements

In order to deploy AtomVM applications to and test on the ESP32 platform, developers will need:

- A computer running MacOS or Linux (Windows support is TBD);
- An ESP32 module with a USB/UART connector (typically part of an ESP32 development board);
- A USB cable capable of connecting the ESP32 module or board to your development machine (laptop or PC);
- The `esptool` program, for flashing the AtomVM image and AtomVM programs;
- (Optional, but recommended) A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.

STM32 Deployment Requirements

- A computer running MacOS or Linux (Windows is not currently supported);
- An stm32 board with a USB/UART connector (these are built into some boards such as the Nucleo product line) and a minimum of 512k (1M recommended) of flash and a minimum of 100k RAM;
- A USB cable capable of connecting the STM32 board or external UART connector to your development machine (laptop or PC);
- `st-flash` via stlink, to flash both AtomVM and your packed AVM applications. Make sure to follow its installation procedure before proceeding further.
- `packbeam` the AtomVM for packing and stripping `*.beam` files into the AtomVM `*.avm` format.
- (Optional, but recommended) A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.

Raspberry Pi RP2 Deployment Requirements

- A computer running MacOS or Linux (Windows support is not currently supported);

- A board with a Raspberry Pi RP2 soc such as Raspberry Pi Pico or Pico W or Pico 2, with a USB connector (typically part of a development board);
- A USB cable capable of connecting the module or board to your development machine (laptop or PC);
- (Optional, but recommended) A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.

5.2.2 Development Workflow

For the majority of users, AtomVM applications are written in the Erlang or Elixir programming language. These applications are compiled to BEAM (`.beam`) files using standard Erlang or Elixir compiler tool chains (`erlc`, `rebar3`, `mix`, etc). The generated BEAM files contain byte-code that can be executed by the Erlang/OTP runtime, or by the AtomVM virtual machine.

Note In a small number of cases, it may be useful to write parts of an application in the C programming language, as AtomVM nifs or ports. However, writing AtomVM nifs and ports is outside of the scope of this document.

Once Erlang and/or Elixir files are compiled to BEAM files, AtomVM provides tooling for processing and aggregating BEAM files into AtomVM Packbeam (`.avm`) files, using AtomVM tooling, distributed as part of AtomVM, or as provided through the AtomVM community.

AtomVM packbeam files are the applications and libraries that run on the AtomVM virtual machine. For micro-controller devices, they are "flashed" or uploaded to the device; for command-line use of AtomVM (e.g., on Linux, FreeBSD, or MacOS), they are supplied as the first parameter to the AtomVM command.

The following diagram illustrates the typical development workflow, starting from Erlang or Elixir source code, and resulting in a deployed Packbeam file:

```
*.erl or *.ex                  *.beam
+-------+                   +-------+
|       |+                  |       |+
|       ||+                 |       ||+
|       |||     -------->   |       |||
|       |||  Erlang/Elixir  |       |||
+-------+||     Compiler    +-------+||
 +-------+|                  +-------+|
  +-------+                   +-------+
     ^                              |
     |                              | packbeam
     |                              |
     |                              v
     |                            +-------+
     |                            |       |
     | test                       |       |
     | debug                      |       |
     | fix                        |       |
     |                            +-------+
     |                             app.avm
     |                              |
     |                              | flash/upload
     |                              |
     |                              v
     +-------------------- Micro-controller
                             device
```

The typical compile-test-debug cycle can be summarized in the following steps:

1. Deploy the AtomVM virtual machine to your device

2. Develop an AtomVM application in Erlang or Elixir
    1. Write application
    2. Deploy application to device
    3. Test/Debug/Fix application
    4. Repeat

Deployment of the AtomVM virtual machine and an AtomVM application currently require a USB serial connection. There is currently no support for over-the-air (OTA) updates.

For more information about deploying the AtomVM image and AtomVM applications to your device, see the Getting Started Guide

5.3 Applications

An AtomVM application is a collection of BEAM files, aggregated into an AtomVM "Packbeam" (`.avm`) file, and typically deployed (flashed) to some device. These BEAM files be be compiled from Erlang, Elixir, or any other language that targets the Erlang VM.

Attention! The return value from the `start/0` function is ignored on the the `generic_unix` platform, most MCU platforms have the option of rebooting the device if the `start/0` function returns a value other than `ok`. Consult the Build Instructions for your device to see how this is configured.

Here, for example is one of the smallest AtomVM applications you can write:

```
-module(myapp).

-export([start/0]).

start() ->
    ok.
```

This particular application doesn't do much, of course. The application will start and immediately terminate, with a return value of `ok`. Typical AtomVM applications will be more complex than this one, and the AVM file that contains the application BEAM files will be considerably larger and more complex than the above program.

Most applications will spawn processes, send and receive messages between processes, and wait for certain conditions to apply before terminating, if they terminate at all. For applications that spawn processes and run forever, you may need to add an empty `receive ... end` block, to prevent the AtomVM from terminating prematurely, e.g.,

```
wait_forever() ->
    receive X -> X end.
```

5.3.1 Packbeam files

AtomVM applications are packaged into Packbeam (`.avm`) files, which contain collections of files, typically BEAM (`.beam`) files that have been generated by the Erlang or Elixir compiler.

At least one BEAM module in this file must contain an exported `start/0` function. The first module in a Packbeam file that contain this function is the entry-point of your application and will be executed when the AtomVM virtual machine starts.

Not all files in a Packbeam need to be BEAM modules – you can embed any type of file in a Packbeam file, for consumption by your AtomVM application.

See also

The Packbeam format is described in more detail in the AtomVM PackBEAM format.

The AtomVM community has provided several tools for simplifying your experience, as a developer. These tools allow you to use standard Erlang and Elixir tooling (such as `rebar3` and `mix`) to build Packbeam files and deploy then to your device of choice.

5.3.2 packbeam tool

The `packbeam` tool is a command-line application that can be used to create Packbeam files from a collection of input files:

```
$ packbeam help
packbeam version 0.7.0
Syntax:
    packbeam <sub-command> <options> <args>

The following sub-commands are supported:

    create <options> <output-avm-file> [<input-file>]+
        where:
            <output-avm-file> is the output AVM file,
            [<input-file>]+ is a list of one or more input files,
            and <options> are among the following:
                [--prune|-p]           Prune dependencies
                [--start|-s <module>]  Start module
                [--remove_lines|-r]    Remove line number information from AVM
 files

    list <options> <avm-file>
        where:
            <avm-file> is an AVM file,
            and <options> are among the following:
                [--format|-f csv|bare|default]  Format output

    extract <options> <avm-file> [<element>]*
        where:
            <avm-file> is an AVM file,
            [<element>]+ is a list of one or more elements to extract
                (if empty, then extract all elements)
            and <options> are among the following:
                [--out|-o <output-directory>]   Output directory into which to
 write elements
                (if unspecified, use the current working directory)

    delete <options> <avm-file> [<element>]+
        where:
            <avm-file> is an AVM file,
            [<element>]+ is a list of one or more elements to delete,
            and <options> are among the following:
                [--out|-o <output-avm-file>]    Output AVM file

    version
        Print version and exit

    help
        Print this help
```

For more information consult the packbeam section of AtomVM Tooling.

5.3.3 Running `AtomVM`

AtomVM is executed in different ways, depending on the platform. On most microcontrollers (e.g., the ESP32), the VM starts when the device is powered on. On UNIX platforms, the VM is started from the command-line using the `AtomVM` executable.

AtomVM will use the first module in the supplied AVM file that exports a `start/0` function as the entrypoint for the application.

`AtomVM` program syntax

On UNIX platforms, you can specify a BEAM file or AVM file as the first argument to the executable, e.g.,

```
$ AtomVM foo.avm
```

Important If you start the `AtomVM` executable with a BEAM file, then the corresponding module may not make any calls to external function in other modules, with the exception of built-in functions and Nifs that are included in the VM.

5.4 Core APIs

The AtomVM virtual machine provides a set of Erlang built-in functions (BIFs) and native functions (NIFs), as well as a collection of Erlang and Elixir libraries that can be used from your applications.

This section provides an overview of these APIs. For more detailed information about specific APIs, please consult the API reference documentation.

5.4.1 Standard Libraries

AtomVM provides a limited implementations of standard library modules, including:

- `base64`
- `gen_server`
- `gen_statem`
- `io` and `io_lib`
- `lists`
- `maps`
- `proplists`
- `supervisor`
- `timer`

In addition AtomVM provides limited implementations of standard Elixir modules, including:

- `List`
- `Tuple`
- `Enum`
- `Kernel`
- `Module`
- `Process`
- `Console`

For detailed information about these functions, please consult the API reference documentation. These modules provide a strict subset of functionality from their Erlang/OTP counterparts. However, they aim to be API-compatible with the Erlang/OTP interfaces, at least for the subset of provided functionality.

5.4.2 Spawning Processes

AtomVM supports the actor concurrency model that is pioneered in the Erlang/OTP runtime. As such, users can spawn processes, send messages to and receive message from processes, and can link or monitor processes to be notified if they have crashed.

To spawn a process using a defined or anonymous function, pass the function to the `spawn/1` function:

```erlang
Pid = spawn(fun run_some_code/0),
```

The function you pass may admit closures, so for example you can pass variables defined outside of the scope of the function to the anonymous function to pass into `spawn/1`:

```erlang
Args = ...
Pid = spawn(fun() -> run_some_code_with_args(Args) end),
```

Alternatively, you can pass a module, function name, and list of arguments to the `spawn/3` function:

```erlang
Args = ...
Pid = spawn(?MODULE, run_some_code_with_args, [Args]),
```

The `spawn_opt/2,4` functions can be be used to spawn a function with additional options that control the behavior of the spawned processes, e.g.,

```erlang
Pid = spawn_opt(fun run_some_code/0, [{min_heap_size, 1342}]),
```

The options argument is a properties list containing optionally the following entries:

Key	Value Type	Default Value	Description
`min_heap_size`	`non_neg_integer()`	none	Minimum heap size of the process. The heap will shrink no smaller than this size.
`max_heap_size`	`non_neg_integer()`	unbounded	Maximum heap size of the process. The heap will grow no larger than this size.
`link`	`boolean()`	`false`	Whether to link the spawned process to the spawning process.
`monitor`	`boolean()`	`false`	Whether to link the spawning process should monitor the spawned process.
`atomvm_heap_growth`	`bounded_free | minimum | fibonacci`	`bounded_free`	Strategy to grow the heap of the process.

5.4.3 Console Output

There are several mechanisms for writing data to the console.

For common debugging, many users will find `erlang:display/1` sufficient for debugging:

```
erlang:display({foo, [{bar, tapas}]}).
```

The output parameter is any Erlang term, and a newline will be appended automatically.

Users may prefer using the `io:format/1,2` functions for more controlled output:

```
io:format("The ~p did a ~p~n", [friddle, frop]).
```

Tip The `io_lib` module can be used to format string data, as well.

5.4.4 Logging

AtomVM supports a subset of the OTP logging facility, allowing users to send log event to log handlers (by default, the console), and to install handlers that handle log events.

To log events, you are encouraged to use the logging macros from the OTP `kernel` application. You can use these macros at compile time, and the generated code can be run in AtomVM.

For example:

```
-include_lib("kernel/include/logger.hrl").
...
?LOG_NOTICE("Something happened that might require your attention: ~p",
 [TheThing])
```

By default, this will result in a message displayed on the console, with a timestamp, log level, PID of the process that initiated the log message, the module, function, and function arity, together with the supplied log message:

```
2023-07-04T18:34:56.387 [notice] <0.1.0> test_logger:test_default_logger/0
 Something
happened that might require your attention: ThatThingThatHappened
```

Tip Note that log messages need not (and generally should not) include newline separators (`~n`) in log format messages, unless necessary.

Users may provide a format string, with an optional list of arguments. Alternatively, users can provide a map encapsulating a "report" in lieu of a format string. Reports provide a mechanism for supplying a set of structured data directly to log handlers (see below), without necessarily incurring the cost of formatting log messages.

As with OTP, the following ordered log levels (from high to low) are supported:

- `emergency`
- `critical`
- `alert`
- `error`
- `warning`
- `notice`
- `info`
- `debug`

By default, the logging facility drops any messages below `notice` level. To set the default log level for the logging subsystem, see the `logger_manager` section, below.

You can use the `logger` interface directly to log messages at different levels, but in general, the OTP logging macros are encouraged, as log events generated using the OTP macros include additional metadata (such as the location of the log event) you do not otherwise get using the functions in the `logger` module.

For example, the expression

```
logger:notice("Something happened that might require your attention: ~p",
  [TheThing])
```

may seem similar to using the `?LOG_NOTICE` macro, but less contextual information will be included in the log event.

For more information about the OTP logging facility, see the Erlang/OTP Logging chapter.

Note AtomVM does not currently support programmatic configuration of the logging subsystem. All changes to default behavior should be done via the AtomVM `logger_manager` module (see below).

The `logger_manager`

In order to use the `logger` interface, you will need to first start the AtomVM `logger_manager` service.

Note Future versions of AtomVM may automatically start the logging subsystem as part of a kernel application, but currently, this service must be managed manually.

To start the `logger_manager`, use the `logger_manager:start_link/1` function, passing in a configuration map for the logging subsystem.

For example, the default logging framework can be started via:

```
{ok, _Pid} = logger_manager:start_link(#{})
```

Tip The logger_manager is a registered process, so the returned Pid may be ignored.

The configuration map supplied to the logger_manager may contain the following keys:

Key	Type	Default	Description
`log_level`	`log_level()`	`notice`	Primary log level
`logger`	`logger_config()`	`{handler, default, logger_std_h, #{}}`	Log configuration
`module_level`	`module_level()`	`undefined`	Log level specific to a set of modules

where `log_level()` is defined to be:

```
-type log_level() :: emergency | critical | alert | error | warning | notice |
  info | debug.
```

and `logger_config()` is defined as follows:

```
-type handler_id() :: default | atom().
-type handler_config() :: #{
    id => atom(),
    module => module(),
    level => logger:level() | all | none,
    config => term()
}.
```

```
-type logger_config() :: [
    {handler, default, undefined} |
    {
        handler,
        HandlerId :: handler_id(),
        Handler :: module(),
        HandlerConfig :: handler_config()
    } |
    {module_level, logger:level(), [module()]}
].
```

You can set the log level for all log handlers by setting the `log_level` in this configuration map. Any messages that are logged at levels "higher" than or equal to the configured log level will be logged by all log handlers.

The standard logger (`logger_std_h`) is included by default, if no default logger is specified (and if the default logger is not disabled – see below). The standard logger will output log events to the console.

You can specify multiple log handlers in the `logger` configuration. If a log entry is allowed for a given log level, then each log handler will handle the log message. For example, you might have a log handler that sends messages over the network to a syslog daemon, or you might have another handler that writes log messages to a file.

You can pass handler configuration int the `config` element of the `handler_config()` you specify when specifying a logger. The value of the `config` element can be any term and is made available to log handlers when events are logged (see below).

If the tuple `{handler, default, undefined}` is included in the logger configuration, the default logger will be disabled.

At most one `default` logger can be specified. If you want to replace the `default` logger (`logger_std_h`), then specify a logger with the handler id `default`.

You can specify different log levels for specific modules. For example, if you want to set the default log level for all handlers to be `notice` or higher, you can set the log level for a given module to `info`, and all `info` and higher messages will be logged for that module or set of modules. Conversely, you can "quiet" a module if it is particularly noisy by setting its level to something relatively high.

For more information about how to configure the logging subsystem, see the Kernel Configuration Parameters section of the OTP Logging chapter.

You can stop the `logger_manager` via the `logger_manager:stop/0` function:

```
ok = logger_manager:stop()
```

Writing your own log handler

Additional loggers can be enabled via handler specifications. A handler module must implement and export the `log/2` function, which takes a log event and a term containing the configuration for the logger handler instance.

For example:

```
-module(my_module).

-export([..., log/2, ...]).

log(LogEvent, HandlerConfig) ->
    %% do something with the log event
    %% return value is ignored
```

You can specify this handler in the `logger_manager` configuration (see above) via a stanza such as:

```
{handler, my_id, my_module, HandlerConfig}
```

A `LogEvent` is a map structure containing the following fields:

Key	Type	Description
`timestamp`	`integer()`	The time (in microseconds since the UNIX epoch) at which the log event was generated
`level`	`logger:level()`	The log level with which the log event was generated
`pid`	`pid()`	The process id of the Erlang process in which the event was generated
`msg`	`string() | {string(), list()}`	The message format and arguments passed when the event was generated
`meta`	`map()`	Metadata passed when the event was generated.

If the log event was generated using a logging macro, then the `meta` map also contains a `location` field with the following fields:

Key	Type	Description
`file`	`string()`	The path of the file in which the event was generated
`line`	`non_neg_integer()`	The line number in the file in which the event was generated
`mfa`	`{module(), function_name(), arity()}`	The MFA of the function in which the event was generated

The `HandlerConfig` is a map structure containing the id and module of the handler and is passed into the log handler via configuration of the `logger_manager` (see above).

5.4.5 Process Management

You can obtain a list of all processes in the system via `erlang:processes/0`:

```
Pids = erlang:processes().
```

And for each process, you can get detailed process information via the `erlang:process_info/2` function:

```
io:format("Heap size for Pid ~p: ~p~n", [Pid, erlang:process_info(Pid,
 heap_size)]).
```

The return value is a tuple containing the key passed into the `erlang:process_info/2` function and its associated value.

The currently supported keys are enumerated in the following table:

Key	Value Type	Description
`heap_size`	`non_neg_integer()`	Number of terms (in machine words) used in the process heap
`stack_size`	`non_neg_integer()`	Number of terms (in machine words) used in the process stack
`message_queue_len`	`non_neg_integer()`	Number of unprocessed messages in the process mailbox
`memory`	`non_neg_integer()`	Total number of bytes used by the process (estimate)

See the `word_size` key in the System APIs section for information about how to find the number of bytes used in a machine word on the current platform.

5.4.6 External Term Format

The `erlang:term_to_binary/1` function can be used to serialize arbitrary term data into and out of binary data. These operations can be useful for applications that wish to share term data over some network protocol, such as HTTP or MQTT, or wish to store serialized term data in some permanant sttorage (e.g., Non-volatile storage on ESP32 devices).

For example, to convert a term to a binary, use `erlang:term_to_binary/1`, e.g.,

```
%% erlang
Term = ...
Binary = erlang:term_to_binary(Term),
```

And to convert the binary back to a term, use `erlang:binary_to_term/1,2`, e.g.,

```
%% erlang
Binary = ...
{Term, _Used} = erlang:binary_to_term(Binary, [used]),
```

By default, AtomVM will encode all atoms using UTF-8 encoding. This encoding is the default encoding for OTP-26 and later releases.

For more information about Erlang external term format, consult the Erlang Documentation

5.4.7 System APIs

You can obtain system information about the AtomVM virtual machine via the `erlang:system_info/1` function, which takes an atom parameter designating the desired datum. Allowable parameters include

- `process_count` The number of processes running in the system.
- `port_count` The number of ports running in the system.
- `atom_count` The number of atoms allocated in the system.
- `word_size` The word size (in bytes) on the current platform (typically 4 or 8).
- `atomvm_version` The version of AtomVM currently running (as a binary).

For example,

```
io:format("Atom Count: ~p~n", [erlang:system_info(atom_count)]).
```

Note Additional platform-specific information is supported, depending on the platform type. See below.

Use the `atomvm:platform/0` to obtain the system platform on which your code is running. The return value of this function is an atom who's value will depend on the platform on which your application is running.

```
case atomvm:platform() of
    esp32 ->
        io:format("I am running on an ESP32!~n");
    stm32 ->
        io:format("I am running on an STM32!~n");
    generic_unix ->
        io:format("I am running on a UNIX box!~n")
end.
```

Use `erlang:garbage_collect/0` or `erlang:garbage_collect/1` to force the AtomVM garbage collector to run on a give process. Garbage collection will in general happen automatically when additional free space is needed and is rarely needed to be called explicitly.

The 0-arity version of this function will run the garbage collector on the currently executing process.

```
Pid = ... %% get a reference to some pid
ok = erlang:garbage_collect(Pid).
```

Use the `erlang:memory/1` function to obtain information about allocated memory.

Currently, AtomVM supports the following types:

Type	Description
`binary`	Return the total amount of memory (in bytes) occupied by (reference counted) binaries

Note Binary data small enough to be stored in the Erlang process heap are not counted in this measurement.

5.4.8 System Time

AtomVM supports numerous function for accessing the current time on the device.

Use `erlang:timestamp/0` to get the current time since the UNIX epoch (Midnight, Jan 1, 1970, UTC), at microsecond granularity, expressed as a triple (mega-seconds, seconds, and micro-seconds):

```
{MegaSecs, Secs, MicroSecs} = erlang:timestamp().
```

Use `erlang:system_time/1` to obtain the seconds, milliseconds or microseconds since the UNIX epoch (Midnight, Jan 1, 1970, UTC):

```
Seconds = erlang:system_time(second).
MilliSeconds = erlang:system_time(millisecond).
MicroSeconds = erlang:system_time(microsecond).
```

Use `erlang:monotonic_time/1` to obtain a (possibly not strictly) monotonically increasing time measurement. Use the same time units to convert to seconds, milliseconds, or microseconds:

```
Seconds = erlang:monotonic_time(second).
MilliSeconds = erlang:monotonic_time(millisecond).
MicroSeconds = erlang:monotonic_time(microsecond).
```

Caution! Note `erlang:monotonic_time/1` should not be used to calculate the wall clock time, but instead should be used by applications to compute time differences in a manner that is independent of the system time on the device, which might change, for example, due to NTP, leap seconds, or similar operations that may affect the wall time on the device.

Use `erlang:universaltime/0` to get the current time at second resolution, to obtain the year, month, day, hour, minute, and second:

```
{{Year, Month, Day}, {Hour, Minute, Second}} = erlang:universaltime().
```

On some platforms, you can use the `atomvm:posix_clock_settime/2` to set the system time. Supply a clock id (currently, the only supported clock id is the atom `realtime`) and a time value as a tuple, containing seconds and nanoseconds since the UNIX epoch (midnight, January 1, 1970). For example,

```
SecondsSinceUnixEpoch = ... %% acquire the time
atomvm:posix_clock_settime(realtime, {SecondsSinceUnixEpoch, 0})
```

Warning This operation is not supported yet on the `stm32` platform. On most UNIX platforms, you typically need `root` permission to set the system time.

On the ESP32 platform, you can use the Wifi network to set the system time automatically. For information about how to set system time on the ESP32 using SNTP, see the Network Programming Guide.

To convert a time (in seconds, milliseconds, or microseconds from the UNIX epoch) to a date-time, use the `calendar:system_time_to_universal_time/2` function. For example,

```
Milliseconds = ... %% get milliseconds from the UNIX epoch
{
    {Year, Month, Day}, {Hour, Minute, Second}
} = calendar:system_time_to_universal_time(Milliseconds, millisecond).
```

Valid time units are `second`, `millisecond`, and `microsecond`.

5.4.9 Date and Time

A `datetime()` is a tuple containing a date and time, where a date is a tuple containing the year, month, and day (in the Gregorian calendar), expressed as integers, and a time is an hour, minute, and second, also expressed in integers.

The following Erlang type specification enumerates this type:

```
-type year() :: integer().
-type month() :: 1..12.
-type day() :: 1..31.
-type date() :: {year(), month(), day()}.
-type gregorian_days() :: integer().
-type day_of_week() :: 1..7.
-type hour() :: 0..23.
-type minute() :: 0..59.
-type second() :: 0..59.
-type time() :: {hour(), minute(), second()}.
-type datetime() :: {date(), time()}.
```

Erlang/OTP uses the Christian epoch to count time units from year 0 in the Gregorian calendar. The, for example, the value 0 in Gregorian seconds represents the date Jan 1, year 0, and midnight (UTC), or in Erlang terms, `{{0, 1, 1}, {0, 0, 0}}`.

Attention! AtomVM is currently limited to representing integers in at most 64 bits, with one bit representing the sign bit. However, even with this limitation, AtomVM is able to resolve microsecond values in the Gregorian calendar for over 292,000 years, likely well past the likely lifetime of an AtomVM application (unless perhaps launched on a deep space probe).

The `calendar` module provides useful functions for converting dates to Gregorian days, and date-times to Gregorian seconds.

To convert a `date()` to the number of days since January 1, year 0, use the `calendar:date_to_gregorian_days/1` function, e.g.,

```
GregorianDays = calendar:date_to_gregorian_days({2023, 7, 23})
```

To convert a `datetime()` to convert the number of seconds since midnight January 1, year 0, use the `calendar:datetime_to_gregorian_seconds/1` function, e.g.,

```
GregorianSeconds = calendar:datetime_to_gregorian_seconds({{2023, 7, 23}, {13, 31,
 7}})
```

Warning The `calendar` module does not support year values before year `0`.

5.4.10 Miscellaneous APIs

Use `atomvm:random/0` to generate a random unsigned 32-bit integer in the range `0..4294967295`:

```
RandomInteger = atomvm:random().
```

Use `crypto:strong_rand_bytes/1` to return a randomly populated binary of a specified size:

```
RandomBinary = crypto:strong_rand_bytes(32).
```

Use `base64:encode/1` and `base64:decode/1` to encode to and decode from Base64 format. The input value to these functions may be a binary or string. The output value from these functions is an Erlang binary.

```
Encoded = base64:encode(<<"foo">>).
<<"foo">> = base64:decode(Encoded).
```

You can Use `base64:encode_to_string/1` and `base64:decode_to_string/1` to perform the same encoding, but to return values as Erlang list structures, instead of as binaries.

5.4.11 StackTraces

You can obtain information about the current state of a process via stacktraces, which provide information about the location of function calls (possibly including file names and line numbers) in your program.

Currently in AtomVM, stack traces can be obtained in one of following ways:

- via try-catch blocks
- via catch blocks, when an error has been raised via the `error` Bif.

Note AtomVM does not support `erlang:get_stacktrace/0` which was deprecated in Erlang /OTP 21 and 22, stopped working in Erlang/OTP 23 and was removed in Erlang/OTP 24. Support for accessing the current stacktrace via `erlang:process_info/2` may be added in the future.

For example a stack trace can be bound to a variable in the catch clause in a try-catch block:

```
try
    do_something()
catch
    _Class:_Error:Stacktrace ->
        io:format("Stacktrace: ~p~n", [Stacktrace])
end
```

Alternatively, a stack trace can be bound to the result of a `catch` expression, but only when the error is raised by the `error` Bif. For example,

```
{'EXIT', {foo, Stacktrace}} = (catch error(foo)),
io:format("Stacktrace: ~p~n", [Stacktrace])
```

Stack traces are printed to the console in a crash report, for example, when a process dies unexpectedly.

Stacktrace data is represented as a list of tuples, each of which represents a stack "frame". Each tuple is of the form:

```
[{Module :: module(), Function :: atom(), Arity :: non_neg_integer(), AuxData ::
 aux_data()}]
```

where `aux_data()` is a (possibly empty) properties list containing the following elements:

```
[{file, File :: string(), line, Line :: pos_integer()}]
```

Stack frames are ordered from the frame "closest" to the point of failure (the "top" of the stack) to the frame furthest from the point of failure (the "bottom" of the stack).

Stack frames will contain file and line information in the AuxData list if the BEAM files (typically embedded in AVM files) include <<"Line">> chunks generated by the compiler. Otherwise, the AuxData will be an empty list.

Tip Adding line information to BEAM files not only increases the size of BEAM files in storage, but calculation of file and line information can have a non-negligible impact on memory usage. Memory-sensitive applications should consider not including line information in BEAM files.

The `packbeam` tool does include file and line information in the AVM files it creates by default, but file and line information can be omitted via a command line option. For information about the packbeam too, see the `atomvm_packbeam` tool.

5.4.12 ETS Tables

AtomVM includes a limited implementation of the Erlang `ets` interface, allowing applications to efficiently store term data in a potentially shared key-value store. Conceptually, and ETS table is a collection of key-value pairs (represented as Erlang tuples), which can be efficiently stored, retrieved, and deleted using insertion, lookup, and deletion functions across processes. Storage and retrieval of data in ETS tables is typically faster than communicating with a process which stores state, but still comes at a cost of copying data in and out of the ETS tables.

The current AtomVM implementation of ETS is limited to the `set` table type, meaning that all entries in an ETS table are unique, and that the entries in the ETS table are unordered, when enumerated.

The `ordered_set`, `bag`, and `duplicate_bag` OTP ETS types are not currently supported.

The lifecycle of an ETS table is associated with the lifecycle of the Erlang process that created it. An Erlang process may create as many ETS tables as memory permits, but ETS tables are automatically destroyed when the process with which they are associated terminates.

Note. AtomVM does not currently support transfer of ETS table ownership across processes.

To create an ETS table, use the `ets:new/2` function:

```
TableId = ets:new(my_table, [])
```

The first argument is the table name, which may be optionally registered internally. The second parameter is a list of configuration options. The return value is an opaque reference to the table, which can be used in subsequent ETS operations. If the table is specified as a named table (see below), then the return value is the atom used to name the table (i.e., the first parameter to the `ets:new/2` function).

The process that creates an ETS table becomes the "owner" of the ETS table. ETS tables owned by a process are automatically destroyed when the process terminates.

The following configuration options are supported:

Access Type	Description
`named_table`	If set, the table name is registered internally and can be used as the table id for subsequent ETS operations. If this option is set, the return value from `ets:new/2` is the table name specified in the first parameter. By default, ETS tables are not named.
`{keypos, K}`	The position of the key field in table entries (Erlang tuples). Key position values should be in the range `{1..n}`, where `n` is the minimum arity of any entry in the table. If unspecified, the default key position is `1`. An attempt to insert an entry into a ETS table whose arity is less than the specified key position will result in a `badarg` error.
`private`	Only the owning process may read from or write to the ETS table.
`protected`	The owning process may read and write to the ETS table; any other process can only read from the ETS table.
`public`	Any process may read from or write to the ETS table.

Note that the `keypos`, `private`, `protected`, and `public` fields should only be specified once in configuration. The presence of more than one key position or protection field results in behavior that is *undefined*.

In the absence of any protection field, tables are marked `protected`.

To insert an entry into an ETS table, use the `ets:insert/2` function:

```
true = ets:insert(TableId, {foo, bar})
```

Specify the table identifier returned from `ets:new/2`, as well as the tuple data you would like to store in the ETS table. The `keypos`'th field of the tuple can be used for subsequent retrieval or deletion.

If the arity of the supplied tuple entry is less than the configured key position for the ETS table, the `ets:insert/2` function will raise a `badarg` error.

> Note that the fields of a tuple entry, whether they are designated key fields or arbitrary data, can be any term type, not just atoms, as in this example.

If an entry already exists with the same key field, the entry will be over-written in the table.

The return type from this function is the atom `true`. Any errors in insertion will resulting in raising an `error` with an appropriate reason, e.g., `badarg`.

> Note that a process may only insert values into an ETS table if they are permitted; i.e., either they are the owner of the table, or if the table is `public`.

To retrieve an entry from an ETS table, use the `ets:lookup/2` function:

```
[{foo, bar}] = ets:lookup(TableId, foo)
```

Specify the table identifier returned from `ets:new/2`, as well as a key with which you would like to search the table. This function will search the ETS table using the `keypos`'th field of tuples in the ETS table for retrieval.

The return value is a list containing the found object(s). An empty list (`[]`) indicates that there is no such entry in the specified ETS table.

> Note. Since the only table type currently supported is `set`, the return value will only contain a singleton value, if an entry exists in the table under the specified key.
>
> Note that a process may only look up values from an ETS table if they are permitted; i.e., either they are the owner of the table, or if the table is `protected` or `public`.

To delete an entry from an ETS table, use the `ets:delete/2` function:

```
true = ets:delete(Table, foo)
```

Specify the table identifier returned from `ets:new/2`, as well as a key with which you would like to search the table to delete the entry. This function will search the ETS table using the `keypos`'th field of tuples in the ETS table for retrieval.

The return value from this function is the atom `true`, regardless of whether the entry existed previously. Any errors in deletion will resulting in raising an `error` with an appropriate reason, e.g., `badarg`.

> Note that a process may only delete values from an ETS table if they are permitted; i.e., either they are the owner of the table, or if the table is `public`.

5.4.13 Reading data from AVM files

AVM files are generally packed BEAM files, but they can also contain non-BEAM files, such as plain text files, binary data, or even encoded Erlang terms.

Typically, these files are included from the `priv` directory in a build tree, for example, when using the `atomvm_rebar3_plugin`, though the `atomvm_packbeam` tool allow you to specify any location for files to include in AVM files.

By convention, these files obey the following path in an AVM file:

```
<application-name>/priv/<file-path>
```

For example, if you wanted to embed `my_file.txt` into your application AVM file (where your application name is, for example, `my_application`), you would use:

```
my_application/priv/my_file.txt
```

The `atomvm:read_priv/2` function can then be used to extract the contents of this file into a binary, e.g.,

```
MyFileBin = atomvm:read_priv(my_application, <<"my_file.txt">>)
```

Tip Embedded files may contain path separators, so for example `<<"my_files/my_file.txt">>` would be used if the AVM file embeds `my_file.txt` using the path `my_application/priv/my_files/my_file.txt`

For more information about how to embed files into AVM files, see the `atomvm_rebar3_plugin`, and the `atomvm_rebar3_plugin` section of the AtomVM Tooling guide.

5.4.14 Code Loading

AtomVM provides a limited set of APIs for loading code and data embedded dynamically at runtime.

To load an AVM file from binary data, use the `atomvm:add_avm_pack_binary/2` function. Supply a reference to the AVM data, together with a (possibly empty) list of options. Specify a `name` option, whose value is an atom, if you wish to close the AVM data at a later point in the program.

For example:

```
AVMData = ... %% load AVM data into memory as a binary
ok = atomvm:add_avm_pack_binary(AVMData, [{name, my_avm}])
```

You can also load AVM data from a file (on the `generic_unix` platform) or from a flash partition (on ESP32 platforms) using the `atomvm:add_avm_pack_file/2` function. Specify a string (or binary) as the path to the AVM file, together with a list of options, such as `name`.

For example:

```
ok = atomvm:add_avm_pack_file("/path/to/file.avm", [{name, my_avm}])
```

On `esp32` platforms, the partition name should be prefixed with the string `/dev/partition/by-name/`. Thus, for example, if you specify `/dev/partition/by-name/main2.avm` as the partition, the ESP32 flash should contain a data partition with the name `main2.avm`

For example:

```
ok = atomvm:add_avm_pack_file("/dev/partition/by-name/main2.avm", [])
```

To close a previous opened AVM by name, use the `atomvm:close_avm_pack/2` function. Specify the name of the AVM pack used to add

```
ok = atomvm:close_avm_pack(my_avm, [])
```

Important Currently, the options parameter is ignored, so use the empty list (`[]`) for forward compatibility.

You can load an individual BEAM file using the `code:load_binary/3` function. Specify the Module name (as an atom), as well as the BEAM data you have loaded into memory.

For Example:

```
BEAMData = ... %% load BEAM data into memory as a binary
```

```
{module, Module} = code:load_binary(Module, Filename, BEAMData)
```

Attention! The `Filename` parameter is currently ignored.

You can load an individual BEAM file from the file system using the `code:load_abs/1` function. Specify the path to the BEAM file. This path should not include the `.beam` extension, as this extension will be added automatically.

For example:

```
{module, Module} = code:load_abs("/path/to/beam/file/without/beam/extension")
```

Attention! This function is currently only supported on the `generic_unix` platform.

5.4.15 Math

AtomVM supports the following standard functions from the OTP `math` module:

- `cos/1`
- `acos/1`
- `acosh/1`
- `asin/1`
- `asinh/1`
- `atan/1`
- `atan2/2`
- `atanh/1`
- `ceil/1`
- `cosh/1`
- `exp/1`
- `floor/1`
- `fmod/2`
- `log/1`
- `log10/1`
- `log2/1`
- `pow/2`
- `sin/1`
- `sinh/1`
- `sqrt/1`
- `tan/1`
- `tanh/1`
- `pi/0`

The input values for these functions may be `float` or `integer` types. The return value is always a value of `float` type.

Input values that are out of range for the specific mathematical function or which otherwise are invalid or yield an invalid result (e.g., division by 0) will result in a `badarith` error.

Attention! If the AtomVM virtual machine is built with floating point arithmetic support disabled, these functions will result in a `badarg` error.

5.4.16 Cryptographic Operations

You can hash binary date using the `crypto:hash/2` function.

```
crypto:hash(sha, [<<"Some binary">>, $\s, "data"])
```

This function takes a hash algorithm, which may be one of:

```
-type md_type() :: md5 | sha | sha224 | sha256 | sha384 | sha512.
```

and an IO list. The output type is a binary, who's length (in bytes) is dependent on the algorithm chosen:

Algorithm	Hash Length (bytes)
`md5`	16
`sha`	20
`sha224`	32
`sha256`	32
`sha384`	64
`sha512`	64

Attention! The `crypto:hash/2` function is currently only supported on the ESP32 and generic UNIX platforms.

You can also use the legacy `erlang:md5/1` function to compute the MD5 hash of an input binary. The output is a fixed-length binary (16 bytes)

```
Hash = erlang:md5(<<foo>>).
```

On ESP32, you can perform symmetric encryption and decryption of any iodata data using `crypto_one_time/4,5` function.

Following ciphers are supported:

Without IV (using `crypto_one_time/4`):

- `aes_128_ecb`
- `aes_192_ecb`
- `aes_256_ecb`

With IV (using `crypto_one_time/5`):

- `aes_128_cbc`
- `aes_192_cbc`
- `aes_256_cbc`
- `aes_128_cfb128`
- `aes_192_cfb128`
- `aes_256_cfb128`
- `aes_128_ctr`
- `aes_192_ctr`
- `aes_256_ctr`

The function is implemented using mbedTLS, so please to its documentation for further details.

Please refer to Erlang crypto documentation for additional details about these two functions.

Important Note: mbedTLS doesn't support padding for ciphers other than CCB, so block size must be accounted otherwise output will be truncated.

5.5 ESP32-specific APIs

Certain APIs are specific to and only supported on the ESP32 platform. This section describes these APIs.

5.5.1 System-Level APIs

As noted above, the `erlang:system_info/1` function can be used to obtain system-specific information about the platform on which your application is deployed.

You can request ESP32-specific information using using the following input atoms:

- `esp32_free_heap_size` Returns the available free space in the ESP32 heap.
- `esp32_largest_free_block` Returns the size of the largest free continuous block in the ESP32 heap.
- `esp32_minimum_free_size` Returns the smallest ever free space available in the ESP32 heap since boot, this will tell you how close you have come to running out of free memory.
- `esp32_chip_info` Returns map of the form `#{features := Features, cores := Cores, revision := Revision, model := Model}`, where `Features` is a list of features enabled in the chip, from among the following atoms: `[emb_flash, bgn, ble, bt]`; `Cores` is the number of CPU cores on the chip; `Revision` is the chip version; and `Model` is one of the following atoms: `esp32`, `esp32_s2`, `esp32_s3`, `esp32_c3`, etc.
- `esp_idf_version` Return the IDF SDK version, as a string.

For example,

```
FreeHeapSize = erlang:system_info(esp32_free_heap_size).
```

5.5.2 Non-volatile Storage

AtomVM provides functions for setting, retrieving, and deleting key-value data in binary form in non-volatile storage (NVS) on an ESP device. Entries in NVS survive reboots of the ESP device, and can be used a limited "persistent store" for key-value data.

Warning NVS storage is limited in size, and NVS keys are restricted to 15 characters. Try to avoid writing frequently to NVS storage, as the flash storage may degrade more rapidly with repeated writes to the medium.

NVS entries are stored under a namespace and key, both of which are expressed as atoms. AtomVM uses the namespace `atomvm` for entries under its control. Applications may read from and write to the `atomvm` namespace, but they are strongly discouraged from doing so, except when explicitly stated otherwise.

To set a value in non-volatile storage, use the `esp:nvs_set_binary/3` function, and specify a namespace, key, and value:

```
Namespace = <<"my-namespace">>,
Key = <<"my-key">>,
esp:set_binary(Namespace, Key, <<"some-value">>).
```

To retrieve a value in non-volatile storage, use the `esp:nvs_get_binary/2` function, and specify a namespace and key. You can optionally specify a default value (of any desired type), if an entry does not exist in non-volatile storage:

```
Value = esp:get_binary(Namespace, Key, <<"default-value">>).
```

To delete an entry, use the `esp:nvs_erase_key/2` function, and specify a namespace and key:

```
ok = esp:erase_key(Namespace, Key).
```

You can delete all entries in a namespace via the `esp:nvs_erase_all/1` function:

```
ok = esp:erase_all(Namespace).
```

Finally, you can delete all entries in all namespaces on the NVS partition via the `esp:nvs_reformat/0` function:

```
ok = esp:reformat().
```

Applications should use the `esp:nvs_reformat/0` function with caution, in case other applications are making using the non-volatile storage.

Caution! NVS entries are currently stored in plaintext and are not encrypted. Applications should exercise caution if sensitive security information, such as account passwords, are stored in NVS storage.

5.5.3 Storage

AtomVM provides support for mounting and unmounting storage on ESP32 devices, such as SD cards or internal flash memory. This functionality is accessible through the `esp:mount/4` and `esp:umount/1` functions.

Mounting MMC SD card

To mount a MMC SD card, use the `esp:mount/4` function:

```
case esp:mount("sdmmc", "/sdcard", fat, []) of
    {ok, MountedRef} ->
        io:format("SD card mounted successfully~n"),
        {ok, MountedRef};
    {error, Reason} ->
        io:format("Failed to mount SD card: ~p~n", [Reason]),
        {error, Reason}
end.
```

Mounting SPI SD card

To mount a SPI SD card, first create a SPI instance configured for your specific board, then use the `esp:mount/4` function:

```
SPIConfig = [
    {bus_config, [
        {miso, 19},
        {mosi, 23},
        {sclk, 18},
        {peripheral, "spi3"}
    ]}],
SPI = spi:open(SPIConfig),
case esp:mount("sdspi", "/sdcard", fat, [{spi_host, SPI}, {cs, 5}]) of
    {ok, MountedRef} ->
        io:format("SD card mounted successfully~n"),
        {ok, MountedRef};
    {error, Reason} ->
        io:format("Failed to mount SD card: ~p~n", [Reason]),
        {error, Reason}
end.
```

Mounting internal flash

To mount internal flash, use the `esp:mount/4` function:

```
case esp:mount("/dev/partition/by-name/partition_name", "/test", fat, []) of
    {ok, MountedRef} ->
        io:format("Flash mounted successfully~n"),
        {ok, MountedRef};
    {error, Reason} ->
        io:format("Failed to mount partition: ~p~n", [Reason]),
        {error, Reason}
end.
```

Unmounting Storage

To unmount a previously mounted storage device, use the `esp:umount/1` function, with the reference returned from `esp:mount/4`:

```
case esp:umount(MountedRef) of
    ok ->
        io:format("Storage unmounted successfully~n");
    {error, Reason} ->
        io:format("Failed to unmount storage: ~p~n", [Reason])
end.
```

These functions allow you to work with external storage devices or partitions on your ESP32, enabling you to read from and write to files on the mounted filesystem. This can be particularly useful for applications that need to store or access large amounts of data that don't fit in the device's main memory or non-volatile storage.

Important Remember to properly unmount any mounted filesystems before powering off or resetting the device to prevent data corruption.

5.5.4 Restart and Deep Sleep

You can use the `esp:restart/0` function to immediately restart the ESP32 device. This function does not return a value.

```
esp:restart().
```

Use the `esp:reset_reason/0` function to obtain the reason for the ESP32 restart. Possible values include:

- `esp_rst_unknown`
- `esp_rst_poweron`
- `esp_rst_ext`
- `esp_rst_sw`
- `esp_rst_panic`
- `esp_rst_int_wdt`
- `esp_rst_task_wdt`
- `esp_rst_wdt`
- `esp_rst_deepsleep`

- `esp_rst_brownout`
- `esp_rst_sdio`

Use the `esp:deep_sleep/1` function to put the ESP device into deep sleep for a specified number of milliseconds. Be sure to safely stop any critical processes running before this function is called, as it will cause an immediate shutdown of the device.

```
esp:deep_sleep(60*1000).
```

Use the `esp:sleep_get_wakeup_cause/0` function to inspect the reason for a wakeup. Possible return values include:

- `sleep_wakeup_ext0`
- `sleep_wakeup_ext1`
- `sleep_wakeup_timer`
- `sleep_wakeup_touchpad`
- `sleep_wakeup_ulp`
- `sleep_wakeup_gpio`
- `sleep_wakeup_uart`
- `sleep_wakeup_wifi`
- `sleep_wakeup_cocpu`
- `sleep_wakeup_cocpu_trag_trig`
- `sleep_wakeup_bt`
- `undefined` (no sleep wakeup)
- `error` (unknown other reason)

The values matches the semantics of `esp_sleep_get_wakeup_cause`.

```
case esp:sleep_get_wakeup_cause() of
    sleep_wakeup_timer ->
        io:format("Woke up from a timer~n");
    sleep_wakeup_ext0 ->
        io:format("Woke up from ext0~n");
    sleep_wakeup_ext1 ->
        io:format("Woke up from ext1~n");
    _ ->
        io:format("Woke up for some other reason~n")
end.
```

Use the `esp:sleep_enable_ext0_wakeup/2` and `esp:sleep_enable_ext1_wakeup/2` functions to configure ext0 and ext1 wakeup mechanisms. They follow the semantics of `esp_sleep_enable_ext0_wakeup` and `esp_sleep_enable_ext1_wakeup`.

```
-spec shutdown() -> no_return().
shutdown() ->
    % Configure wake up when GPIO 37 is set to low (M5StickC main button)
    ok = esp:sleep_enable_ext0_wakeup(37, 0),
    % Deep sleep for 1 hour
    esp:deep_sleep(60*60*1000).
```

RTC Memory

On ESP32 systems, you can use (slow) "real-time clock" memory to store data between deep sleeps. This storage can be useful, for example, to store interim state data in your application.

Important RTC memory is initialized if power is lost.

To store data in RTC slow memory, use the `esp:rtc_slow_set_binary/1` function:

```
esp:rtc_slow_set_binary(<<"some binary data">>)
```

To retrieve data in RTC slow memory, use the `esp:rtc_slow_get_binary/0` function:

```
Data = esp:rtc_slow_get_binary()
```

Caution! Calling `esp:rtc_slow_get_binary/0` without having stored a binary first using `esp:rtc_slow_set_binary/1`, will raise with badarg. So make sure to wrap such a call with a try/catch or similar.

By default, RTC slow memory in AtomVM is limited to 4098 (4k) bytes. This value can be modified at build time using an IDF SDK `KConfig` setting. For instructions about how to build AtomVM, see the AtomVM Build Instructions.

5.5.5 Miscellaneous ESP32 APIs

- `esp:freq_hz/0` The `esp:freq_hz/0` function can be used to retrieve the clock frequency of the chip.
- `esp:partition_list/0` The `esp:partition_list/0` function can be used to retrieve information about the partitions on an ESP32 flash.

  The return type is a list of tuples, each of which contains the partition id (as a binary), partition type and sub-type (both of which are represented as integers), the start of the partition as an address along with its size, as well as a list of properties about the partition, as a properties list.

  ```
  PartitionList = esp:partition_list(),
  lists:foreach(
     fun({
          PartitionId, PartitionType, PartitionSubtype, PartitionAddress,
   PartitionSize,
          PartitionProperties
          }) ->
              %% ...
          end,
          PartitionList
     )
  ```

  Note The partition properties are currently empty (`[]`).

  See also

  For information about the encoding of partition types and sub-types, see the IDF SDK partition type definitions.

- `esp:get_mac/1` The `esp:get_mac/1` function can be used to retrieve the network Media Access Control (MAC) address for a given interface, `wifi_sta` or `wifi_softap`. The return value is a 6-byte binary, in accordance with the IEEE 802 family of specifications.

  ```
  MacAddress = esp:get_mac(wifi_sta)
  ```

5.6 Peripherals

The AtomVM virtual machine and libraries support APIs for interfacing with peripheral devices connected to the ESP32 and other supported microcontrollers. This section provides information about these APIs. Unless otherwise stated the documentation for these peripherals is specific to the ESP32, most peripherals are not yet supported on rp2040 or stm32 devices - but work is on-going to expand support for these platforms.

5.6.1 GPIO

The GPIO peripheral has nif support on all platforms. One notable difference on the STM32 platform is that `Pin()` is defined as a tuple consisting of the bank (a.k.a. port) and pin number. For example a pin labeled PB7 on your board would be `{b,7}`.

You can read and write digital values on GPIO pins using the `gpio` module, using the `digital_read/1` and `digital_write/2` functions. You must first set the pin mode using the `gpio:set_pin_mode/2` function, using `input` or `output` as the direction parameter.

Digital Read

To read the value of a GPIO pin (`high` or `low`), use `gpio:digital_read/1`.

For ESP32 family:

```
Pin = 2,
gpio:set_pin_mode(Pin, input),
case gpio:digital_read(Pin) of
    high ->
        io:format("Pin ~p is high ~n", [Pin]);
    low ->
        io:format("Pin ~p is low ~n", [Pin])
end.
```

For STM32 only the line with the Pin definition needs to be a tuple:

```
Pin = {c, 13},
gpio:set_pin_mode(Pin, input),
case gpio:digital_read(Pin) of
    high ->
        io:format("Pin ~p is high ~n", [Pin]);
    low ->
        io:format("Pin ~p is low ~n", [Pin])
end.
```

The RP2 has an additional initialization step `gpio:init/1` before using a pin for gpio:

```
Pin = 2,
gpio:init(Pin),
gpio:set_pin_mode(Pin, input),
case gpio:digital_read(Pin) of
    high ->
        io:format("Pin ~p is high ~n", [Pin]);
    low ->
        io:format("Pin ~p is low ~n", [Pin])
end.
```

Digital Write

To set the value of a GPIO pin (`high` or `low`), use `gpio:digital_write/2`.

For ESP32 family:

```
Pin = 2,
gpio:set_pin_mode(Pin, output),
gpio:digital_write(Pin, low).
```

For the STM32 use a pin tuple:

```
Pin = {b, 7},
gpio:set_pin_mode(Pin, output),
gpio:digital_write(Pin, low).
```

RP2 needs the extra `gpio:init/1` before `gpio:read/1` too:

```
Pin = 2,
gpio:init(Pin),
gpio:set_pin_mode(Pin, output),
gpio:digital_write(Pin, low).
```

Interrupt Handling

Interrupts are supported on both the ESP32 and STM32 platforms. They require using the GPIO port driver, using `gpio:open/0` and `gpio:set_direction/3`.

You can get notified of changes in the state of a GPIO pin by using the `gpio:set_int/3` function. This function takes a reference to a GPIO instance, a Pin, and a trigger. Allowable triggers are `rising`, `falling`, `both`, `low`, `high`, and `none` (to disable an interrupt).

When a trigger event occurs, such as a pin rising in voltage, a tuple will be delivered to the process that set the interrupt containing the atom `gpio_interrupt` and the pin.

```
Pin = 2,
GPIO = gpio:open(),
gpio:set_direction(GPIO, Pin, input),
ok = gpio:set_int(GPIO, Pin, rising),
receive
    {gpio_interrupt, Pin} ->
        io:format("Pin ~p is rising ~n", [Pin])
end.
```

You can also use the `gpio:set_int/4` function, and specify a listener `pid()` or registered name as the recipient of interrupt messages as the fourth parameter.

```
Pin = 2,
GPIO = gpio:open(),
gpio:set_direction(GPIO, Pin, input),
Listener = spawn(fun() -> my_gen_statem() end),
ok = gpio:set_int(GPIO, Pin, rising, Listener),
timer:sleep(infinity).
```

Interrupts can be removed by using the `gpio:remove_int/2` function.

Use the `gpio:close/1` function to close the GPIO driver and free any resources in use by it, supplying a reference to a previously opened GPIO driver instance. Any references to the closed GPIO instance are no longer valid after a successful call to this function, and all interrupts will be removed.

```
ok = gpio:close(GPIO).
```

Since only one instance of the GPIO driver is allowed, you may also simply use `gpio:stop/0` to remove all interrupts, free the resources, and close the GPIO driver port.

```
ok = gpio:stop().
```

5.6.2 ESP32 ADC

The esp_adc module provides the functionality to use the ESP32 family SAR ADC peripheral to measure (analog) voltages from a pin and obtain both raw bit values as well as calibrated voltage values in millivolts.

The module provides two sets of APIs for using the ADC peripheral; there is a set of low level resource based nifs, and a gen_server managed set of convenience functions. The nifs rely on unit and channel handle resources for configuring and taking measurements. The convenience functions use the gen_server to maintain these resources and use pin numbers to interact with the driver. Examples for both APIs can be found the AtomVM repository atomvm/examples/erlang/esp32 directory. A demonstration of the simple APIs is as follows:

```
...
    Pin = 33,
    ok = esp_adc:start(Pin, [{bitwidth, bit_12}, {atten, db_2_5}]),
    {ok, {Raw, Mv}} = esp_adc:read(Pin, [raw, voltage, {samples, 48}]),
    io:format("ADC pin ~p raw value=~p millivolts=~p~n", [Pin, Raw, Mv]),
    ok = esp_adc:stop(),
...
```

ESP32 ADC configuration options

Some newer ESP32 family devices only use a single fixed bit width, this is typically 12 bits, but some provide 13 bit resolution. The ESP32 classic supports 9 bit up to 12 bit resolutions. The bitwidth option bit_max will use the highest supported resolution for the device.

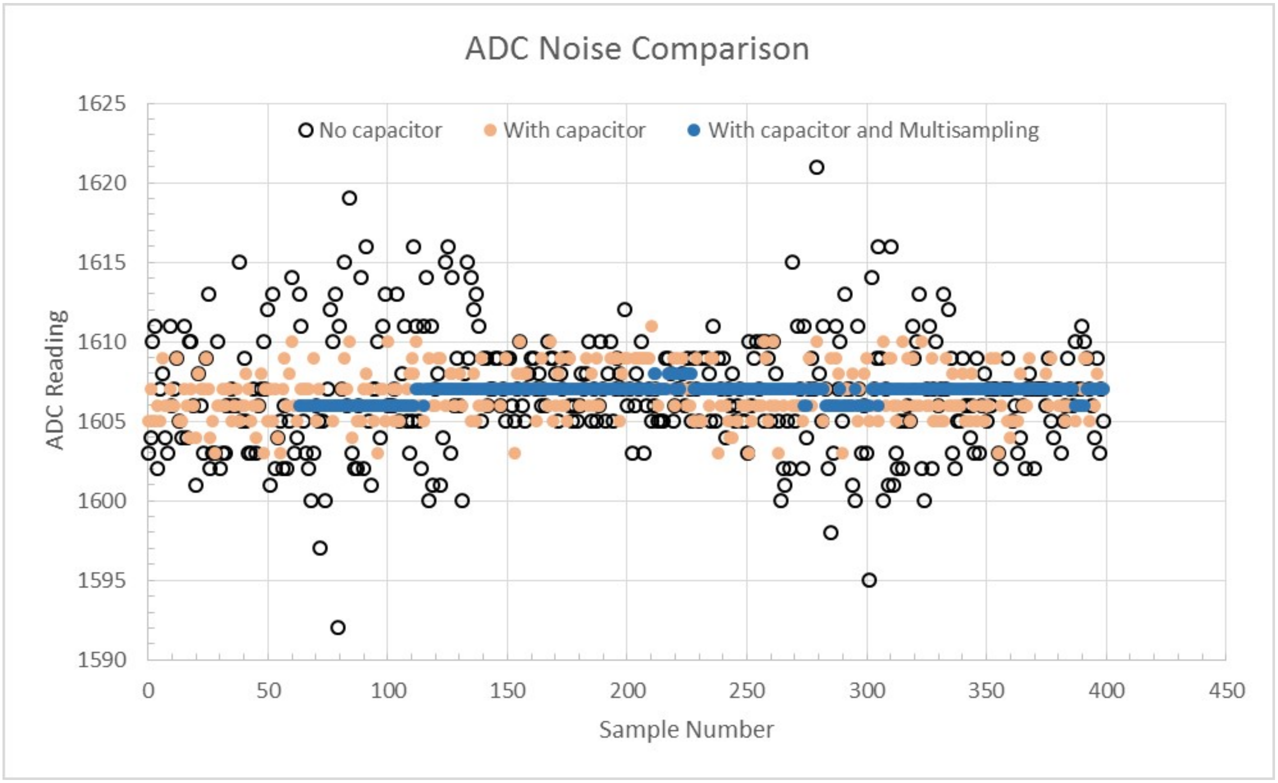
The attenuation option determines the range of voltage to be measured, the specific voltage range for each setting varies by chip, so as always consult your devices datasheet before connecting an ADC pin to a voltage supply to be measured. The chart below depicts the approximate safe voltage ranges for each attenuation level:

Attenuation	Min Millivolts	Max Millivolts
db_0	0-100	750-950
db_2_5	0-100	1050-1250
db_6	0-150	1300-1750
db_11 | db_12	0-150	2450-2500

Consult the datasheet of your device for the exact voltage ranges supported by each attenuation level.

Warning The option db_11 has been superseded by db_12. The option db_11 and will be deprecated in a future release, applications should be updated to use db_12 (except for builds with ESP-IDF versions prior to v5.2). To Continue to support older IDF version builds, the default will remain db_11, which is the maximum tolerated voltage on all builds, as db_12 supported builds will automatically use db_12 in place of db_11. After db_11 is deprecated in all builds (with the sunset of ESP-IDF v5.1 support) the default will be changed to db_12.

Note For a higher degree of accuracy increase the number of sample taken, the default is 64. If highly stable and accurate ADC measurements are required for an application you may need to connect a bypass capacitor (e.g., a 100 nF ceramic capacitor) to the ADC input pad in use, to minimize noise. This chart from the Espressif ADC Calibration Driver documentation shows the difference between the use of a capacitor and without, as well as with a capacitor and multisampling of 64 samples.



You can clearly see the noisy results without a capacitor. This is mitigated by the use of multisampling but without a decoupling capacitor results will likely still contain some noise.

When an ADC channel is configured by the use of `esp_adc:acquire/2,4` or `esp_adc:start/1,2` the driver will select the optimal calibration mechanism supported by the device and channel configuration. If neither the line fitting or curve fitting mechanisms are supported by the device using the provided configuration options an estimated result will be used to provide `voltage` values, based on the formula suggested by Espressif. For chips using the line fitting calibration scheme that do not have the default vref efuse set, a default vref of 1100 mV is used, this is not currently settable.

ESP32 ADC read options

The read options take the form of a proplist, if the key `raw` is true (`{raw, true}` or simply appears in the list as the atom `raw`), then the raw value will be returned in the first element of the returned tuple. Otherwise, this element will be the atom `undefined`.

If the key `voltage` is true (or simply appears in the list as an atom), then a calibrated voltage value will be returned in millivolts in the second element of the returned tuple. Otherwise, this element will be the atom `undefined`.

You may specify the number of samples (1 - 100000) to be taken and averaged over using the tuple `{samples, Samples :: 1..100000}`, the default is `64`.

Warning Using a large number of samples can significantly increase the amount of time before a response, up to several seconds.

5.6.3 I2C

The `i2c` module encapsulates functionality associated with the 2-wire Inter-Integrated Circuit (I2C) interface.

See also

Information about the ESP32 I2C interface can be found in the IDF SDK I2C Documentation.

The AtomVM I2C implementation uses the AtomVM Port mechanism and must be initialized using the `i2c:open/1` function. The single parameter contains a properties list, with the following elements:

Key	Value Type	Required	Description
`scl`	`integer()`	yes	I2C clock pin (SCL)
`sda`	`integer()`	yes	I2C data pin (SDA)
`clock_speed_hz`	`integer()`	yes	I2C clock frequency (in hertz)
`peripheral`	`string()	binary()`	no (platform dependent default)	

For example,

```
I2C = i2c:open([{scl, 21}, {sda, 22}, {clock_speed_hz, 40000}]),
```

Once the port is opened, you can use the returned `I2C` instance to read and write bytes to the attached device.

Both read and write operations require the I2C bus address from which data is read or to which data is written. A devices address is typically hard-wired for the specific device type, or in some cases may be changed by the addition or removal of a resistor.

In addition, you may optionally specify a register to read from or write to, as some devices require specification of a register value. Consult your device's data sheet for more information and the device's I2C bus address and registers, if applicable.

There are two patterns for writing data to an I2C device:

1. Queuing `i2c:write_bytes/2,3,4` write operations between calls to `i2c:begin_transmission/2` and `i2c:end_transmission/1`. In this case, write operations are queued locally and dispatched to the target device when the `i2c:end_transmission/1` operation is called;
2. Writing a byte or sequence of bytes in one `i2c:write_bytes/2,3,4` operation.

The choice of which pattern to use will depend on the device being communicated with. For example, some devices require a sequence of write operations to be queued and written in one atomic write, in which case the first pattern is appropriate. E.g.,

```
ok = i2c:begin_transmission(I2C),
ok = i2c:qwrite_bytes(I2C, DeviceAddress, Register1, <<"some sequence of
 bytes">>),
ok = i2c:qwrite_bytes(I2C, DeviceAddress, Register2, <<"some other of bytes">>),
ok = i2c:end_transmission(I2C),
```

In other cases, you may just need to write a byte or sequence of bytes in one operation to the device:

```
ok = i2c:write_bytes(I2C, DeviceAddress, Register1, <<"write it all in one go">>),
```

Reading bytes is more straightforward. Simply use `i2c:read_bytes/3,4`, specifying the port instance, device address, optionally a register, and the number of bytes to read:

```
{ok, BinaryData} = i2c:read_bytes(I2C, DeviceAddress, Register, Len)
```

To close the I2C driver and free any resources in use by it, use the `i2c:close/1` function, supplying a reference to the I2C driver instance created via `i2c:open/1`:

```
ok = i2c:close(I2C)
```

Once the I2C driver is closed, any calls to `i2c` functions using a reference to the I2C driver instance should return with the value `{error, noproc}`.

5.6.4 SPI

The `spi` module encapsulates functionality associated with the 4-wire Serial Peripheral Interface (SPI) in leader mode.

See also

Information about the ESP32 SPI leader mode interface can be found in the IDF SDK SPI Documentation.

The AtomVM SPI implementation uses the AtomVM Port mechanism and must be initialized using the `spi:open/1` function. The single parameter to this function is a properties list containing:

- `bus_config` – a properties list containing entries for the SPI bus
- `device_config` – an optional properties list containing entries for each device attached to the SPI Bus

The `bus_config` properties list contains the following entries:

Key	Value Type	Required	Description
`poci` (`miso`)	`integer()`	yes	SPI peripheral-out, controller-in pin (MOSI)
`pico` (`mosi`)	`integer()`	yes	SPI peripheral-in, controller-out pin (MISO)
`sclk`	`integer()`	yes	SPI clock pin (SCLK)

The `device_config` entry is a properties list containing entries for each device attached to the SPI Bus. Each entry in this list contains the user-selected name (as an atom) of the device, followed by configuration for the named device.

Each device configuration is a properties list containing the following entries:

Key	Value Type	Required	Description
`clock_speed_hz`	`integer()`	yes	SPI clock frequency (in hertz)
`mode`	`0..3`	yes	SPI mode, indicating clock polarity (`CPOL`) and clock phase (`CPHA`). Consult the SPI specification and data sheet for your device, for more information about how to control the behavior of the SPI clock.
`cs`	`integer()`	yes	SPI chip select pin (CS)
`address_len_bits`	`0..64`	yes	number of bits in the address field of a read /write operation (for example, 8, if the transaction address field is a single byte)
`command_len_bits`	`0..16`	default: 0	number of bits in the command field of a read /write operation (for example, 8, if the transaction command field is a single byte)

For example,

```
SPIConfig = [
    {bus_config, [
        {miso, 19},
        {mosi, 27},
        {sclk, 5}
    ]},
    {device_config, [
        {my_device_1, [
            {clock_speed_hz, 1000000},
            {mode, 0},
            {cs, 18},
            {address_len_bits, 8}
        ]}
        {my_device_2, [
            {clock_speed_hz, 1000000},
            {mode, 0},
            {cs, 15},
            {address_len_bits, 8}
        ]}
    ]}
],
SPI = spi:open(SPIConfig),
...
```

In the above example, there are two SPI devices, one using pin 18 chip select (named `my_device_1`), and once using pin 15 chip select (named `my_device_2`).

Once the port is opened, you can use the returned `SPI` instance, along with the selected device name, to read and write bytes to the attached device.

To read a byte at a given address on the device, use the `spi:read_at/4` function:

```
{ok, Byte} = spi:read_at(SPI, DeviceName, Address, 8)
```

To write a byte at a given address on the device, use the `spi_write_at/5` function:

```
write_at(SPI, DeviceName, Address, 8, Byte)
```

Hint The `spi:write_at/5` takes integer values as inputs and the `spi:read_at/4` returns integer values. You may read and write up to 32-bit integer values via these functions.

Consult your local device data sheet for information about various device addresses to read from or write to, and their semantics.

The above functions are optimized for small reads and writes to an SPI device, typically one byte at a time.

The SPI interface also supports a more generic way to read and write from an SPI device, supporting arbitrary-length reads and writes, as well as a number of different "phases" of writes, per the SPI specification.

These phases include:

- Command phase – write of an up-to 16-bit command to the SPI device
- Address Phase – write of an up-to 64-bit address to the SPI device
- Data Phase – read or write of an arbitrary amount of of data to and from the device.

Any one of these phases may be included or omitted in any given SPI transaction.

In order to achieve this level of flexibility, these functions allow users to specify the SPI transaction through a map structure, which includes fields that specify the behavior of an SPI transaction.

The following table enumerates the permissible fields in this structure:

Key	Value Type	Description
`command`	`integer()` (16-bit)	(Optional) SPI command. The low-order `command_len_bits` are written to the device.
`address`	`integer()` (64-bit)	(Optional) Device address. The low-order `address_len_bits` are written to the device.
`write_data`	`binary()`	(Optional) Data to write

`write_bits`	`non_neg_integer()`	Number of bits to write from `write_data'. If not included,then all bits will be written.
`read_bits`	`non_neg_integer()`	Number of bits to read from the SPI device. If not included, then the same number of bits will be read as were written.

To write a blob of data to the SPI device, for example, you would use:

```
WriteData = <<"some binary data">>,
ok = spi:write(SPI, DeviceName, #{write_data => WriteData})
```

To write and simultaneously read back a blob of data to the SPI device, you would use:

```
{ok, ReadData} = spi:write_read(SPI, DeviceName, #{write_data => WriteData})
```

The size of the returned data is the same as the size of the written data, unless otherwise specified by the `read_bits` field.

Use the `spi:close/1` function to close the SPI driver and free any resources in use by it, supplying a reference to a previously opened SPI driver instance. Any references to the closed SPI instance are no longer valid after a successful call to this function.

```
ok = spi:close(SPI).
```

5.6.5 UART

The `uart` module encapsulates functionality associated with the Universal Asynchronous Receiver /Transmitter (UART) interface supported on ESP32 devices. Some devices, such as NMEA GPS receivers, make use of this interface for communicating with an ESP32.

See also

Information about the ESP32 UART interface can be found in the IDF SDK UART Documentation.

The AtomVM UART implementation uses the AtomVM Port mechanism and must be initialized using the `uart:open/2` function.

The first parameter indicates the ESP32 UART hardware interface. Valid values are:

```
"UART0" | "UART1" | "UART2"
```

The second parameter is a properties list, containing the following elements:

Key	Value Type	Required	Default Value	Description
`rx`	`integer()`	no	-1 (`UART_PIN_NO_CHANGE`)	RX GPIO Pin
`tx`	`integer()`	no	-1 (`UART_PIN_NO_CHANGE`)	TX GPIO Pin
`speed`	`integer()`	no	115200	UART baud rate (bits/sec)
`data_bits`	`5 | 6 | 7 | 8`	no	8	UART data bits
`stop_bits`	`1 | 2`	no	1	UART stop bits
`flow_control`	`hardware | software | none`	no	`none`	Flow control
`parity`	`even | odd | none`	no	`none`	UART parity check

These are the usual RX and TX pins for the various UARTs on the ESP32 (as always check your board specs):

Port	RX pin	TX pin
UART0	GPIO_3	GPIO_1
UART1	GPIO_9	GPIO_10
UART2	GPIO_16	GPIO_17

For example,

```
UART = uart:open("UART0", [{rx, 3}, {tx, 1}, {speed, 9600}])
```

Once the port is opened, you can use the returned `UART` instance to read and write bytes to the attached device.

To read data from the UART channel, use the `uart:read/1` function. The return value from this function is a binary:

```
Bin = uart:read(UART)
```

To write data to the UART channel, use the `uart_write/2` function. The input data is any Erlang I /O list:

```
uart:write(UART, [<<"any">>, $d, $a, $t, $a, "goes", <<"here">>])
```

Consult your local device data sheet for information about the format of data to be read from or written to the UART channel.

To close the UART driver and free any resources in use by it, use the `uart:close/1` function, supplying a reference to the UART driver instance created via `uart:open/2`:

```
ok = uart:close(UART)
```

Once the UART driver is closed, any calls to `uart` functions using a reference to the UART driver instance should return with the value `{error, noproc}`.

5.6.6 LED Control

The LED Control API can be used to drive LEDs, as well as generate PWM signals on GPIO pins.

The LEDC API is encapsulated in the `ledc` module and is a direct translation of the IDF SDK LEDC API, with a natural mapping into Erlang. This API is intended for users with complex use-cases, and who require low-level access to the LEDC APIs.

The `ledc.hrl` module should be used for common modes, channels, duty cycle resolutions, and so forth.

```
-include("ledc.hrl").

%% create a 5khz timer
SpeedMode = ?LEDC_HIGH_SPEED_MODE,
Channel = ?LEDC_CHANNEL_0,
ledc:timer_config([
    {duty_resolution, ?LEDC_TIMER_13_BIT},
    {freq_hz, 5000},
    {speed_mode, ?LEDC_HIGH_SPEED_MODE},
    {timer_num, ?LEDC_TIMER_0}
]).

%% bind pin 2 to this timer in a channel
ledc:channel_config([
    {channel, Channel},
    {duty, 0},
    {gpio_num, 2},
    {speed_mode, ?LEDC_HIGH_SPEED_MODE},
    {hpoint, 0},
    {timer_sel, ?LEDC_TIMER_0}
]).

%% set the duty cycle to 0, and fade up to 16000 over 5 seconds
ledc:set_duty(SpeedMode, Channel, 0).
ledc:update_duty(SpeedMode, Channel).
TargetDuty = 16000.
FadeMs = 5000.
ok = ledc:set_fade_with_time(SpeedMode, Channel, TargetDuty, FadeMs).
```

5.7 Protocols

AtomVM supports network programming on devices that support it, specifically the ESP32 platform, with its built-in support for WIFI networking, and of course on the UNIX platform.

This section describes the network programming APIs available on AtomVM.

5.7.1 Network (ESP32 only)

The ESP32 supports WiFi connectivity as part of the built-in WiFi and Bluetooth radio (and in most modules, an integrated antenna). The WIFI radio on an ESP32 can operate in several modes:

- STA (Station) mode, whereby it acts as a member of an existing WiFi network;
- AP (Access Point) mode, whereby the ESP32 acts as an access point for other devices; or
- AP+STA mode, whereby the ESP32 behaves both as a member of an existing WiFi network and as an access point for other devices.

AtomVM supports these modes of operation via the `network` module, which is used to initialize the network and allow applications to respond to events within the network, such as a network disconnect or reconnect, or a connection to the ESP32 from another device.

See also

Establishment and maintenance of network connections on roaming devices is a complex and subtle art, and the AtomVM `network` module is designed to accommodate as many IoT scenarios as possible. This section of the programmer's guide is deliberately brief and only addresses the most basic scenarios. For a more detailed explanation of the AtomVM `network` module and its many use-cases, please refer to the AtomVM Network Programming Guide.

STA mode

To connect your ESP32 to an existing WiFi network, use the `network:wait_for_sta/1,2` convenience function, which abstracts away some of the more complex details of ESP32 STA mode.

This function takes a station mode configuration, as a properties list, and optionally a timeout (in milliseconds) before connecting to the network should fail. The default timeout, if unspecified, is 15 seconds.

The station mode configuration supports the following options:

Key	Value Type	Required	Default Value	Description
`ssid`	`string() | binary()`	yes	-	WiFi AP SSID
`psk`	`string() | binary()`	yes, if network is encrypted	-	WiFi AP password

`dhcp_hostname`	`string()` | `binary()`	no	`atomvm-<MAC>` where `<MAC>` is the factory-assigned MAC-address of the device	DHCP hostname for the connecting device

Important The WiFi network to which you are connecting must support DHCP and IPv4. IPv6 addressing is not yet supported on AtomVM.

If the ESP32 device connects to the specified network successfully, the device's assigned address, netmask, and gateway address will be returned in an `{ok, ...}` tuple; otherwise, an error is returned.

For example:

```
Config = [
    {ssid, <<"myssid">>},
    {psk,  <<"mypsk">>},
    {dhcp_hostname, <<"mydevice">>}
],
case network:wait_for_sta(Config, 15000) of
    {ok, {Address, _Netmask, _Gateway}} ->
        io:format("Acquired IP address: ~p~n", [Address]);
    {error, Reason} ->
        io:format("Network initialization failed: ~p~n", [Reason])
end
```

Once connected to a WiFi network, you may begin TCP or UDP networking, as described in more detail below.

For information about how to handle disconnections and re-connections to a WiFi network, see the AtomVM Network Programming Guide.

AP mode

To turn your ESP32 into an access point for other devices, you can use the `network:wait_for_ap/1,2` convenience function, which abstracts away some of the more complex details of ESP32 AP mode. When the network is started, the ESP32 device will assign itself the `192.168.4.1` address. Any devices that connect to the ESP32 will take addresses in the `192.168.4/24` network.

This function takes an access point mode configuration, as a properties list, and optionally a timeout (in milliseconds) before starting the network should fail. The default timeout, if unspecified, is 15 seconds.

The access point mode configuration supports the following options:

Key	Value Type	Required	Default Value	Description
`ssid`	`string()` | `binary()`	no	`atomvm-<MAC>` where `<MAC>` is the factory-assigned MAC-address of the device	WiFi AP SSID
`ssid_hidden`	`boolean()`	no	`false`	Whether the AP SSID should be hidden (i.e., not broadcast)

`psk`	`string()` | `binary()`	yes, if network is encrypted	-	WiFi AP password. Warning: If this option is not specified, the network will be an open network, to which anyone who knows the SSID can connect and which is not encrypted.
`ap_max_connections`	`non_neg_integer()`	no	`4`	Maximum number of devices that can be connected to this AP

If the ESP32 device starts the AP network successfully, the `ok` atom is returned; otherwise, an error is returned.

For example:

```erlang
Config = [
    {psk,  <<"mypsk">>}
],
case network:wait_for_ap(Config, 15000) of
    ok ->
        io:format("AP network started at 192.168.4.1~n");
    {error, Reason} ->
        io:format("Network initialization failed: ~p~n", [Reason])
end
```

Once the WiFi network is started, you may begin TCP or UDP networking, as described in more detail below.

For information about how to handle connections and disconnections from attached devices, see the AtomVM Network Programming Guide.

STA+AP mode

For information about how to run the AtomVM network in STA and AP mode simultaneously, see the AtomVM Network Programming Guide.

SNTP

For information about how to use SNTP to synchronize the clock on your device, see the AtomVM Network Programming Guide.

5.7.2 UDP

AtomVM supports network programming using the User Datagram Protocol (UDP) via the `gen_udp` module. This modules obeys the syntax and semantics of the Erlang/OTP `gen_udp` interface.

Attention! Not all of the Erlang/OTP `gen_udp` functionality is implemented in AtomVM. For details, consult the AtomVM API documentation.

To open a UDP port, use the `gen_udp:open/1,2` function. Supply a port number, and if your application plans to receive UDP messages, specify that the port is active via the `{active, true}` property in the optional properties list.

For example:

```erlang
Port = 44404,
case gen_udp:open(Port, [{active, true}, binary]) of
    {ok, Socket} ->
        {ok, SockName} = inet:sockname(Socket)
        io:format("Opened UDP socket on ~p.~n", [SockName])
    Error ->
        io:format("An error occurred opening UDP socket: ~p~n", [Error])
end
```

If the port is active, you can receive UDP messages in your application. They will be delivered as a 5-tuple, starting with the `udp` atom, and containing the socket, address and port from which the message was sent, as well as the datagram packet, itself, as a list (by default) or a binary. To choose the format, pass `list` or `binary` in options, as with Erlang/OTP.

```erlang
receive
    {udp, _Socket, Addr, Port, Packet} ->
        io:format("Received UDP packet ~p from address ~p port ~p~n", [Packet,
 Addr, Port])
end,
```

With a reference to a UDP `Socket`, you can send messages to a target UDP endpoint using the `gen_udp:send/4` function. Specify the UDP socket returned from `gen_udp:open/1,2`, the address (as a 4-tuple of octets), port number, and the datagram packet to send:

```erlang
Packet = <<":?>>,
Address = {192, 168, 1, 101},
Port = 44404,
case gen_udp:send(Socket, Address, Port, Packet) of
    ok ->
        io:format("Sent ~p~n", [Packet]);
    Error ->
        io:format("An error occurred sending a packet: ~p~n", [Error])
end
```

Important IPv6 networking is not currently supported in AtomVM.

5.7.3 TCP

AtomVM supports network programming using the Transport Connection Protocol (TCP) via the `gen_tcp` module. This modules obeys the syntax and semantics of the Erlang/OTP `gen_tcp` interface.

Attention! Not all of the Erlang/OTP `gen_tcp` functionality is implemented in AtomVM. For details, consults the AtomVM API documentation.

Server-side TCP

Server side TCP requires opening a listening socket, and then waiting to accept connections from remote clients. Once a connection is established, the application may then use a combination of sending and receiving packets over the established connection to or from the remote client.

Attention! Programming TCP on the server-side using the `gen_tcp` interface is a subtle art, and this portion of the documentation will not go into all of the design choices available when designing a TCP application.

Start by opening a listening socket using the `gen_tcp:listen/2` function. Specify the port number on which the TCP server should be listening:

```erlang
case gen_tcp:listen(44405, []) of
    {ok, ListenSocket} ->
        {ok, SockName} = inet:sockname(Socket),
        io:format("Listening for connections at address ~p.~n", [SockName]),
        spawn(fun() -> accept(ListenSocket) end);
    Error ->
        io:format("An error occurred listening: ~p~n", [Error])
end.
```

In this particular example, the server will spawn a new process to wait to accept a connection from a remote client, by calling the `gen_tcp:accept/1` function, passing in a reference to the listening socket. This function will block until a client has established a connection with the server.

When a client connects, the function will return a tuple `{ok, Socket}`, where `Socket` is a reference to the connection between the client and server:

```erlang
accept(ListenSocket) ->
    io:format("Waiting to accept connection...~n"),
    case gen_tcp:accept(ListenSocket) of
        {ok, Socket} ->
            {ok, SockName} = inet:sockname(Socket),
            {ok, Peername} = inet:peername(Socket),
            io:format("Accepted connection.  local: ~p peer: ~p~n", [SockName,
 Peername]),
            spawn(fun() -> accept(ListenSocket) end),
            echo();
        Error ->
            io:format("An error occurred accepting connection: ~p~n", [Error])
    end.
```

Note Note that immediately after accepting a connection, this example code will spawn a new process to accept any new connections from other clients.

The socket returned from `gen_tcp:accept/1` can then be used to send and receive messages to the connected client:

```erlang
echo() ->
    io:format("Waiting to receive data...~n"),
    receive
        {tcp_closed, _Socket} ->
            io:format("Connection closed.~n"),
            ok;
        {tcp, Socket, Packet} ->
            {ok, Peername} = inet:peername(Socket),
            io:format("Received packet ~p from ~p.  Echoing back...~n", [Packet,
 Peername]),
            gen_tcp:send(Socket, Packet),
            echo()
    end.
```

In this case, the server program will continuously echo the received input back to the client, until the client closes the connection.

For more information about the `gen_tcp` server interface, consult the AtomVM API Reference Documentation.

Client-side TCP

Client side TCP requires establishing a connection with an endpoint, and then using a combination of sending and receiving packets over the established connection.

Start by opening a connection to another TCP endpoint using the `gen_tcp:connect/3` function. Supply the address and port of the TCP endpoint.

For example:

```erlang
Address = {192, 168, 1, 101},
Port = 44405,
case gen_tcp:connect(Address, Port, []) of
    {ok, Socket} ->
        {ok, SockName} = inet:sockname(Socket),
        {ok, Peername} = inet:peername(Socket),
        io:format("Connected to ~p from ~p~n", [Peername, SockName]);
    Error ->
        io:format("An error occurred connecting: ~p~n", [Error])
end
```

Once a connection is established, you can use a combination of

```erlang
SendPacket = <<":?>>,
case gen_tcp:send(Socket, SendPacket) of
    ok ->
        receive
            {tcp_closed, _Socket} ->
                io:format("Connection closed.~n"),
                ok;
            {tcp, _Socket, ReceivedPacket} ->
                {ok, Peername} = inet:peername(Socket),
                io:format("Received ~p from ~p~n", [ReceivedPacket, Peername])
        end;
    Error ->
        io:format("An error occurred sending a packet: ~p~n", [Error])
end.
```

For more information about the `gen_tcp` client interface, consults the AtomVM API documentation.

5.8 Socket Programming

AtomVM supports a subset of the OTP `socket` interface, giving users more fine-grained control in socket programming.

The OTP socket APIs are relatively new (they were introduced in OTP 22 and have seen revisions in OTP 24). These APIs broadly mirror the BSD Sockets API, and should be familiar to most programmers who have had to work with low-level operating system networking interfaces. AtomVM supports a strict subset of the OTP APIs. Future versions of AtomVM may add additional coverage of these APIs.

The following types are relevant to this interface and are referenced in the remainder of this section:

```erlang
-type domain() :: inet.
-type type() :: stream | dgram.
-type protocol() :: tcp | udp.
-type socket() :: any().
-type sockaddr() :: sockaddr_in().
-type sockaddr_in() :: #{
    family := inet,
    port := port_number(),
    addr := any | loopback | in_addr()
}.
-type in_addr() :: {0..255, 0..255, 0..255, 0..255}.
-type port_number() :: 0..65535.
-type socket_option() :: {socket, reuseaddr} | {socket, linger}.
```

Create a socket using the `socket:open/3` function, providing a domain, type, and protocol. Currently, AtomVM supports the `inet` domain, `stream` and `dgram` types, and `tcp` and `udp` protocols.

For example:

```
{ok, Socket} = socket:open(inet, stream, tcp),
```

5.8.1 Server-side TCP Socket Programming

To program using sockets on the server side, you can bind an opened socket to an address and port number using the `socket:bind/2` function, supplying a map that specifies the address and port number.

This map may contain the following entries:

Key	Type	Default	Description
`family`	`inet`		The address family. (Currently, only `inet` is supported)
`addr`	`in_addr() | any | loopback`		The address to which to bind. The `any` value will bind the socket to all interfaces on the device. The `loopback` value will bind the socket to the loopback interface on the device.
`port`	`port_number()`		The port to which to bind the socket. If no port is specified, the operating system will choose a port for the user.

For example:

```
PortNumber = 8080,
ok = socket:bind(Socket, #{family => inet, addr => any, port => PortNumber}),
```

To listen for connections, use the `socket:listen/1` function:

```
ok = socket:listen(Socket),
```

Once your socket is listening on an interface and port, you can wait to accept a connection from an incoming client using the `socket:accept/1` function.

This function will block the current execution context (i.e., Erlang process) until a client establishes a TCP connection with the server:

```
{ok, ConnectedSocket} = socket:accept(Socket),
```

Tip Many applications will spawn processes to listen for socket connections, so that the main execution context of your application is not blocked.

5.8.2 Client-side TCP Socket Programming

To program using sockets on the client side, you can connect an opened socket to an address and port number using the `socket:connect/2` function, supplying a map that specifies the address and port number.

This map may contain the following entries:

Key	Type	Default	Description
family	inet		The address family. (Currently, only inet is supported)
addr	in_addr() | loopback		The address to which to connect. The loopback value will connect the socket to the loopback interface on the device.
port	port_num()		The port to which to connect the socket.

```
ok = socket:connect(Socket, #{family => inet, addr => loopback, port => 44404})
```

5.8.3 Sending and Receiving Data

Once you have a connected socket (either via socket:connect/2 or socket:accept/1), you can send and receive data on that socket using the socket:send/2 and socket:recv/1 functions. Like the socket:accept/1 function, these functions will block until data is sent to a connected peer (or until the data is written to operating system buffers) or received from a connected peer.

The socket:send/2 function can take a binary blob of data or an io-list, containing binary data.

For example, a process that receives data and echos it back to the connected peer might be implemented as follows:

```
case socket:recv(ConnectedSocket) of
    {ok, Data} ->
        case socket:send(ConnectedSocket, Data) of
            ok ->
                io:format("All data was sent~n");
            {ok, Rest} ->
                io:format("Some data was sent.  Remaining: ~p~n", [Rest]);
            {error, Reason} ->
                io:format("An error occurred sending data: ~p~n", [Reason])
        end;
    {error, closed} ->
        io:format("Connection closed.~n");
    {error, Reason} ->
        io:format("An error occurred waiting on a connected socket: ~p~n",
 [Reason])
end.
```

The socket:recv/1 function will block the current process until a packet has arrived or until the local or remote socket has been closed, or some other error occurs.

Note that the socket:send/2 function may return ok if all of the data has been sent, or {ok, Rest}, where Rest is the remaining part of the data that was not sent to the operating system. If the supplied input to socket:send/2 is an io-list, then the Rest will be a binary containing the rest of the data in the io-list.

5.8.4 Getting Information about Connected Sockets

You can obtain information about connected sockets using the socket:sockname/1 and socket:peername/1 functions. Supply the connected socket as a parameter. The address and port are returned in a map structure

For example:

```
{ok, #{addr := LocalAddress, port := LocalPort}} =
 socket:sockname(ConnectedSocket),
{ok, #{addr := PeerAddress, port := PeerPort}} = socket:peername(ConnectedSocket),
```

5.8.5 Closing and Shutting down Sockets

Use the `socket:close/1` function to close a connected socket:

```
ok = socket:close(ConnectedSocket)
```

Attention! Data that has been buffered by the operating system may not be delivered, when a socket is closed via the `close/1` operation.

For a more controlled way to close full-duplex connected sockets, use the `socket:shutdown/2` function. Provide the atom `read` if you only want to shut down the reads on the socket, `write` if you want to shut down writes on the socket, or `read_write` to shut down both reads and writes on a socket. Subsequent reads or writes on the socket will result in an `einval` error on the calls, depending on how the socket has been shut down.

For example:

```
ok = socket:shutdown(Socket, read_write)
```

5.8.6 Setting Socket Options

You can set options on a socket using the `socket:setopt/3` function. This function takes an opened socket, a key, and a value, and returns `ok` if setting the option succeeded.

Currently, the following options are supported:

Option Key	Option Value	Description
`{socket, reuseaddr}`	`boolean()`	Sets `SO_REUSEADDR` on the socket.
`{socket, linger}`	`#{onoff => boolean(), linger => non_neg_integer()}`	Sets `SO_LINGER` on the socket.
`{otp, rcvbuf}`	`non_neg_integer()`	Sets the default buffer size (in bytes) on receive calls. This value is only used if the `Length` parameter of the `socket:recv` family of functions has the value `0`; otherwise, the specified non-zero length in the `socket:recv` takes precedence. Note that the OTP option value `default` is not currently supported.

For example:

```
ok = socket:setopt(Socket, {socket, reuseaddr}, true),
ok = socket:setopt(Socket, {socket, linger}, #{onoff => true, linger => 0}),
ok = socket:setopt(Socket, {otp, rcvbuf}, 1024),
```

5.8.7 UDP Socket Programming

You can use the `socket` interface to send and receive messages over the User Datagram Protocol (UDP), in addition to TCP.

To use UDP sockets, open a socket using the `dgram` type and `udp` protocol.

For example:

```
{ok, Socket} = socket:open(inet, dgram, udp)
```

To listen for UDP connections, use the `socket:bind/2` function, as described above.

For example:

```
PortNumber = 512,
ok = socket:bind(Socket, #{family => inet, addr => any, port => PortNumber}),
```

Use the `socket:recvfrom/1` function to receive UDP packets from clients on your network. When a packet arrives, this function will return the received packet, as well as the address of the client that sent the packet.

For example:

```
case socket:recvfrom(dSocket) of
    {ok, {From, Packet}} ->
        io:format("Received packet ~p from ~p~n", [Packet, From]);
    {error, Reason} ->
        io:format("Error on recvfrom: ~p~n", [Reason])
end;
```

Important The `socket:recvfrom/1` function will block the current process until a packet has arrived or until the local or remote socket has been closed, or some other error occurs.

Use the `socket:sendto/3` function to send UDP packets to a specific destination. Specify the socket, data, and destination address you would like the packet to be delivered to.

For example:

```
Dest = #{family => inet, addr => loopback, port => 512},
case socket:sendto(Socket, Data, Dest) of
    ok ->
        io:format("Send packet ~p to ~p.~n", [Data, Dest]);
    {ok, Rest} ->
        io:format("Send packet ~p to ~p.  Remaining: ~p~n", [Data, Dest, Rest]);
    {error, Reason} ->
        io:format("An error occurred sending a packet: ~p~n", [Reason])
end
```

Close a UDP socket just as you would a TCP socket, as described above.

5.8.8 Miscellaneous Networking APIs

You can retrieve information about hostnames and services using the `net:getaddrinfo/1` and `net:getaddrinfo/2` functions. The return value is a list of maps each of which contains address information about the host, including its family (`inet`), protocol (`tcp` or `udp`), type (`stream` or `dgram`), and the address, currently an IPv4 tuple.

Important Currently, the `net:getaddrinfo/1,2` functions only supports reporting of IPv4 addresses.

For example:

```
{ok, AddrInfos} = net:getaddrinfo("www.atomvm.net"),

lists:foreach(
    fun(AddrInfo) ->
        #{
            family := Family,
            protocol := Protocol,
            type := Type,
            address := Address
        } = AddrInfo,

        io:format(
            "family: ~p protocol: ~p type: ~p address: ~p", [Family, Protocol,
 Type, Address]
        )

    end,
    AddrInfos
),
```

The `host` parameter can be a domain name (typically) or a dotted pair IPv4 address.

The returned map contains the network family (currently, only `inet` is supported), the protocol, type, and address of the host.

The address is itself a map, containing the family, port and IPv4 address of the requested host, e.g.,

```
#{family => inet, port => 0, addr => {192, 168, 212, 153}}
```

Note The OTP documentation states that the address is returned under the `address` key in the address info map. However, OTP appears to use `addr` as the key. For compatibility with OTP 22 ff., AtomVM supports both the `address` and `addr` keys in this map (they reference the same inner map).

If you want to narrow the information you get back to a specific service type, you can specify a service name or port number (as a string value) as the second parameter:

```
{ok, AddrInfos} = net:getaddrinfo("www.atomvm.net", "https"),
...
```

Service names are well-known identifiers on the internet, but they may vary from operating system to operating system. See the `services(3)` man pages for more information.

Note Narrowing results via the service parameter is not supported on all platforms. In the case where it is not supported, AtomVM will resort to retrying the request without the service parameter.

5.9 Where to go from here

For more examples of how to use the AtomVM APIs check out the AtomVM Example Programs.

If you have not already, you may want to read the chapter on AtomVM Tooling to help you get your applications built and flashed to a microcontroller.

Chapter 6

Network Programming Guide

One of the exciting features of the ESP32 and the Pico W/Pico 2 W is their support for WiFi networking, allowing ESP32 and Pico W/Pico 2 W micro-controllers to communicate with the outside world over common IP networking protocols, such as TCP or IDP. The ESP32 and the Pico W/Pico 2 W can be configured in station mode (STA), whereby the devices connect to an existing access point, as well as "softAP" mode (AP), whereby they function as an access point, to which other stations can connect. The ESP32 also supports a combined STA+softAP mode, which allows the device to function in both STA and softAP mode simultaneously.

AtomVM provides an Erlang API interface for interacting with the WiFi networking layer on ESP32 and Pico W/Pico 2 W devices, providing support for configuring your ESP32 or Pico W/Pico 2 W device in STA mode, AP mode, or a combined STA+AP mode, allowing Erlang/Elixir applications to send and receive data from other devices on a network. This interface is encapsulated in the `network` module, which implements a simple interface for connecting to existing WiFi networks or for functioning as a WiFi access point. The same `network` module is used for both the ESP32 and the Pico W /Pico 2 W.

Once the network has been set up (in STA or AP mode), AtomVM can use various socket interfaces to interact with the socket layer to create a client or server application. For example, on ESP32, AtomVM supports the `gen_udp` and `gen_tcp` APIs, while AtomVM extensions may support HTTP, MQTT, and other protocols built over low-level networking interfaces.

The AtomVM networking API leverages callback functions, allowing applications to be responsive to changes in the underlying network, which can frequently occur in embedded applications, where devices can easily lose and then regain network connectivity. In such cases, it is important for applications to be resilient to changes in network availability, by closing or re-opening socket connections in response to disconnections and re-connections in the underlying network.

This document describes the basic design of the AtomVM network interfaces, and how to interact programmatically with it.

6.1 Station (STA) mode

In STA mode, the ESP32 or the Pico W/Pico 2 W connect to an existing WiFi network.

In this case, the input configuration should be a properties list containing a tuple of the form `{sta, <sta-properties>}`, where `<sta-properties>` is a property list containing configuration properties for the device in station mode.

The `<sta-properties>` property list should contain the following entries:

- `{ssid, string() | binary()}` The SSID to which the device should connect.
- `{psk, string() | binary()}` The password required to authenticate to the network, if required.

The `network:start/1` will immediately return `{ok, Pid}`, where `Pid` is the process ID of the network server instance, if the network was properly initialized, or `{error, Reason}`, if there was an error in configuration. However, the application may want to wait for the device to connect to the target network and obtain an IP address, for example, before starting clients or services that require network access.

Applications can specify callback functions, which get triggered as events emerge from the network layer, including connection to and disconnection from the target network, as well as IP address acquisition.

Callback functions can be specified by the following configuration parameters:

- `{connected, fun(() -> term())}` A callback function which will be called when the device connects to the target network.
- `{disconnected, fun(() -> term())}` A callback function which will be called when the device disconnects from the target network.
- `{got_ip, fun((ip_info()) -> term())}` A callback function which will be called when the device obtains an IP address. In this case, the IPv4 IP address, net mask, and gateway are provided as a parameter to the callback function.

Warning IPv6 addresses are not yet supported in AtomVM.

Callback functions are optional, but are highly recommended for building robust WiFi applications. The return value from callback functions is ignored, and AtomVM provides no guarantees about the execution context (i.e., BEAM process) in which these functions are invoked.

In addition, the following optional parameters can be specified to configure the AP network (ESP32 only):

- `{dhcp_hostname, string()|binary()}` The DHCP hostname as which the device should register (`<<"atomvm-<hexmac>">>`, where `<hexmac>` is the hexadecimal representation of the factory-assigned MAC address of the device).
- `{beacon_timeout, fun(() -> term())}` A callback function which will be called when the device does not receive a beacon frame from the connected access point during the "inactive time" (6 second default, currently not configurable).

The following example illustrates initialization of the WiFi network in STA mode. The example program will configure the network to connect to a specified network. Events that occur during the lifecycle of the network will trigger invocations of the specified callback functions.

```
Config = [
    {sta, [
        {ssid, <<"myssid">>},
        {psk,  <<"mypsk">>},
        {connected, fun connected/0},
        {got_ip, fun got_ip/1},
        {disconnected, fun disconnected/0}
        {dhcp_hostname, <<"myesp32">>}
    ]}
],
{ok, Pid} = network:start(Config),
...
```

The following callback functions will be called when the corresponding events occur during the lifetime of the network connection.

```
connected() ->
    io:format("Connected to AP.~n").

gotIp(IpInfo) ->
    io:format("Got IP: ~p~n", [IpInfo]).
```

```
disconnected() ->
    io:format("Disconnected from AP.~n").
```

In a typical application, the network should be configured and an IP address should be acquired first, before starting clients or services that have a dependency on the network.

6.1.1 STA Mode Convenience Functions

The `network` module supports the `network:wait_for_sta/1,2` convenience functions for applications that do not need robust connection management. These functions are synchronous and will wait until the device is connected to the specified AP. Supply the properties list specified in the `{sta, [...]}` component of the above configuration, in addition to an optional timeout (in milliseconds).

For example:

```
Config = [
    {ssid, <<"myssid">>},
    {psk,  <<"mypsk">>},
    {dhcp_hostname, <<"mydevice">>}
],
case network:wait_for_sta(Config, 15000) of
    {ok, {Address, _Netmask, _Gateway}} ->
        io:format("Acquired IP address: ~p~n", [Address]);
    {error, Reason} ->
        io:format("Network initialization failed: ~p~n", [Reason])
end
```

To obtain the signal strength (in decibels) of the connection to the associated access point use `network:sta_rssi/0`.

6.2 AP mode

In AP mode, the ESP32 starts a WiFi network to which other devices (laptops, mobile devices, other ESP32 devices, etc) can connect. The ESP32 will create an IPv4 network, and will assign itself the address `192.168.4.1`. Devices that attach to the ESP32 in AP mode will be assigned sequential addresses in the `192.168.4.0/24` range, e.g., `192.168.4.2`, `192.168.4.3`, etc.

To initialize the ESP32 device in AP mode, the input configuration should be a properties list containing a tuple of the form `{ap, <ap-properties>}`, where `<ap-properties>` is a property list containing configuration properties for the device in AP mode.

The `<ap-properties>` property list may contain the following entries:

- `{ssid, string() | binary()}` The SSID to which the device should connect.
- `{psk, string() | binary()}` The password required to authenticate to the network, if required. Note that this password must be a minimum of 8 characters.
- `{ap_channel, wifi_channel()}` The channel the access point should use.

If the SSID is omitted in configuration, the SSID name `atomvm-<hexmac>` will be created, where `<hexmac>` is the hexadecimal representation of the factory-assigned MAC address of the device. This name should be sufficiently unique to disambiguate it from other reachable ESP32 devices, but it may also be difficult to read or remember.

If the password is omitted, then an *open network* will be created, and a warning will be printed to the console. Otherwise, the AP network will be started using WPA+WPA2 authentication.

If the channel is omitted the default chanel for esp32 is `1`. This setting is only used while a device is operation is AP mode only. If `ap_channel` is configured, it will be temporarily changed to match the associated access point if AP + STA mode is used and the station is associated with an access point. This is a hardware limitation due to the modem radio only being able to operate on a single channel (frequency) at a time.

The `network:start/1` will immediately return `{ok, Pid}`, where `Pid` is the process id of the network server, if the network was properly initialized, or `{error, Reason}`, if there was an error in configuration. However, the application may want to wait for the device to to be ready to accept connections from other devices, or to be notified when other devices connect to this AP.

Applications can specify callback functions, which get triggered as events emerge from the network layer, including when a station connects or disconnects from the AP, as well as when a station is assigned an IP address.

Callback functions can be specified by the following configuration parameters:

- `{ap_started, fun(() -> term())}` A callback function which will be called when the AP endpoint has started and is ready to be connected to.
- `{sta_connected, fun((Mac::binary()) -> term())}` A callback function which will be called when a device connects to the AP. The MAC address of the connected station, as a 6-byte binary, is passed to the callback function.
- `{sta_disconnected, fun((Mac::binary()) -> term())}` A callback function which will be called when a device disconnects from the AP. The MAC address of the disconnected station, as a 6-byte binary, is passed to the callback function.
- `{sta_ip_assigned, fun((ipv4_address()) -> term())}` A callback function which will be called when the AP assigns an IP address to a station. The assigned IP address is passed to the callback function.

Warning IPv6 addresses are not yet supported in AtomVM.

Callback functions are completely optional, but are highly recommended for building robust WiFi applications. The return value from callback functions is ignored, and AtomVM provides no guarantees about the execution context (i.e., BEAM process) in which these functions are invoked.

In addition, the following optional parameters can be specified to configure the AP network:

- `{ssid_hidden, boolean()}` Whether the AP network should be not be broadcast (false, by default)
- `{max_connections, non_neg_integer()}` The maximum number of devices that can connect to this network (by default, 4)

The following example illustrates initialization of the WiFi network in AP mode. The example program will configure the network to connect to start a WiFi network with the name `myssid` and password `mypsk`. Events that occur during the lifecycle of the network will trigger invocations of the specified callback functions.

```erlang
Config = [
    {ap, [
        {ssid, <<"myssid">>},
        {psk,  <<"mypsk">>},
        {ap_started, fun ap_started/0},
        {sta_connected, fun sta_connected/1},
        {sta_ip_assigned, fun sta_ip_assigned/1},
        {sta_disconnected, fun sta_disconnected/1},
    ]}
],
{ok, Pid} = network:start(Config),
...
```

The following callback functions will be called when the corresponding events occur during the lifetime of the network connection.

```
ap_started() ->
    io:format("AP started.~n").

sta_connected(Mac) ->
    io:format("STA connected with mac ~p~n", [Mac]).

sta_disconnected(Mac) ->
    io:format("STA disconnected with mac ~p~n", [Mac]).

sta_ip_assigned(Address) ->
    io:format("STA assigned address ~p~n", [Address]).
```

In a typical application, the network should be configured and the application should wait for the AP to report that it has started, before starting clients or services that have a dependency on the network.

6.2.1 AP Mode Convenience Functions

The `network` module supports the `network:wait_for_ap/1,2` convenience functions for applications that do not need robust connection management. These functions are synchronous and will wait until the device is successfully starts an AP. Supply the properties list specified in the `{ap, [...]}` component of the above configuration, in addition to an optional timeout (in milliseconds).

For example:

```
Config = [
    {psk,  <<"mypsk">>}
],
case network:wait_for_ap(Config, 15000) of
    ok ->
        io:format("AP network started at 192.168.4.1~n");
    {error, Reason} ->
        io:format("Network initialization failed: ~p~n", [Reason])
end
```

6.3 STA+AP mode

The `network` module can be started in both STA and AP mode. In this case, the ESP32 device will both connect to an access point in its STA mode, and will simultaneously serve as an access point in its role in AP mode.

In order to enable both STA and AP mode, simply provide valid configuration for both modes in the configuration structure supplied to the `network:start/1` function.

6.4 SNTP Support

You may configure the networking layer to automatically synchronize time on the ESP32 with an NTP server accessible on the network.

To synchronize time with an NTP server, add a property list with the tag `sntp` at the top level configuration passed into the `network:start/1` function. Specify the NTP hostname or IP address with which your device should sync using the `host` property tag. The `host` value can be a string or binary.

You can also specify a callback function that will get called when the clock is synchronized with the SNTP server via the `synchronized` property tag. This function takes a tuple with the updated time in seconds and microseconds.

For example:

```
{sntp, [
    {host, <<"pool.ntp.org">>},
    {synchronized, fun sntp_synchronized/1}
]}
```

where the `sntp_synchronized/1` function is defined as:

```
sntp_synchronized({TVSec, TVUsec}) ->
    io:format("Synchronized time with SNTP server. TVSec=~p TVUsec=~p~n", [TVSec,
 TVUsec]).
```

Note The device must be in STA mode and connected to an access point in order to use an SNTP server on your network or on the internet.

6.5 NVS Credentials

It can become tiresome to enter an SSID and password for every application, and in general it is bad security practice to hard-wire WiFi credentials in your application source code.

You may instead store an STA or AP SSID and PSK in non-volatile storage (NVS) on and ESP32 device.

Caution! Credentials are stored un-encrypted and in plaintext and should not be considered secure. Future versions may use encrypted NVS storage.

6.6 Stopping the Network

To stop the network and free any resources in use, issue the `stop/0` function:

```
network:stop().
```

Caution! Stop is currently unimplemented.

Distributed Erlang

For a generic introduction to Distributed Erlang Systems, please refer to the dedicated section of Erlang/OTP documentation.

AtomVM provides an implementation of Erlang distribution protocol and AtomVM nodes can take part in clusters with both AtomVM and BEAM nodes.

Distribution is currently available on all platforms with TCP/IP communication, namely:

- Generic Unix
- ESP32
- RP2 (Pico)

Two examples are provided:

- disterl in `examples/erlang/disterl.erl`: distribution on Unix systems
- epmd_disterl in `examples/esp32/epmd_disterl.erl`: distribution on ESP32 devices

7.1 Starting and stopping distribution

Distribution has to be started programmatically. Following Erlang/OTP, distribution relies on `kernel` which needs to be started.

The following lines will start distribution on Unix systems with long name `atomvm@127.0.0.1`.

```
{ok, _KernelPid} = kernel:start(normal, []),
{ok, _NetKernelPid} = net_kernel:start('atomvm@127.0.0.1', #{name_domain =>
 longnames}),
ok = net_kernel:set_cookie(<<"AtomVM">>).
```

`net_kernel:stop/0` can be used to stop distribution.

7.2 epmd

AtomVM nodes can use Erlang/OTP's epmd on Unix systems. AtomVM is also bundled with a pure Erlang implementation of `epmd` which can be used on all platforms. Module is called `epmd`, to be distinguished from `erl_epmd` which is the client.

AtomVM's epmd daemon can be started with:

```
{ok, _EPMDPid} = epmd:start_link([]).
```

This has to be called before invoking `net_kernel:start/2`.

7.3 Erlang/OTP compatibility

AtomVM can connect to Erlang/OTP 24 and higher.

7.4 Security

AtomVM supports cookie authentication. However, distribution over TLS is not supported yet.

7.5 Alternative carrier

Following Erlang/OTP, AtomVM supports alternative carriers with distribution modules. Please refer to Erlang/OTP's dedicated documentation.

The main difference is that packets exchanged by `f_recv` and `f_send` handlers must be binaries instead of list of integers, for memory usage reasons.

AtomVM's `f_send` has the following signature:

```
fun (DistCtrlr, Data :: binary()) -> ok | {error, Error}
```

AtomVM's `f_recv` has the following signature:

```
fun (DistCtrlr, Length :: pos_integer(), Timeout :: timeout()) -> {ok, Packet} |
 {error, Reason}
```

AtomVM's distribution is based on `socket_dist` and `socket_dist_controller` modules which can also be used with BEAM by definining `BEAM_INTERFACE` to adjust for the difference.

7.6 Distribution features

Distribution implementation is (very) partial. The most basic features are available:

- serialization of all types
- epmd protocol (client and server)
- message passing
- monitoring processes
- I/O distribution ("group leader").

RPC (remote procedure call) from Erlang/OTP to AtomVM is also supported. Shell is not supported yet.

Please do not hesitate to file issues or pull requests for additional features.

Chapter 8

Build Instructions

This guide is intended for anyone interested in building the AtomVM virtual machine from source code. You may be interested in building the AtomVM source code if you want to provide bug fixes or enhancements to the VM, or if you want to simply learn more about the platform. In addition, some "downstream" drivers for specific devices may need to be built specifically for the target platform (e.g., ESP32), in which case building the VM from source code is required.

Tip Many applications do not require building the AtomVM runtime from source code. Instead, you can download pre-built VM images for platforms such as ESP32, and use Erlang and Elixir tooling to build and deploy your applications.

The AtomVM virtual machine itself, including the runtime code execution engine, as well as built-in functions and Nifs is implemented in C. The core standard and AtomVM libraries are implemented in Erlang and Elixir.

The native C parts of AtomVM compile to machine code on MacOS, Linux, and FreeBSD platforms. The C code also compiles to run on the ESP32 and STM32 platforms. Typically, binaries for these platforms are created on a UNIX-like environment (MacOS or Linux, currently) using tool-chains provided by device vendors to cross-compile and target specific device architectures.

The Erlang and Elixir parts are compiled to BEAM byte-code using the Erlang (`erlc`) and Elixir compilers. For information about specific versions of required software, see the Release Notes.

This guide provides information about how to build AtomVM for the various supported platforms (Generic UNIX, ESP32, and STM32).

Attention! In order to build AtomVM AVM files for ESP32 and STM32 platforms, you will also need to build AtomVM for the Generic UNIX platform of your choice.

8.1 Downloading AtomVM

The AtomVM source code is available by cloning the AtomVM github repository:

```
$ git clone https://github.com/atomvm/AtomVM
```

See also

Downloading the AtomVM github repository requires the installation of the `git` program. Consult your local OS documentation for installation of the `git` package.

If you want to build a release version of AtomVM, simply checkout the desired release:

```
$ git checkout v0.6.0-alpha.2
```

Tip You may need to refresh the tags if you have already cloned the repository and you want to build a more recent release version.

```
$ git pull --tags --rebase
```

The use of `--rebase` is necessary if you are in a working branch and have made commits, otherwise it is optional.

To return to the current `master` branch use `git switch master`.

8.2 Source code organization

Source code is organized as follows:

- `src` Contains the core AtomVM virtual machine source code;
- `lib` Contains the Erlang and Elixir core library source code;
- `tools` Contains AtomVM tooling, including the `PackBEAM` executable, as well as build support tooling;
- `examples` Contains sample programs for demonstration purposes;
- `tests` Contains test code run as part of test qualification;
- `doc` Contains documentation source code and content.

The `src` directory is broken up into the core platform-independent AtomVM library (`libAtomVM`), and platform-dependent code for each of the supported platforms (Generic UNIX, ESP32, and STM32).

8.3 External dependencies

8.3.1 `uf2tool`

AtomVM depends on `uf2tool`. It is used to pack both native and Erlang/Elixir/Gleam code for RP2. uf2tool is downloaded automatically by `rebar3` from `hex` mirrors.

It is possible to use a local copy of uf2tool by setting `UF2TOOL_PATH` variable to a path to a source checkout of `uf2tool` when invoking CMake.

8.4 Platform Specific Build Instructions

- Generic UNIX
- ESP32
- STM32
- RP2 (including Pico boards)
- WASM (NodeJS or web)

8.5 Building for Generic UNIX

The following instructions apply to unix-like environments, including Linux, FreeBSD, DragonFly and

MacOS.

Hint The Generic UNIX is useful for running and testing simple AtomVM programs. Not all of the AtomVM APIs, specifically, APIs that are dependent on various device integration, are supported on this platform.

8.5.1 Generic UNIX Build Requirements

The following software is required in order to build AtomVM in generic UNIX systems:

- `gcc` or `llvm` tool chains
- `cmake`
- `make`
- `gperf`
- `zlib`
- `Mbed TLS`
- Erlang/OTP compiler (`erlc`)
- Elixir compiler

Consult Release Notes for currently supported versions of required software.

Consult your local OS documentation for instructions about how to install these components.

8.5.2 Generic UNIX Build Instructions

The AtomVM build for generic UNIX systems makes use of the `cmake` tool for generating `make` files from the top level AtomVM directory. With CMake, you generally create a separate directory for all output files (make files, generated object files, linked binaries, etc). A common pattern is to create a local `build` directory, and then point `cmake` to the parent directory for the root of the source tree:

```
$ mkdir build
$ cd build
$ cmake ..
```

This command will create all of the required make files for creating the AtomVM binary, tooling, and core libraries. You can create all of these object using the `make` command:

```
$ make -j 8
```

Tip You may specify `-j <n>`, where `<n>` is the number of CPUs you would like to assign to run the build in parallel.

Upon completion, the `AtomVM` executable can be found in the `build/src` directory.

The AtomVM core Erlang library can be found in the generated `libs/atomvmlib.avm` AVM file.

Use the `install` target to install the `atomvm` command and associated binary files. On most UNIX systems, these artifacts will be installed in the `/usr/local` directory tree.

Attention! On some systems, you may need to run this target with `root` or `sudo` permissions.

```
$ sudo make install
```

Once installed, you can use the `atomvm` command to execute an AtomVM application. E.g.,

```
$ atomvm /path/to/myapp.avm
```

For users doing incremental development on the AtomVM virtual machine, you may want to run the `AtomVM` binary directly instead of installing the VM on your machine. If you do, you will typically need to also specify the path to the AtomVM core Erlang library. For example,

```
$ cd build
$ ./src/AtomVM /path/to/myapp.avm ./libs/atomvmlib.avm
```

Special Note for MacOS users

You may build an Apple Xcode project, for developing, testing, and debugging in the Xcode IDE, by specifying the Xcode generator. For example, from the top level AtomVM directory:

```
$ mkdir xcode
$ cmake -G Xcode ..
...
$ open AtomVM.xcodeproj
```

The above commands will build and open an AtomVM project in the Xcode IDE.

8.5.3 Running tests

There are currently two sets of suites of tests for AtomVM:

- Erlang tests (`erlang_tests`) A set of unit tests for basic Erlang functionality, exercising support BEAM opcodes, built-in functions (Bifs) and native functions (Nifs).
- Library tests, exercising functionality in the core Erlang and Elixir libraries.

To run the Erlang tests, run the `test-erlang` executable in the `tests` directory:

```
$ ./tests/test-erlang
```

This will run a suite of several score unit tests. Check the status of the executable after running the tests. A non-zero return value indicates a test failure.

To run the Library tests, run the corresponding AVM module in the `tests/libs` directory using the `AtomVM` executable. For example:

```
$ ./src/AtomVM ./tests/libs/estdlib/test_estdlib.avm
```

This will run a suite of several unit tests for the specified library. Check the status of the executable after running the tests. A non-zero return value indicates a test failure.

Tests for the following libraries are supported:

- `estdlib`
- `eavmlib`
- `alisp`

8.6 Building for ESP32

Building AtomVM for ESP32 must be done on either a Linux or MacOS build machine.

In order to build a complete AtomVM image for ESP32, you will also need to build AtomVM for the Generic UNIX platform (typically, the same build machine you are suing to build AtomVM for ESP32).

8.6.1 ESP32 Build Requirements

The following software is required in order to build AtomVM for the ESP32 platform:

- Espressif Xtensa tool chains
- Espressif IDF SDK (consult Release Notes for currently supported versions)
- `cmake`

Instructions for downloading and installing the Espressif IDF SDK and tool chains are outside of the scope of this document. Please consult the IDF SDKGetting Started guide for more information.

8.6.2 ESP32 Build Instructions

To activate the ESP-IDF build environment change directories to the tree root of your local ESP-IDF:

```
$ cd <ESP-IDF-ROOT-DIR>
$ . ./export.sh
```

Hint If you followed Espressif's installation guide the ESP-IDF directory is `${HOME}/esp/esp-idf`

Change directories to the `src/platforms/esp32` directory under the AtomVM source tree root:

```
$ cd <atomvm-source-tree-root>
$ cd src/platforms/esp32
```

If you want to build an image with Elixir modules included you must first have a version of Elixir installed that is compatible with your OTP version, then add the following line to sdkconfig.defaults:

```
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions-elixir.csv"
```

Start by updating the default build configuration of local `sdkconfig` file via the `idf.py reconfigure` command:

```
$ idf.py set-target esp32
$ idf.py reconfigure
```

Tip For those familiar with esp-idf the build can be customized using `menuconfig` instead of `reconfigure`:

```
$ idf.py menuconfig
```

This command will bring up a curses dialog box where you can make adjustments such as not including AtomVM components that are not desired in a particular build. You can also change the behavior of a crash in the VM to print the error and reboot, or halt after the error is printed. Extreme caution should be used when changing any non AtomVM settings. You can quit the program by typing `Q`. Save the changes, and the program will exit.

You can now build AtomVM using the build command:

```
$ idf.py build
```

This command, once completed, will create the Espressif bootloader, partition table, and AtomVM binary. The last line of the output should read something like the following:

```
Project build complete. To flash, run this command:
~/.espressif/python_env/idf5.1_py3.11_env/bin/python ~/esp/esp-idf-v5.1/components
/esptool_py/esptool/esptool.py -p (PORT) -b 921600 --before default_reset
--after hard_reset --chip esp32 write_flash --flash_mode dio --flash_size detect
--flash_freq 40m 0x1000 build/bootloader/bootloader.bin 0x8000
build/partition_table/partition-table.bin 0x10000 build/atomvm-esp32.bin
or run 'idf.py -p (PORT) flash'
```

At this point, you can run `idf.py flash` to upload the 3 binaries up to your ESP32 device, and in some development scenarios, this is a preferable shortcut.

However, first, we will build a single binary image file containing all of the above 3 binaries, as well as the AtomVM core libraries. See Building a Release Image, below. But first, it is helpful to understand a bit about how the AtomVM partitioning scheme works, on the ESP32.

8.6.3 Running tests for ESP32

Tests for ESP32 are run on the desktop (or CI) using qemu.

Install or compile Espressif's fork of qemu. Espressif provides binaries for Linux amd64 and it's also bundled in espressif/idf:5.1 docker image.

Also install Espressif pytest's extensions for embedded testing with:

```
$ cd <ESP-IDF-ROOT-DIR>
$ . ./export.sh
$ pip install pytest==7.0.1 \
    pytest-embedded==1.2.5 \
    pytest-embedded-serial-esp==1.2.5 \
    pytest-embedded-idf==1.2.5 \
    pytest-embedded-qemu==1.2.5
```

Change directory to the `src/platforms/esp32/test` directory under the AtomVM source tree root:

```
$ cd <atomvm-source-tree-root>
$ cd src/platforms/esp32/test
```

Build tests using the build command:

```
$ idf.py build
```

Note This eventually compiles host AtomVM to be able to build and pack erlang test modules.

Run tests using the command:

```
$ pytest --embedded-services=idf,qemu -s
```

ESP32 tests are erlang modules located in `src/platforms/esp32/test/main/test_erl_sources/` and executed from `src/platforms/esp32/test/main/test_main.c`.

8.6.4 Performance and Power

AtomVM comes with conservative defaults for broad compatibility with different ESP32 boards, and a reasonable performance/power/longevity tradeoff.

You may want to change these settings to optimize for your specific application's performance and power needs.

Factors like heat dissipation should also be considered, and the effect on overall longevity of components.

CPU frequency

Use `idf.py menuconfig` in `src/platforms/esp32` `Component config ---> ESP System Settings ---> CPU frequency (160 MHz) --->`

You can increase or decrease the CPU frequency, this is a tradeoff against power usage. Eg. 160 MHz is the conservative default for the ESP32, but you can increase it to 240 MHz or decrease it to 80 MHz. The higher the frequency, the more power is consumed. The lower the frequency, the less power is consumed.

Flash mode and speed

Use `idf.py menuconfig` in `src/platforms/esp32` `Serial flasher config ---> Flash`

`SPI mode (DIO) --->` You can change the mode of the SPI flash. QIO is the fastest mode, but not all flash chips support it.

`Serial flasher config ---> Flash SPI speed (40 MHz) --->` You can change the speed of the SPI flash. The higher the speed, the faster the flash will be, at the cost of higher power usage, but not all flash chips support higher speeds.

See external docs: ESP-IDF flash modes

PSRAM speed

Use `idf.py menuconfig` in `src/platforms/esp32` `Component config ---> ESP PSRAM ---> SPI RAM config --->`

If your board has PSRAM and it's enabled, you can configure the SPI RAM settings here. `Set RAM clock speed (40MHz clock speed) --->` You can increase or decrease the clock speed of the PSRAM.

Warning You may have to increase "Flash SPI speed" (see above) before you can increase PSRAM speed.

The higher the speed, the faster the PSRAM will be, at the cost of higher power usage, but not all PSRAM chips support higher speeds.

Sleep mode - Deep sleep

For low power applications, you should use the deep sleep functionality of the ESP32.

This will put the ESP32 into a very low power state, and it will consume very little power. You can wake the ESP32 from deep sleep using a timer, or an interrupt etc.

Make sure your board is suitable for low power deep sleep, some boards have voltage regulators and /or LEDs constantly draining power, also make sure sensors are powered down or in low power mode when the ESP32 is in deep sleep.

For persisting small amounts of data during deep sleep, you can use the RTC memory of the ESP32, which is preserved during deep sleep.

Sleep mode - Light sleep

Usage of light sleep is untested, and no support for controlling light sleep is currently implemented. Reach out if you do any experiments and measurements.

Other Build settings

Various other build settings can be changed in `idf.py menuconfig` in `src/platforms/esp32`, that affect the performance and power usage of the ESP32.

Of note AtomVM releases are built with the `Optimize for performance (-O2)` compiler option: Use `idf.py menuconfig` in `src/platforms/esp32` `Compiler options ---> Optimization Level () --->`

See CONFIG_COMPILER_OPTIMIZATION_PERF

And all builds are built with the mbedTLS `Enable fixed-point multiplication optimisations` option: Use `idf.py menuconfig` in `src/platforms/esp32` `Component config ---> mbedTLS ---> Enable fixed-point multiplication optimisations`

See MBEDTLS_ECP_FIXED_POINT_OPTIM

8.6.5 Flash Layout

The AtomVM Flash memory is partitioned to include areas for the above binary artifacts created from the build, as well areas for runtime information used by the ESP32 and compiled Erlang/Elixir code.

The flash layout is roughly as follows (not to scale):

```
+-----------------+  ------------- 0x0 | 0x1000 | 0x2000
|                 |              ^
|   boot loader   | 28KB         |
|                 |              |
+-----------------+              |
| partition table | 3KB          |
+-----------------+              |
|                 |              |
|       NVS       | 24KB         |
|                 |              |
+-----------------+              |
|     PHY_INIT    | 4KB          |
+-----------------+              | AtomVM
|                 |              | binary
|                 |              | image
|                 |              |
|     AtomVM      |              |
|     Virtual     | 1.75MB       |
|     Machine     |              |
|                 |              |
|                 |              |
+-----------------+              |
|     boot.avm    | 256-512KB    v
+-----------------+  ------------- 0x210000 for Erlang only images or
|                 |              ^  0x250000 for images with Elixir modules
|                 |              |
|     main.avm    | 1MB+         | Erlang/Elixir
|                 |              | Application
|                 |              |
|                 |              v
+-----------------+  ------------- end
```

The following table summarizes the partitions created on the ESP32 when deploying AtomVM:

Partition	Offset	Length	Description
Bootloader	0x0 | 0x1000 | 0x2000	28kB	The ESP32 bootloader, as built from the IDF-SDK. AtomVM does not define its own bootloader. The offset of the bootloader varies by chip.
Partition Table	0x8000	3kB	The AtomVM-defined partition table.
NVS	0x9000	24kB	Space for non-volatile storage.
PHY_INIT	0xF000	4kB	Initialization data for physical layer radio signal data.
AtomVM virtual machine	0x10000	1.75mB	The AtomVM virtual machine (compiled from C code).
boot.avm	0x1D0000	256k	The AtomVM BEAM library, compiled from Erlang and Elixir files in the AtomVM source tree.
main.avm	`0x210000` | `0x250000`	1mB	The user application. This is where users flash their compiled Erlang/Elixir code

Warning There is an important difference in the partition layout between the minimal images and those build with Elixir support. To accommodate the extra Elixir modules the boot.avm partition on these images is larger, and the application offset is moved accordingly. When working with Elixir supported images it is important to always use the offset `0x250000` whether using `mix` or the `atomvm_rebar3_plugin` (possibly to test an Erlang app), otherwise part of the boot.avm partition (specifically the area where many Elixir modules are located) will be overwritten with the application, but the VM will still be trying to load from the later `0x250000` offset. This should be kept in mind reading the rest of build instructions, and AtomVM Tooling sections of the docs that cover the use of rebar3, for these sections an Erlang only image is assumed.

8.6.6 The `boot.avm` and `main.avm` partitions

The `boot.avm` and `main.avm` partitions are intended to store Erlang/Elixir libraries (compiled down to BEAM files, and assembled as AVM files).

The `boot.avm` partition is intended for core Erlang/Elixir libraries that are built as part of the AtomVM build. The release image of AtomVM (see below) includes both the AtomVM virtual machine and the `boot.avm` partition, which includes the BEAM files from the `estdlib` and `eavmlib` libraries.

In contrast, the `main.avm` partition is intended for user applications. Currently, the `main.avm` partition starts at address `0x210000` for thin images or `0x250000` for images with Elixir modules, and it is to that location to which application developers should flash their application AVM files.

The AtomVM search path for BEAM modules starts in the `main.avm` partition and falls back to `boot.avm`. Users should not have a need to override any functionality in the `boot.avm` partition, but if necessary, a BEAM module of the same name in the `main.avm` partition will be loaded instead of the version in the `boot.avm` partition.

Warning The location of the `main.avm` partition may change over time, depending on the relative sizes of the AtomVM binary and `boot.avm` partitions.

8.6.7 Building a Release Image

The `<atomvm-source-tree-root>/tools/release/esp32` directory contains the `mkimage.sh` script that can be used to create a single AtomVM image file, which can be distributed as a release, allowing application developers to develop AtomVM applications without having to build AtomVM from scratch.

Attention! Before running the `mkimage.sh` script, you must have a complete build of both the esp32 project, as well as a full build of the core Erlang libraries in the `libs` directory. The script configuration defaults to assuming that the core Erlang libraries will be written to the `build/libs` directory in the AtomVM source tree. You should pass the `--build_dir <path>` option to the `mkimage.sh` script, with `<path>` pointing to your AtomVM build directory, if you target a different build directory when running CMake.

Running this script will generate a single `atomvm-<sha>.img` file in the `build` directory of the esp32 source tree, where `<sha>` is the git hash of the current checkout. This image contains the ESP32 bootloader, AtomVM executable, and the `eavmlib` and `estdlib` Erlang libraries in one file, which can then be flashed to address `0x1000` for the esp32. The bootloader address varies for other chip variants. See the flashing a binary image to ESP32 section of the Getting Started Guide for a chart with the bootloader offset address of each model.

To build a thin image with only Erlang libraries `mkimage.sh` script is run from the `src/platform/esp32` directory as follows:

```
$ ./build/mkimage.sh
Writing output to /home/joe/AtomVM/src/platforms/esp32/build/atomvm-esp32.img
=============================================
Wrote bootloader at offset 0x1000 (4096)
Wrote partition-table at offset 0x8000 (32768)
Wrote AtomVM Virtual Machine at offset 0x10000 (65536)
Wrote AtomVM Core BEAM Library at offset 0x1D0000 (1114112)
```

To build a full image with Erlang and Elixir libraries the path to the previously (during the generic_unix build) built `elixir_esp32boot.avm` must be passed to the `mkimage.sh` script as follows (Note: this is still run from the AtomVM/src/platforms/esp32 directory for the relative path to work - feel free to use the absolute path to this file):

```
$ ./build/mkimage.sh --boot ../../../build/libs/esp32boot/elixir_esp32boot.avm
Writing output to /home/joe/AtomVM/src/platforms/esp32/build/atomvm-esp32.img
=============================================
Wrote bootloader at offset 0x1000 (4096)
Wrote partition-table at offset 0x8000 (32768)
Wrote AtomVM Virtual Machine at offset 0x10000 (65536)
Wrote AtomVM Core BEAM Library at offset 0x1D0000 (1114112)
```

Users can then use the `esptool.py` directly to flash the entire image to the ESP32 device, and then flash their applications to the `main.app` partition at address `0x210000`, (or `0x250000` for Elixir images)

But first, it is a good idea to erase the flash, e.g.,

```
$ esptool.py --chip esp32 --port /dev/ttyUSB0 erase_flash
esptool.py v2.1
Connecting........_
Chip is ESP32D0WDQ6 (revision 1)
Uploading stub...
Running stub...
Stub running...
Erasing flash (this may take a while)...
Chip erase completed successfully in 5.4s
Hard resetting...
```

Flashing Release Images

After preparing a release image you can use the `flashimage.sh`, which is generated with each build that will flash the full image using the correct flash offset for the chip the build was configured for using the either the default Erlang only `partitions.cvs` table, or the `partitions-elixir.cvs` table if that was used during the configuration.

```
$ ./build/flashimage.sh
```

To perform this action manually you can use the `./build/flash.sh` tool (or `esptool.py` directly, if you prefer):

```
$ FLASH_OFFSET=0x1000 ./build/flash.sh ./build/atomvm-esp32-0.6.6.img
esptool.py v2.8-dev
Serial port /dev/tty.SLAB_USBtoUART
Connecting........_
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, Coding Scheme None
Crystal is 40MHz
MAC: 30:ae:a4:1a:37:d8
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 921600
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Wrote 1163264 bytes at 0x00001000 in 15.4 seconds (603.1 kbit/s)...
Hash of data verified.
Leaving...
Hard resetting via RTS pin...
```

Caution! Flashing the full AtomVM image will delete all entries in non-volatile storage. Only flash the full image if you have a way to recover and re-write any such data, if you need to retain it.

8.6.8 Flashing Applications

Applications can be flashed using the `flash.sh` script in the esp32 build directory (the application offset is set correctly depending on the build configuration):

```
$ ./build/flash.sh ../../../build/examples/erlang/esp32/blink.avm
%%
%% Flashing examples/erlang/esp32/blink.avm (size=4k)
%%
esptool.py v2.8-dev
Serial port /dev/tty.SLAB_USBtoUART
Connecting........_
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, Coding Scheme None
Crystal is 40MHz
MAC: 30:ae:a4:1a:37:d8
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 921600
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Wrote 16384 bytes at 0x00210000 in 0.2 seconds (611.7 kbit/s)...
Hash of data verified.
Leaving...
Hard resetting via RTS pin...
```

Tip Since the Erlang core libraries are flashed to the ESP32 device, it is not necessary to include core libraries in your application AVM files. Users may be interested in using downstream development tools, such as the Elixir ExAtomVM Mix task, or the Erlang AtomVM Rebar3 Plugin for doing day-to-day development of applications for the AtomVM platform.

Flashing the core libraries

If you are doing development work on the core Erlang/Elixir libraries and wish to test changes that do not involve the `C` code in the core VM you may flash `esp32boot.avm` (or `elixir_esp32boot.avm` when using an Elixir partition table) to the boot.avm partition (offset 0x1D0000) by using the `flash.sh` script in the esp32 build directory as follows:

```
$ build/flash.sh -l ../../../build/libs/esp32boot.avm
%%
%% Flashing ../../../build/libs/esp32boot.avm (size=116k)
%%
esptool.py v4.5.1
Serial port /dev/ttyUSB0
Connecting.....
Detecting chip type... Unsupported detection protocol, switching and trying
again...
Connecting.....
Detecting chip type... ESP32
Chip is ESP32-D0WD (revision v1.0)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme
None
Crystal is 40MHz
MAC: 1a:57:c5:7f:ac:5b
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 921600
Changed.
```

```
Configuring flash size...
Auto-detected Flash size: 8MB
Flash will be erased from 0x001d0000 to 0x001ecfff...
Wrote 131072 bytes at 0x001d0000 in 1.8 seconds (582.1 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

8.6.9 Adding custom Nifs, Ports, and third-party components

While AtomVM is a functional implementation of the Erlang virtual machine, it is nonetheless designed to allow developers to extend the VM to support additional integrations with peripherals and protocols that are not otherwise supported in the core virtual machine.

AtomVM supports extensions to the VM via the implementation of custom native functions (Nifs) and processes (AtomVM Ports), allowing users to extend the VM for additional functionality, and you can add your own custom Nifs, ports, and additional third-party components to your ESP32 build by adding them to the `components` directory, and the ESP32 build will compile them automatically.

See also

For more information about building components for the IDF SDK, consult the IDF SDK Build System documentation.

The instructions for adding custom Nifs and ports differ in slight detail, but are otherwise quite similar. In general, they involve:

1. Adding the custom Nif or Port to the `components` directory of the AtomVM source tree.
2. Run `idf.py reconfigure` to pick up any menuconfig options, many extra drivers have an option to disable them (they are enabled by default). Optionally use `idf.py menuconfig` and confirm the driver is enabled and save when quitting.
3. Building the AtomVM binary.

Attention! The Espressif SDK and tool chains do not, unfortunately, support dynamic loading of shared libraries and dynamic symbol lookup. In fact, dynamic libraries are not supported at all on the ESP32 using the IDF SDK; instead, any code that is needed at runtime must be statically linked into the application.

Custom Nifs and Ports are available through third parties. Follow the instructions provided with these custom components for detailed instruction for how to add the Nif or Port to your build.

More detailed instructions follow, below, for implementing your own Nif or Port.

Adding a custom AtomVM Nif

To add support for a new peripheral or protocol using custom AtomVM Nif, you need to do the following:

- Choose a name for your nif (e.g, "my_nif"). Call this `<moniker>`.
- In your source code, implement the following two functions:
  - `void <moniker>_nif_init(GlobalContext *global);`
    - This function will be called once, when the application is started.
  - `const struct Nif *<moniker>_nif_get_nif(const char *nifname);`
    - This function will be called to locate the Nif during a function call. Example:

```
void my_nif_init(GlobalContext *global);
const struct Nif *my_nif_get_nif(const char *nifname);
```

Note Instructions for implementing Nifs is outside of the scope of this document.

- Add the `REGISTER_NIF_COLLECTION` using the parameters `NAME`, `INIT_CB`, `DESTROY_CB`, `RESOLVE_NIF_CB` macro to the end of your nif code. Example:

```
REGISTER_NIF_COLLECTION(my_nif, NULL, NULL, my_nif_get_nif);
```

Adding a custom AtomVM Port

To add support for a new peripheral or protocol using an AtomVM port, you need to do the following:

- Choose a name for your port (e.g, "my_port"). Call this `<moniker>`.
- In your source code, implement the following two functions:
  - `void <moniker>_init(GlobalContext *global);`
    - This function will be called once, when the application is started.
  - `Context *<moniker>_create_port(GlobalContext *global, term opts);`
    - This function will be called to locate the Nif during a function call. Example:

```
void my_port_init(GlobalContext *global);
Context *my_port_create_port(GlobalContext *global, term opts);
```

Note Instructions for implementing Ports is outside of the scope of this document.

- Add the `REGISTER_PORT_COLLECTION` using the parameters `NAME`, `INIT_CB`, `DESTROY_CB`, `RESOLVE_NIF_CB` macro to the end of your nif code. Example:

```
REGISTER_PORT_COLLECTION(my_port, my_port_init, NULL,
my_port_create_port);
```

8.7 Building for STM32

8.7.1 STM32 Prerequisites

The following software is required to build AtomVM for the STM32 platform:

- 11.3 ARM toolchain (or compatible with your system)
- libopencm3 version 0.8.0
- `cmake`
- `make`
- `git`
- `python`
- Erlang/OTP `escript`

Note AtomVM tests this build on the latest Ubuntu github runner.

8.7.2 Setup libopencm3

Before building for the first time you need to have a compiled clone of the libopencm3 libraries, from inside the AtomVM/src/platforms/stm32 directory:

```
$ git clone -b v0.8.0 https://github.com/libopencm3/libopencm3.git
$ cd libopencm3 && make -j4 && cd ..
```

Tip You can put libopencm3 wherever you want on your PC as long as you update LIBOPENCM3_DIR to point to it. This example assumes it has been cloned into /opt/libopencm3 and built. From inside the AtomVM/src/platforms/stm32 directory:

```
$ cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/arm-toolchain.cmake \
-DLIBOPENCM3_DIR=/opt/libopencm3 ..
```

8.7.3 Build AtomVM with cmake toolchain file

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/arm-toolchain.cmake ..
$ make
```

8.7.4 Changing the target device

The default build is based on the STM32F4Discovery board chip (`stm32f407vgt6`). If you want to target a different chip, pass the `-DDEVICE` flag when invoking cmake. For example, to use the BlackPill V2.0, pass `-DDEVICE=stm32f411ceu6`. At this time any `STM32F4` or `STM32F7` device with 512KB or more of on package flash should work with AtomVM. If an unsupported device is passed with the `DEVICE` parameter the configuration will fail. For devices with either 512KB or 768KB of flash the available application flash space will be limited to 128KB. Devices with only 512KB of flash may also suffer from slightly reduced performance because the compiler must optimize for size rather than performance.

Attention! For devices with only 512KB of flash the application address is different and must be adjusted when flashing your application with st-flash, or using the recommended `atomvm_rebar3_plugin`. The application address for these devices is `0x8060000`.

8.7.5 Configuring the Console

The default build for any `DEVICE` will use `USART2` and output will be on `PA2`. This default will work well for most `Discovery` and generic boards that do not have an on-board TTL to USB-COM support (including the `stm32f411ceu6` A.K.A. `BlackPill V2.0`). For `Nucleo` boards that do have on board UART to USB-COM support you may pass the `cmake` parameter `-DBOARD=nucleo` to have the correct USART and TX pins configured automatically. The `Nucleo-144` series use `USART3` and `PD8`, while the supported `Nucleo-64` boards use `USART2`, but passing the `BOARD` parameter along with `DEVICE` will configure the correct `USART` for your model. If any other boards are discovered to have on board USB UART support pull requests, or opening issues with the details, are more than welcome.

Example to configure a `NUCLEO-F429ZI`:

```
$ cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/arm-toolchain.cmake -DDEVICE=stm32f429zit6
 \
-DBOARD=nucleo
```

The AtomVM system console `USART` may also be configured to a specific uart peripheral. Pass one of the parameters from the chart below with the `cmake` option `-DAVM_CFG_CONSOLE=CONSOLE_#`, using the desired console parameter in place of `CONSOLE_#`. Not all UARTs are available on every supported board, but most will have several options that are not already used by other on board peripherals. Consult your data sheets for your device to select an appropriate console.

Parameter	USART	TX Pin	AtomVM Default	Nucleo-144	Nucleo-64
CONSOLE_1	USART1	PA9			
CONSOLE_2	USART2	PA2	?		?
CONSOLE_3	USART3	PD8		?	
CONSOLE_4	UART4	PC10			
CONSOLE_5	UART5	PC12			
CONSOLE_6	USART6	PC6			
CONSOLE_7	UART7	PF7			
CONSOLE_8	UART8	PJ8			

8.7.6 Configure STM32 logging with `cmake`

The default maximum log level is `LOG_INFO`. To change the maximum level displayed pass `-DAVM_LOG_LEVEL_MAX="{level}"` to `cmake`, with one of `LOG_ERROR`, `LOG_WARN`, `LOG_INFO`, or `LOG_DEBUG` (listed from least to most verbose). Log messages can be completely disabled by using `-DAVM_LOG_DISABLE=on`.

For log entries colorized by log level pass `-DAVM_ENABLE_LOG_COLOR=on` to cmake. With color enable there is a very small performance penalty (~1ms per message printed), the log entries are colored as follows:

Message Level	Color
ERROR	Red
WARN	Orange
INFO	Green
DEBUG	Blue

By default only `ERROR` messages contain file and line number information. This can be included with all log entries by passing `-DAVM_ENABLE_LOG_LINES=on` to cmake, but it does incur a significant performance penalty and is only suggested for debugging during development.

8.7.7 Console Printing on STM32

AtomVM is built with standard `newlib` to support `long long` integers (`signed` and `unsigned`). If you are building for a device with extremely limited flash space the `nano` version of `newlib` can be used instead. This may be done by passing `-DAVM_NEWLIB_NANO=on`. If the `nano newlib` is used logs will be automatically disabled, this is because many of the VM low level log messages will include `%ull` formatting and will cause buffer overflows and crash the VM if logging is not disabled for `nano newlib` builds. The total flash savings of using `nano newlib` and disabling logs is just under 40kB.

By default, stdout and stderr are printed on USART2. On the STM32F4Discovery board, you can see them using a TTL-USB with the TX pin connected to board's pin PA2 (USART2 RX). Baudrate is 115200 and serial transmission is 8N1 with no flow control.

See also

If building for a different target USART may be configure as explained above in Configuring the Console.

8.7.8 Configuring deployment builds for STM32

After your application has been tested (*and debugged*) and is ready to put into active use you may want to tune the build of AtomVM. For instance disabling logging with `-DAVM_LOG_DISABLE=on` as a `cmake` configuration option may result in slightly better performance. This will have no affect on the console output of your application, just disable low level log messages from the AtomVM system. You may also want to enabling automatic reboot in the case that your application ever exits with a return other than `ok`. This can be enabled with the `cmake` option `-DAVM_CONFIG_REBOOT_ON_NOT_OK=on`.

8.8 Building for Raspberry Pi RP2

You can build with all boards supported by Raspberry Pi pico SDK, including Pico, Pico W, Pico2 and Pico 2 W. AtomVM also works with clones such as RP2040 Zero.

8.8.1 RP2 Prerequisites

- `cmake`
- `ninja`
- `Erlang/OTP`
- `Elixir` (optional)
- A toolchain for the target (ARM or Risc-V)

8.8.2 AtomVM build steps (Pico or most boards based on RP2040)

```
$ cd src/platforms/rp2/
$ mkdir build
$ cd build
$ cmake .. -G Ninja
$ ninja
```

Tip You may want to build with option `AVM_REBOOT_ON_NOT_OK` so AtomVM restarts on error.

8.8.3 AtomVM build steps (Pico W)

```
$ cd src/platforms/rp2/
$ mkdir build
$ cd build
$ cmake .. -G Ninja -DPICO_BOARD=pico_w
$ ninja
```

Tip You may want to build with option `AVM_REBOOT_ON_NOT_OK` so AtomVM restarts on error.

8.8.4 AtomVM build steps (Pico 2 or boards based on RP2350)

For ARM S platform (recommended) :

```
$ cd src/platforms/rp2/
$ mkdir build
$ cd build
$ cmake .. -G Ninja -DPICO_BOARD=pico2
$ ninja
```

For RISC-V platform (supported but slower) :

```
$ cd src/platforms/rp2/
$ mkdir build
$ cd build
$ cmake .. -G Ninja -DPICO_BOARD=pico2 -DPICO_PLATFORM=rp2350-riscv
$ ninja
```

Tip You may want to build with option `AVM_REBOOT_ON_NOT_OK` so AtomVM restarts on error.

8.8.5 AtomVM build steps (Pico 2 W)

For ARM S platform (recommended) :

```
$ cd src/platforms/rp2/
$ mkdir build
$ cd build
$ cmake .. -G Ninja -DPICO_BOARD=pico2_w
$ ninja
```

For RISC-V platform (supported but slower) :

```
$ cd src/platforms/rp2/
$ mkdir build
$ cd build
$ cmake .. -G Ninja -DPICO_BOARD=pico2_w -DPICO_PLATFORM=rp2350-riscv
$ ninja
```

Tip You may want to build with option `AVM_REBOOT_ON_NOT_OK` so AtomVM restarts on error.

The default build configuration allows the device to be re-flashed with the `atomvm_rebar3_plugin atomvm pico_flash` task or restarting the application after exiting using `picotool`. This behaviour can be changed to hang the CPU when the application exits, so that power must be cycled to restart, and `BOOTSEL` must be held when power on to flash a new application. To disable software resets use `-DAVM_WAIT_BOOTSEL_ON_EXIT=off` when configuring `cmake`.

The 20 second default timeout for a USB serial connection can be changed using option `AVM_USB_WAIT_SECONDS`. The device can also be configured to wait indefinitely for a serial connection using the option `AVM_WAIT_FOR_USB_CONNECT=on`.

8.8.6 libAtomVM build steps for RP2

Build of standard libraries is part of the generic unix build.

From the root of the project:

```
$ mkdir build
$ cd build
$ cmake .. -G Ninja
$ ninja
```

8.8.7 Running tests for RP2

Tests for RP2040 are run on the desktop (or CI) using rp2040js. Running tests currently require nodejs 20.

Change directory to the `src/platforms/rp2/tests` directory under the AtomVM source tree root:

```
$ cd <atomvm-source-tree-root>
$ cd src/platforms/rp2/tests
$
```

Install the emulator and required Javascript dependencies:

```
$ npm install
```

We are assuming tests were built as part of regular build of AtomVM. Run them with the commands:

```
$ npx tsx run-tests.ts ../build/tests/rp2040_tests.uf2 \
../build/tests/test_erl_sources/rp2040_test_modules.uf2
```

8.9 Building for `emscripten`

Two different builds are possible, depending on link options: for NodeJS and for the web browser.

8.9.1 WASM Prerequisites

- emscripten SDK
- `cmake`
- Erlang/OTP
- Elixir (optional)

8.9.2 Building for NodeJS

This is the default. Execute the following commands:

```
$ cd src/platforms/emscripten/
$ mkdir build
$ cd build
$ emcmake cmake ..
$ emmake make -j
```

AtomVM can then be invoked as on Generic Unix with node:

```
$ node ./src/AtomVM.js
```

8.9.3 Running tests with NodeJS

NodeJS build currently does not have dedicated tests. However, you can run AtomVM library tests that do not depend on unimplemented APIs.

Build them first by building AtomVM for Generic Unix (see above.) Then execute the tests with:

```
$ cd src/platforms/emscripten/build/
$ node ./src/AtomVM.js ../../../../build/tests/libs/eavmlib/test_eavmlib.avm
$ node ./src/AtomVM.js ../../../../build/tests/libs/alisp/test_alisp.avm
```

8.9.4 Building for the web

Execute the following commands:

```
$ cd src/platforms/emscripten/
$ mkdir build
$ cd build
$ emcmake cmake .. -DAVM_EMSCRIPTEN_ENV=web
$ emmake make -j
```

8.9.5 Running tests with Cypress

AtomVM WebAssembly port on the web uses SharedArrayBuffer feature which is restricted by browsers. Tests require an HTTP server that returns the proper HTTP headers.

Additionally, tests require Cypress. Plus, because of a current bug in Cypress, tests only run with Chrome-based browsers except Electron (Chromium, Chrome or Edge).

Build first AtomVM for Generic Unix (see above). This will include the web server.

Then run the web server with:

```
$ cd build
$ ./src/AtomVM examples/emscripten/wasm_webserver.avm
```

In another terminal, compile specific test modules that are not part of examples.

```
$ cd src/platforms/emscripten/build/
$ make emscripten_erlang_test_modules
```

Then run tests with Cypress with:

```
$ cd src/platforms/emscripten/tests/
$ npm install cypress
$ npx cypress run --browser chrome
```

You can alternatively specify: `chromium` or `edge` depending on what is installed.

Alternatively, on Linux, you can run tests with docker:

```
$ cd src/platforms/emscripten/tests/
$ docker run --network host -v $PWD:/mnt -w /mnt cypress/included:12.17.1 \
--browser chrome
```

Or you can open Cypress to interactively run selected test suites.

```
$ cd src/platforms/emscripten/tests/
$ npm install cypress
$ npx cypress open
```

9 AtomVM Internals

9.1 What is an Abstract Machine?

AtomVM is an "abstract" or "virtual" machine, in the sense that it simulates, in software, what a physical machine would do when executing machine instructions. In a normal computing machine (e.g., a desktop computer), machine code instructions are generated by a tool called a compiler, allowing an application developer to write software in a high-level language (such as C). (In rare cases, application developers will write instructions in assembly code, which is closer to the actual machine instructions, but which still requires a translation step, called "assembly", to translate the assembly code into actual machine code.) Machine code instructions are executed in hardware using the machine's Central Processing Unit (CPU), which is specifically designed to efficiently execute machine instructions targeted for the specific machine architecture (e.g., Intel x86, ARM, Apple M-series, etc.) As a result, machine code instructions are typically tightly packed, encoded instructions that require minimum effort (on the part of the machine) to unpack an interpret. These a low level instructions unsuited for human interpretation, or at least for most humans.

AtomVM and virtual machines generally (including, for example, the Java Virtual Machine) perform a similar task, except that i) the instructions are not machine code instructions, but rather what are typically called "bytecode" or sometimes "opcode" instructions; and ii) the generated instructions are themselves executed by a runtime execution engine written in software, a so-called "virtual" or sometimes "abstract" machine. These bytecode instructions are generated by a compiler tailored specifically for the virtual machine. For example, the `javac` compiler is used to translate Java source code into Java VM bytecode, and the `erlc` compiler is used to translate Erlang source code into BEAM opcodes.

AtomVM is an abstract machine designed to implement the BEAM instruction set, the 170+ (and growing) set of virtual machine instructions implemented in the Erlang/OTP BEAM.

Note There is no abstract specification of the BEAM abstract machine and instruction set. Instead, the BEAM implementation by the Erlang/OTP team is the definitive specification of its behavior.

At a high level, the AtomVM abstract machine is responsible for:

- Loading and execution of the BEAM opcodes encoded in one or more BEAM files;
- Managing calls to internal and external functions, handling return values, exceptions, and crashes;
- Creation and destruction of Erlang "processes" within the AtomVM memory space, and communication between processes via message passing;
- Memory management (allocation and reclamation) of memory associated with Erlang "processes"
- Pre-emptive scheduling and interruption of Erlang "processes"
- Execution of user-defined native code (Nifs and Ports)
- Interfacing with the host operating system (or facsimile)

This document provides a description of the AtomVM abstract machine, including its architecture and the major components and data structures that form the system. It is intended for developers who want to get involved in bug fixing or implementing features for the VM, as well as for anyone interested in virtual machine internals targeted for BEAM-based languages, such as Erlang or Elixir.

9.2 AtomVM Data Structures

This section describes AtomVM internal data structures that are used to manage the load and runtime state of the virtual machine. Since AtomVM is written in C, this discussion will largely be in the context of native C data structures (i.e., `structs`). The descriptions will start at a fairly high level but drill down to some detail about the data structures, themselves. This narrative is important, because memory is limited on the target architectures for AtomVM (i.e., micro-controllers), and it is important to always be aware of how memory is organized and used in a way that is as space-efficient as possible.

9.2.1 The GlobalContext

We start with the top level data structure, the `GlobalContext` struct. This object is a singleton object (currently, and for the foreseeable future), and represents the root of all data structures in the virtual machine. It is in essence in 1..1 correspondence with instances of the virtual machine.

Note Given the design of the system, it is theoretically possible to run multiple instances of the AtomVM in one process space. However, no current deployments make use of this capability.

In order to simplify the exposition of this structure, we break the fields of the structure into manageable subsets:

- Process management – fields associated with the management of Erlang (lightweight) "processes"
- Atoms management – fields associated with the storage of atoms
- Module Management – fields associated with the loading of BEAM modules
- Reference Counted Binaries – fields associated with the storage of binary data shared between processes
- Other data structures

These subsets are described in more detail below.

Warning Not all fields of the `GlobalContext` structure are described in this document.

Process Management

As a BEAM implementation, AtomVM must be capable of spawning and managing the lifecycle of Erlang lightweight processes. Each of these processes is encapsulated in the `Context` structure, described in more detail in subsequent sections.

The `GlobalContext` structure maintains a list of running processes and contains the following fields for managing the running Erlang processes in the VM:

- `processes_table` the list of all processes running in the system
- `waiting_processes` the subset of processes that are waiting to run (e.g., waiting for a message or timeout condition).
- `running_processes` the subset of processes that are currently running.
- `ready_processes` the subset of processes that are ready to run.

Processes are in either `waiting_processes`, `running_processes` or `ready_processes`. A running process can technically be moved to the ready list while running to signify that if it yields, it will be eligible for being run again, typically if it receives a message. Also, native handlers (ports) are never moved to the `running_processes` list but are in the `waiting_processes` list when they run (and can be moved to `ready_processes` list if they are made ready while running).

Each of these fields are doubly-linked list (ring) structures, i.e, structs containing a `prev` and `next` pointer field. The `Context` data structure begins with two such structures, the first of which links the `Context` struct in the `processes_table` field, and the second of which is used for either the `waiting_processes`, the `ready_processes` or the `running_processes` field.

Tip The C programming language treats structures in memory as contiguous sequences of fields of given types. Structures have no hidden preamble data, such as you might find in C++ or who knows what in even higher level languages. The size of a struct, therefore, is determined simply by the size of the component fields.

The relationship between the `GlobalContext` fields that manage BEAM processes and the `Context` data structures that represent the processes, themselves, is illustrated in the following diagram:

GlobalContext Processes

See also

The Context data structure is described in more detail below.

9.2.2 Contexts

9.3 The Scheduler

In SMP builds, AtomVM runs one scheduler thread per core. Scheduler threads are actually started on demand. The number of scheduler threads can be queried with `erlang:system_info/1` and be modified with `erlang:system_flag/2`. All scheduler threads are considered equal and there is no notion of main thread except when shutting down (main thread is shut down last).

Each scheduler thread picks a ready process and execute it until it yields. Erlang processes yield when they are waiting (for a message) and after a number of reductions elapsed. Native processes yield when they are done consuming messages (when the handler returns).

Once a scheduler thread is done executing a process, if no other thread is waiting into `sys_poll_events`, it calls `sys_poll_events` with a timeout that correspond to the time to wait for next execution. If there are ready processes, the timeout is 0. If there is no ready process, this scheduler thread will wait into `sys_poll_event` and depending on the platform implementation, the CPU usage can drop.

If there already is one thread in `sys_poll_events`, other scheduler threads pick the next ready process and if there is none, wait. Other scheduler threads can also interrupt the wait in `sys_poll_events` if a process is made ready to run. They do so using platform function `sys_signal`.

9.4 Tasks and synchronization mechanisms

AtomVM SMP builds run on operating or runtime systems implementing tasks (FreeRTOS SMP on ESP32, Unix and WebAssembly) as well as on systems with no task implementation (Raspberry Pi RP2).

On runtime systems with tasks, each scheduler thread is implemented as a task. On RP2, a scheduler thread runs on Core 0 and another one runs on Core 1, and they are effectively pinned to each core.

For synchronization purposes, AtomVM uses mutexes, condition variables, RW locks, spinlocks and Atomics.

Availability of RW Locks and atomics are verified at compile time using detection of symbols for RW Locks and `ATOMIC_*_LOCK_FREE` C11 macros for atomics.

Mutexes and condition variables are provided by the SDK or the runtime system. If RW Locks are not available, AtomVM uses mutexes. Atomics are not available on RP2 and are replaced by critical sections. Spinlocks are implemented by AtomVM on top of Atomics, or using mutexes on RP2.

Importantly, locking synchronization mechanisms (mutexes, RW locks, spinlocks) are not interrupt-safe. Interrupt service routines must not try to lock as they could fail forever if interrupted code owns the lock. Atomics, including emulation on RP2040, are interrupt-safe.

Drivers can send messages from event callbacks typically called from FreeRTOS tasks using `globalcontext_send_message_from_task` or `port_send_message_from_task` functions instead of `globalcontext_send_message` or `port_send_message`. These functions try to acquire required locks and if they fail, enqueue sent message in a queue, so it is later processed when the scheduler performs context switching. The functions are undefined if option `AVM_DISABLE_TASK_DRIVER` is passed. Some platforms do not include support for task drivers. Define `AVM_TASK_DRIVER_ENABLED` can be checked to determine if these functions are available.

9.5 Mailboxes and signals

Erlang processes receive messages in a mailbox. The mailbox is the interface with other processes.

When a sender process sends a message to a recipient process, the message is first enqueued into an outer mailbox. The recipient process eventually moves all messages from the outer mailbox to the inner mailbox. The reason for the inner and outer mailbox is to use lock-free data structures using atomic CAS operations.

Sometimes, Erlang processes need to query information from other processes but without sending a regular message, for example when using `process_info/1,2` nif. This is handled by signals. Signals are special messages that are enqueued in the outer mailbox of a process. Signals are processed by the recipient process when regular messages from the outer mailbox are moved to the inner mailbox. Signal processing code is part of the main loop and transparent to recipient processes. Both native handlers and erlang processes can receive signals. Signals are also used to run specific operation on other processes that cannot be done from another thread. For example, signals are used to perform garbage collection on another process.

When an Erlang process calls a nif that requires such an information from another process such as `process_info/1,2`, the nif returns a special value and set the `Trap` flag on the calling process. The calling process is effectively blocked until the other process is scheduled and the information is sent back using another signal message. This mechanism can also be used by nifs that want to block until a condition is true.

9.6 Stacktraces

Stacktraces are computed from information gathered at load time from BEAM modules loaded into the application, together with information in the runtime stack that is maintained during the execution of a program. In addition, if a BEAM file contains a `Line` chunk, additional information is added to stack traces, including the file name (as defined at compile time), as well as the line number of a function call.

Tip Adding line information to a BEAM file adds non-trivial memory overhead to applications and should only be used when necessary (e.g., during the development process). For applications to make the best use of memory in tightly constrained environments, packagers should consider removing line information all together from BEAM files and rely instead on logging or other mechanisms for diagnosing problems in the field.

Newcomers to Erlang may find stacktraces slightly confusing, because some optimizations taken by the Erlang compiler and runtime can result in stack frames "missing" from stack traces. For example, tail-recursive function calls, as well as function calls that occur as the last expression in a function clause, don't involve the creation of frames in the runtime stack, and consequently will not appear in a stacktrace.

9.6.1 Line Numbers

Including file and line number information in stacktraces adds considerable overhead to both the BEAM file data, as well as the memory consumed at module load time. The data structures used to track line numbers and file names are described below and are only created if the associated BEAM file contains a `Line` chunk.

The line-refs table

The line-refs table is an array of 16-bit integers, mapping line references (as they occur in BEAM instructions) to the actual line numbers in a file. (Internally, BEAM instructions do not reference line numbers directly, but instead are indirected through a line index). This table is stored on the `Module` structure.

This table is populated when the BEAM file is loaded. The table is created from information in the `Line` chunk in the BEAM file, if it exists. Note that if there is no `Line` chunk in a BEAM file, this table is not created.

The memory cost of this table is `num_line_refs * 2` bytes, for each loaded module, or 0, if there is no `Line` chunk in the associated BEAM file.

The filenames table

The filenames table is a table of (usually only 1?) file name. This table maps filename indices to `ModuleFilename` structures, which is essentially a pointer and a length (of type `size_t`). This table generally only contains 1 entry, the file name of the Erlang source code module from which the BEAM file was generated. This table is stored on the `Module` structure.

Note that a `ModuleFilename` structure points to data directly in the `Line` chunk of the BEAM file. Therefore, for ports of AtomVM that memory-map BEAM file data (e.g., ESP32), the actual file name data does not consume any memory.

The memory cost of this table is `num_filenames * sizeof(struct ModuleFilename)`, where `struct ModuleFilename` is a pointer and length, for each loaded module, or 0, if there is no `Line` chunk in the associated BEAM file.

The line-ref-offsets list

The line-ref-offsets list is a sequence of `LineRefOffset` structures, where each structure contains a ListHead (for list book-keeping), a 16-bit line-ref, and an unsigned integer value designating the code offset at which the line reference occurs in the code chunk of the BEAM file. This list is stored on the `Module` structure.

This list is populated at code load time. When a line reference is encountered during code loading, a `LineRefOffset` structure is allocated and added to the line-ref-offsets list. This list is used at a later time to find the line number at which a stack frame is called, in a manner described below.

The memory cost of this list is `num_line_refs * sizeof(struct LineRefOffset)`, for each loaded module, or 0, if there is no `Line` chunk in the associated BEAM file.

TODO

9.7 ETS Tables

AtomVM includes a very basic implementation of the OTP ETS interface, allowing applications to efficiently store term data in tables, and to access term data across processes.

> For information about ETS tables from the user's perspective, see the AtomVM Programmers Guide.

ETS tables are represented internally by the `EtsTable` structure, which records information about the table configured by the user (via `ets:new/2`), such as the table name, whether is it a named table, the access type (`private`, `protected`, or `public`), the key position, etc.

ETS tables are uniquely identifiable via Erlang references, and generally this reference is returned as the "table identifier" when an ETS table is created (via `ets:new/2`). ETS tables can also be referenced via their name, if they are designated as a "named table" in configuration. Table identifiers are unique, and the ETS implementation also ensures that if tables are named, they are named uniquely.

In addition, the `EtsTable` structure records the table type, which in the current implementation is limited to the `set` type. Future versions may support alternative table types (`ordered_set`, `bag`, `duplicate_bag`).

For the `set` table type, the `EtsTable` instantiates an `EtsHashTable`, a hash table implementation with an array of buckets (to which keys hash), and for each bucket, a (possibly empty) linked list of `HNode` elements, where `HNode` in the list contains a reference to an AtomVM `Heap` (fragment). Each `Heap` instance stores a copy of the tuple element that has been added to the table (via `ets:insert /2`). The `HNode` also stores a `term` representing the key used to store the element, as well as a term representing the element, itself. Note that the `key` and `entry` term fields refer to elements stored in the heap fragment referenced from the `HNode`, not terms in a process heap.

When an element is inserted into the table or retrieved from the table (e.g., via `ets:lookup/2`), the `term` data is copied into or from, respectively, the `Heap` instance stored in the table. The `Heap` instance in each entry is allocated to hold exactly the amount of data needed to store the `term`. This `Heap` is owned by the `EtsHashTable`, in the sense that an instance is created when an entry is inserted into the table, and destroyed when an entry is deleted from the table.

For more information about the `Heap` data structure, see the Memory Management chapter of the AtomVM documentation.

The `Ets` structure contains a `SyncList` of `EtsTable` structures, protected from concurrent access via a read-write lock. A single instance of an `Ets` structure is stored in the `GlobalContext` structure.

Note A deficiency in this implementation is that lookup of an ETs table in this list is `O(<number of ETS tables>)`, whereas a "promise" of ETS is that insertion, lookup, and deletion of entries in at least the `set` table type is or should be constant (modulo poor hashing functions). A future implementation of this data structure could use a map or table structure to make lookup of ETS tables more efficient (i.e., constant), but at the expense of i) increased memory consumption and heap fragmentation, and ii) code complexity. Given the likely relative paucity of ETS tables instances likely to be instantiated by applications, we feel that using a list of ETS tables is an acceptable tradeoff for embedded systems deployments, but we are open to being wrong about our assumptions.

9.7.1 Ownership and Lookup

An ETS table is conceptually "owned" by the Erlang process that creates it (via `ets:new/2`). This generally means that:

- The owning process has exclusive access for reading and writing entries in the table (unless the table is declared `public`, in which case other processes may change entries in the table, e.g., via `ets:insert/2` or `ets:delete/2`);
- The lifecycle of the ETS table is associated with the lifecycle of the owning process, meaning that the ETS table is destroyed when the owning process terminates. Note that any references to the ETS table are invalid once the table has been destroyed.

```
{important} AtomVM does not currently support transfer of ownership of
an ETS table from one process to another.
```

When a table is used in an `ets` operation (e.g., `ets:insert/2`, `ets:lookup/2`, `ets:delete/2`, etc), the `ets_tables` sync list is searched for the first entry in the table that matches the requested table id (as a name or reference).

If the Erlang process that requests a table is the same process that owns the table, then the operation is permitted. Otherwise, the access type on the table must be `public` in order for an operation that modifies the table (e.g., `ets:insert/2` or `ets:delete/2`) to be permitted, or `protected` in order for an operation that simply reads from the table (e.g., `ets:lookup/2`).

9.7.2 Key Hashing

A "key" in an ETS table is an arbitrary term. In order to insert, lookup, or delete a key in an `EtsHashTable`, we need a way to efficiently hash a key into a bucket, preferably without any memory allocation.

The implementation of the `EtsHashTable` uses a variant of the Character Folding hash algorithm, using a collection of sufficiently large prime numbers for each Erlang term type (atom, integer, tuple, list, map, etc), and adapted to Erlang recursively structured terms.

This algorithm resembles the term hashing algorithm in OTP; however, for the purposes of the `EtsHashTable` implementation, the algorithm is not designed to be interoperable with the OTP implementation. Future versions of AtomVM that, for example, implement the Erlang portable hash function (`erlang:phash2/1,2`) can be adapted to use the OTP algorithm for this implementation.

9.7.3 Concurrency

ETS tables are designed to be accessed concurrently from multiple Erlang processes. Depending on how table are configured, they may be accessed concurrently for reads but serially for writes (typical), or in probably non-standard cases, concurrently for both reads and writes.

The `ets_tables` sync list is protected by a read-write lock, so that multiple processes calling operations that read the list (e.g., for common `ets` operations like `ets:insert/1`, `ets:lookup/2`, or `ets:delete/2`) can do so concurrently, whereas operation that mutate this list (e.g., `ets:new/2` or termination of a process) are serialized with a write lock.

When an `EtsTable` is retrieved from this list, before the read lock is released on the sync list, either a read or write lock is acquired on the `EtsTable`, depending on the operation being called. For example, operations that change the state of the table, such as `ets:insert/2` or `ets:delete/2`, acquire a write lock, whereas operation that merely read table contents, such as `ets:lookup/2` acquire a read lock. This way, a lock is held on the `EtsTable` before another thread might race to delete the table, which would otherwise render the reference to the `EtsTable` invalid, and likely crash the VM.

9.8 AtomVM WebAssembly port

WebAssembly or Wasm port of AtomVM relies on Emscripten SDK and library. Even when SMP is disabled (with `-DAVM_DISABLE_SMP=On`), it uses pthread library to sleep when Erlang processes are not running (to not waste CPU cycles).

9.8.1 NodeJS environment build

The NodeJS environment build of this port is relatively straightforward, featuring NODERAWFS which means it can access files directly like node does.

9.8.2 Web environment build

The Web environment build of this port is slightly more complex.

Regarding files, `main` function can load modules (beam or AVM packages) using FetchAPI, which means they can be served by the same HTTP server. This is a fallback and users can preload files using Emscripten `file_packager` tool.

The port also uses Emscripten's proxy-to-pthread feature which means AtomVM's `main` function is run in a web worker. The rationale is the browser thread (or main thread) with WebAssembly cannot run a loop such as AtomVM's schedulers. Web workers typically cannot manipulate the DOM and do other things that only the browser's main thread can do. For this purpose, Erlang processes can call `emscripten:run_script/2` function which dispatches the Javascript to execute to the main thread, waiting for completion (with `[main_thread]`) or not waiting for completion (with `[main_thread, async]`). Waiting for completion of a script on the main thread does not block the Erlang scheduler, other Erlang processes can be scheduled. Execution of Javascript on the worker thread, however, does block the scheduler.

Javascript code can also send messages to Erlang processes using `call` and `cast` functions from `main.c`. These functions are actually wrapped in `atomvm.pre.js`. Usage is demonstrated by `call_cast.html` example.

Cast is straightforward: the message is enqueued and picked up by the scheduler. It is freed when it is processed.

Call allows Javascript code to wait for the result and is based on Javascript promises (related to `async /await` syntax).

1. A promise is created (in the browser's main thread) in a map to prevent Javascript garbage collection (this is done by Emscripten's promise glue code).
2. An Erlang resource is created to encapsulate the promise so it is properly destroyed when garbage collected
3. A message is enqueued with the resource as well as the registered name of the target process and the content of the message
4. C code returns the handle of the promise (actually the index in the map) to Javascript Module.-call wrapper.
5. The `Module.call` wrapper converts the handle into a Promise object and returns it, so Javascript code can await on the promise.
6. A scheduler dequeues the message with the resource, looks up the target process and sends it the resource as a term
7. The target process eventually calls `emscripten:promise_resolve/1,2` or `emscripten:promise_reject/1,2` to resolve or reject the promise.
8. The `emscripten:promise_resolve/1,2` and `emscripten:promise_reject/1,2` nifs dispatch a message in the browser's main thread.
9. The dispatched function retrieves the promise from its index, resolves or rejects it, with the value passed to `emscripten:promise_resolve/2` or `emscripten:promise_reject/2` and destroys it.

Values currently can only be integers or strings.

If the scheduler cannot find the target process, the promise is rejected with "noproc" as a value. As the promise is encapsulated into an Erlang resource, if the resource object's reference count reaches 0, the promise is rejected with "noproc" as the value.

10 Memory Management

Like most managed execution environments, AtomVM provides automated memory management for compiled Erlang/Elixir applications that run on the platform, allowing developers to focus on the logic of application programs, instead of the minutiae of managing the allocation and disposal of memory in the process heap of the program.

Because Erlang/Elixir, and the BEAM, specifically, is a shared-nothing, concurrency-based language, AtomVM can manage memory independently, for each unit of concurrency, viz., the Erlang process. While there is some global state, internally, that AtomVM manages (e.g., to manage all running processes in the system), memory management for each individual process can be performed independently of any other process.

AtomVM internally uses a "Context" structure, to manage aspects of a process (including memory management), and we use "execution context" and "Erlang process" interchangeably in this document. As usual, an Erlang process should be distinguished from the Operating System (OS) process in which Erlang processes run.

For any given execution context, there are three regions of memory that are relevant: i) the stack, ii) the heap, and iii) registers. The stack and heap actually occupy one region of memory allocated in the OS process heap (via malloc or equiv), and grow in opposite directions towards each other. Registers in AtomVM are a fixed size array of 16 elements.

The fundamental unit of memory that occupies space in the stack, heap, and registers is the `term`, which is typedef'd internally to be an integral type that fits in a single word of machine memory (i.e., a C `int`). Various tricks are used, described below, to manage and reference multi-word terms, but in general, a term (or in some cases, a term pointer) is intended to fit into a single word or memory.

This document describes the memory layout for each execution context (i.e., Erlang/Elixir process), how memory is allocated and used, how terms are represented internally, and how AtomVM makes room for more terms, as memory usage increases and as terms go out of scope and are no longer used by the application, and can hence be garbage collected.

10.1 The Context structure

10.1.1 The Heap and Stack

The heap and stack for each AtomVM process are stored in a single allocated block of memory (e.g., via the `malloc` C function) in the heap space of the AtomVM program, and the AtomVM runtime manages the allocation of portions of this memory during the execution of a program. The heap starts at the bottom of the block of memory, and grows incrementally towards the top of the allocated block, as memory is allocated in the program. Each word in the heap and stack (or in some cases, a sequence of words) represent a term that has been allocated.

The heap contains all of the allocated terms in an execution context. In some cases, the terms occupy more than one word of memory (e.g., a tuple), but in general, the heap contains a record of memory in use by the program.

The heap grows incrementally, as memory is allocated, and terms are allocated sequentially, in increasing memory addresses. There is, therefore, no memory fragmentation, properly speaking, at least insofar as a portion of memory might be in use and then freed. However, it is possible that previously allocated blocks of memory in the context heap are no longer referenced by the program. In this case, the allocated blocks are "garbage", and are reclaimed at the next garbage collection. The actual growth of the heap is controlled by a heap growth strategy (`atomvm_heap_growth` spawn option) as described below.

> It is possible for the AtomVM heap, as provided by the underlying operating system, to become fragmented, as the execution context stack and heap are allocated via `malloc` or equiv. But

that is a different kind of fragmentation that does not refer to the allocated block used by an individual AtomVM process.

The stack grows from the top of the allocated block toward the heap in decreasing addresses. Terms in the stack, as opposed to the heap, are either single-word terms, i.e., simple terms like small integers, process ids, etc, or *pointers* to terms in the heap. In either case, they only occupy one word of memory.

The region between the stack and heap is the free space available to the Erlang/Elixir process.

The following diagram illustrates an allocated block of memory that stores terms (or term pointers) in the heap and stack:

```
+================================+ <- heap_start --
|             word[0]            |      ^      ^
+--------------------------------+      |      |
|             word[1]            |      |      |
+--------------------------------+      |      |
|             word[2]            |      | heap |
+--------------------------------+      |      |
|               ...              |      |      |
+--------------------------------+      |      |
|                                |      v      |
+--------------------------------+ <- heap_ptr |
|                                |      ^      |
|                                |      |      |
|                                |      |      |
|                                |      | free |
|                                |      |      |
|                                |      |      |
|                                |      v      |
+--------------------------------+ <- e ----   |
|                                |      ^      |
+--------------------------------+      |      |
|                                |      |      |
+--------------------------------+      | stack|
|                                |      |      |
+--------------------------------+      |      |
|           word[n-1]            |      v      v
+================================+ <- stack_base --
```

The initial size of the allocated block for the stack and heap in AtomVM is 8 words. As heap and stack allocations grow, eventually, the amount of free space will decrease to the point where a garbage collection is required. In this case, a new but larger block of memory is allocated by the AtomVM OS process, and terms are copied from the old stack and heap to the new stack and heap. Garbage collection is described in more detail below.

10.1.2 Heap growth strategies

AtomVM aims at minimizing memory footprint and several heap growth strategies are available. The heap is grown or shrunk when an allocation is required and the current execution context allows for a garbage collection (that will move data structures), allows for shrinking or forces shrinking (typically in the case of a call to `erlang:garbage_collect/0,1`).

Each strategy is set at the process level.

Default strategy is bounded free (`{atomvm_heap_growth, bounded_free}`). In this strategy, when more memory is required, the allocator keeps the free amount between fixed boundaries (currently 16 and 32 terms). If no allocation is required but free space is larger than boundary, a garbage collection is triggered. After copying data to a new heap, if the free space is larger than the maximum, the heap is shrunk within the boundaries.

With minimum strategy (`{atomvm_heap_growth, minimum}`), when an allocation can happen, it is always adjusted to have the free space at 0.

With fibonacci strategy (`{atomvm_heap_growth, fibonacci}`), heap size grows following a variation of fibonacci until a large value and then grows by 20%. If free space is larger than 75% of heap size, the heap is shrunk. This strategy is inspired from Erlang/OTP's implementation.

10.1.3 Registers

Registered are allocated in an array of 16 terms (words) and are referenced by the `x` field in the Context data structure:

```
+---------+---------+---------+--------+
|   x[0]  |   x[1]  |   ...   |  x[15] |
+---------+---------+---------+--------+
```

Like terms in the stack, terms in registers are either single-word terms, i.e., simple terms like small integers, process ids, etc, or *pointers* to terms in the heap, in a manner described in more detail below. In either case, they only occupy one word of memory.

Registers are used as part of the BEAM instruction set to store and retrieve values that are passed between BEAM instruction opcodes.

10.1.4 Process Dictionary

AtomVM processes support a process dictionary, or map of process-specific data, as supported via the `erlang:put/2` and `erlang:get/1` functions.

The Process Dictionary contains a list of key-value pairs, where each key and value is a single-word term, either a simple term like an atom or pid, or a reference to an allocated object in the process heap. (see below)

10.1.5 Heap Fragments

AtomVM makes use of heap fragments in some edge cases, such as loading external terms from the literals table in a BEAM file. Heap fragments are individually allocated blocks of memory that contain may contain multi-word term structures. The data in heap fragments are copied into the heap during a garbage collection event, and then deleted, so heap fragments are generally short lived. However, during execution of a program, there may be references to term structures in such fragments from the stack, registers, the process dictionary, or from nested terms in the process heap.

10.1.6 Mailbox

Each Erlang process contains a process mailbox, which is a linked-list structure of messages. Each message in this list contains a term structure, which is a copy of a term sent to it, e.g., via the `erlang:send/2` operation, or `!` operator.

The representation of terms in a message is identical to that in the heap and heap fragments. Messages are allocated like fragments and they actually become heap fragments of the receiving process when the message is read off the mailbox (e.g., via `receive ... end`). Messages (and their term contents) are moved to the main heap as part of regular garbage collection of the process, and the fragment is freed.

10.1.7 Memory Graph

Memory is allocated in the execution context heap, and structured types, such as tuples and lists, generally include references to the blocks of memory that have been previously allocated.

For example, if we look at the memory allocated for the term

```
{foo, [{bar, self()}]}
```

we would generally see something like the following in the execution context heap:

```
|              ...              |
|                            |
+----------------------------+
|          tuple             |<--+
+----------------------------+   |
|          bar               |   |
+----------------------------+   |
|          <0.1.0>           |   |
+----------------------------+   |
|          []                |<- | --+
+----------------------------+   |   |
|          tuple ptr         |---+   |
+----------------------------+       |
|          tuple             |       |
+----------------------------+       |
|          foo               |       |
+----------------------------+       |
|          list ptr          |-------+
+----------------------------+
|                            |
|           ...              |
0123456789012345678901234567890123456789
```

The tuple `{bar, self()}` is allocated in a block, and the list `[{bar, self()}]` (or, technically, `[{bar, self()} | []]`) contains elements that *point* to it elements (in this case, `[]` and `{bar, self()}` – note that in general, in AtomVM, the address of the tail of a list occupies the first byte in the list – more details on that below). Finally, the tuple `{foo, [{bar, self()}]}` contains the atom `foo` and a *pointer* to the list it contains.

In this way, the set of allocated blocks in the execution context heap forms a directed graph of objects, whose nodes are structured terms (lists, tuples, etc) and whose leaves are simple terms, like atoms, pids, and so forth. Note that because BEAM-based languages such as Erlang and Elixir are true functional programming languages, these directed graphs have no cycles.

The stack, registers, and process dictionary contain pointers to terms in the heap. We call these terms "root" nodes, and any term in the heap that is referenced by a root node, or any term that is so referenced by such a term, is in the path of a root node. Some terms in the heap are not in the path of a root node. We call these terms "garbage".

Note that the values in the stack and register root nodes change over time as the result of the execution of Erlang opcodes, and are dependent on the BEAM output of the Erlang compiler, along with inputs to the program being executed. Thus, a term in the process heap may become garbage, once it is no longer reachable from the root set. But once garbage, the term will always remain garbage, at least until it is reclaimed during a garbage collection event. For more information about how the garbage collector works, see the Garbage Collection section, below.

10.2 Simple Terms

The fundamental unit of memory in AtomVM is the `term` object, which is designed to fit either into a single machine work (single-word terms), or into multiple words (so called "boxed terms" and lists).

This section enumerates the AtomVM term types, and how they are represented in memory.

Note The term type is overloaded in some cases to store raw pointers to memory addresses, but this is rare and well controlled.

The following term types take up a single word, referred to as "immediates" in the BEAM documentation[1]. The low-order bits of the word are used to represent the type of the term, and the high order bits represent the term contents, in a manner described in the following sections.

10.2.1 Atoms

An atom is represented as a single word, with the low-order 6 bits having the value `0xB` (`001011b`). The high order word-size-6 bits are used to represent the index of the atom in the global atom table:

```
                            |< 6  >|
+===========================+======+
|       atom index          |001011| <- 0xB
+===========================+======+
|                                  |
|<---------- word-size --------->|
```

There may therefore only be `2^{word-size-6}` atoms in an AtomVM program (e.g., on a 32-bit platform, `67,108,864`). Plenty to work with!

Note The global atom table is a table of all allocated atoms, and is generally (at least in the limit, as modules are loaded) a fixed size table. Management of the global atom table is outside of the scope of this document.

10.2.2 Integers

An integer is represented as a single word, with the low-order 4 bits having the value `0xF` (`1111b`). The high order word-size-6 bits are used to represent the integer value:

```
                              |< 4>|
+=============================+====+
|       integer value         |1111| <- 0xF
+=============================+====+
|                                  |
|<---------- word-size --------->|
```

The magnitude of an integer is therefore limited to `2^{word-size - 4}` in an AtomVM program (e.g., on a 32-bit platform, `+- 134,217,728`).

Attention! Arbitrarily large integers (bignums) are not currently supported in AtomVM.

10.2.3 nil

The special value `nil` (typically the tail of the tail … of the tail of a list, or `[]`) is the special value 0x3B:

```
+==================================+
|000          ...        0000111011| <- 0x3B
+==================================+
|                                  |
|<---------- word-size --------->|
```

10.2.4 Pids

A Pid is represented as a single word, with the low order 4 bits indicating the Pid term type (`0x03`), and (for now), the high order `word-size - 4` bits store the local process id:

```
                              |< 4>|
+=============================+====+
|     local process id        |0011| <- 0x3
+=============================+====+
|                                  |
|<---------- word-size --------->|
```

There may therefore only be `2^{word-size - 4}` Pids in an AtomVM program (e.g., on a 32-bit platform, `268,435,456`).

Note Global process IDs are not currently supported, but they may be in the future, which may result in segmentation of the high order `word-size - 4` bits.

10.3 Boxed terms

Some term types cannot fit in a single word, and must therefore used a sequence of contiguous words to represent the term contents. These terms are called "Boxed" terms. Boxed terms use the low-order 6 bits of the first byte (`boxed[0]`) to represent the term type, and the high order `word-size - 6` bits to represent the remaining size (in words) of the boxed term, not including the first word.

10.3.1 Boxed term pointers

Before discussing the different types of boxed terms in detail, let us first see how boxed terms are referenced from the stack, registers, process dictionary, and from embedded terms in the heap. We call such references to boxed terms boxed term pointers.

A boxed term pointer is a single-word term that contains the address of the referenced term in the high-order `word-size - 2` bits, and `0x2` (`10b`) in the low-order 2 bits.

```
                               |2 |
+==============================+==+
|       term address           |10| <- term pointer type (2 bits)
+==============================+==+
|                                 |
|<---------- word-size --------->|
```

Because terms (and hence the heap) are always aligned on boundaries that are divisible by the word size, the low-order 2 bits of a term address are always 0. Consequently, the high-order word-size - 2 (`1,073,741,824`, on a 32-bit platform) are sufficient to address any term address in the AtomVM address space, for 32-bit and greater machine architectures.

10.3.2 References

A reference (e.g., created via `erlang:make_ref/0`) stores a 64-bit incrementing counter value (a "ref tick"). On 64 bit machines, a Reference takes up two words – the boxed header and the 64-bit value, which of course can fit in a single word. On 32-bit platforms, the high-order 28 bits are stored in `boxed[1]`, and the low-order 32 bits are stored in `boxed[2]`:

```
                          |< 6   >|
+=========================+======+
|        boxed-size       |010000| boxed[0]
+-------------------------+------+
|     high-order ref-ticks       | boxed[1]
+================================+
|      low-order ref-ticks       | boxed[2] (32-bit only)
+= = = = = = = = = = = = = = = ==+
|                                |
|<---------- word-size --------->|
```

10.3.3 Tuples

Tuples are represented as boxed terms containing a boxed header (`boxed[0]`), a type tag of `0x00` (`000000b`), followed by a sequence of `n`-many words, which may either (copies of) single-word terms, or boxed term pointers, where `n` is the arity of the tuple:

```
                          |< 6  >|
+=========================+======+
|    boxed-size (n)       |000000| boxed[0]
+-------------------------+------+
|            element-1           | boxed[1]
+--------------------------------+
|            element-2           | boxed[2]
+--------------------------------+
|              ...               | boxed[i]
+--------------------------------+
|            element-n           | boxed[n]
+================================+
|                                |
|<---------- word-size --------->|
```

10.3.4 Maps

Maps are represented as boxed terms containing a boxed header (`boxed[0]`), a type tag of `0x3C` (`111100b`), followed by:

- a term pointer to a tuple of arity `n` containing the keys in the map;
- a sequence of `n`-many words, containing the values of the map corresponding (in order) to the keys in the reference tuple.

The keys and values are single word terms, i.e., either immediates or pointers to boxed terms or lists.

```
        +=========================+======+
+-----> |    boxed-tuple (n)      |000000|
|       +-------------------------+------+
|       |            key-1               |
|       +--------------------------------+
|       |            key-2               |
|       +--------------------------------+
|       |              ...               |
|       +--------------------------------+
|       |            key-n               |
|       +================================+
|       |                                |
|                      ...
|       |                         |< 6  >|
|       +=========================+======+
|       |    boxed-size (n)       |111100| boxed[0]
|       +-------------------------+------+
+-----------------<   keys               | boxed[1]
        +--------------------------------+
        |            value-1             | boxed[2]
        +--------------------------------+
        |              ...               | ...
        +--------------------------------+
        |            value-n             | boxed[2 + n]
        +================================+
        |                                |
        |<---------- word-size --------->|
```

The tuple of keys may or may not be contiguous with the boxed term holding the map itself (and in general will not be, after garbage collection). In addition, maps that are modified [sic] via the `:=` operator (or via `=>`, when the key already exists in the source map) share the keys tuple, for space efficiency.

10.3.5 Binaries

Binaries are stored in several different ways, depending on their size and the kinds of data to which they refer.

Binary data less than 64 bytes in length are stored in the process heap, as so-called Heap Binaries.

Binary data greater or equal to 64 bytes is stored in two manners, depending on whether the data stored is constant data (e.g., literal binary data compiled directly into a BEAM file), or dynamically allocated data, e.g., as the result of a call to the `erlang:list_to_binary/1` Nif.

Non-const binaries are stored outside of the heap in dynamically allocated memory and are reference-counted, whereby references to dynamically allocated blocks are tracked from pointers in heap storage. This way, large blocks of binary data can be efficiently shared between processes; only a relatively small term that contains a reference to the dynamically allocated storage needs to be copied. When the reference count of non-literal binary reaches 0, the dynamically allocated memory is free'd.

Const binaries share similar features to non-const binaries in the process heap; however, instead of pointing to dynamically allocated memory that requires reference counting and memory management, the boxed term in the process heap points directly to constant memory (e.g., a term literal stored in a memory-mapped BEAM file). This is especially useful in memory constrained applications, such as the ESP32 micro-controller, where the BEAM file contents are not read into memory, but are instead directly mapped from flash storage.

Finally, a special kind of binary is used in the heap to maintain the state of a match context, when, for example, matching binary terms using Erlang bit syntax. Like non-const binaries, creation and destruction of match context binaries will affect the reference count on the binaries to which they refer.

The following sub-sections describe these storage mechanisms and memory management in more detail.

Heap Binaries

Heap binaries are represented as boxed terms containing a boxed header (`boxed[0]`), a type tag of `0x024` (`100100b`), followed by the size in bytes of the binary, and then a sequence of `n`-many words, which contains the sequence of `size`-many bytes (`<= word-size * n`):

```
                           |< 6  >|
+=========================+======+
|    boxed-size (n)       |100100| boxed[0]
+-------------------------+------+
|         size (in bytes)        | boxed[1]
+--------------------------------+
| byte-1, byte-2, byte-3,   ...  | boxed[2]
+--------------------------------+
|              ...               | boxed[i]
+------------------+-------------+
| ..., byte-{size-1}| -unused-   | boxed[n+1]
+==================+=============+
|                                |
|<---------- word-size --------->|
```

Note If the number of bytes in a binary is not evenly divisible by the machine word size, then the remaining sequence of bytes in the last word are unused.

Reference Counted Binaries

Reference counted binaries are represented as boxed terms containing a boxed header (`boxed[0]`), a type tag of `0x020` (`100000b`), followed by the size in bytes of the binary data, a word containing a set of flags, and then a pointer to the off-heap data.

Currently, only the low-order bit of the flags field is used. A `0` value of indicates that the referenced binary is non-literal.

The off-heap data is a block of allocated data, containing:

- a ListHead structure, used to maintain a list of dynamically allocated data (mostly for book-keeping purposes);
- a reference count (unsigned integer);
- the size of the stored data;
- the stored data, itself.

All of the above data is allocated in a single block, so that it can be easily `free`'d when no longer referenced.

The reference count is initialized to 1, under the principle that that reference count is incremented for any occurrence of boxed terms that reference the same data in any heap space, including process heaps, mailbox messages, heap fragments, and so forth. Decrementing reference counts and `free`'ing data in off-heap storage is discussed in more detail below, in the Garbage Collection section.

```
                               |< 6  >|
+--> +=========================+======+
|    |    boxed-size (5)       |100000| boxed[0]
|    +-------------------------+------+
|    |         size (in bytes)        | boxed[1]
|    +--------------------------------+
|    |             flags             0| boxed[2]
|    +--------------------------------+                  off-heap storage
|    |              ptr >-------------- boxed[3] ---> +----------------------+
 ------
|    +--------------------------------+               |         prev         |
     ^
|    |              cdr               | boxed[4]      +----------------------+
     | ListHead
|    +--------------------------------+               |         next         |
     v
+-----------------< car               | boxed[5]      +----------------------+
 ------
     +================================+               |   reference-count    |
     |<---------- word-size --------->|               +----------------------+
                                                      |         size         |
                                                      +----------------------+
                                                      |         data         |
                                                                ...
                                                      |                      |
                                                      +----------------------+
```

Note The size of a reference counted binary is stored both in the process heap (in the boxed term), as well as in the off-heap storage. The size count in the off-heap storage is needed in order to report the amount of data in use by binaries (e.g., via `erlang:memory/0,1`).

In addition, a reference-counted boxed term contains a cons-cell appended to the end of the boxed term, which is used by the garbage collector for tracking references. The `car` of this cell points to the boxed term, itself, and the `cdr` points to the "previous" cons cell associated with a reference counted binary in the heap, if there is one, or the empty list (`nil`), otherwise. The cons cell forms an element in the "Mark and Sweep Object" (MSO) list, used to reclaim unreferenced storage during a garbage collection event.. See the Garbage Collection section, below, for more information about the critical role of this structure in the process of reclaiming unused memory in the AtomVM virtual machine.

Const Binaries

Const binaries are stored in the same manner as Reference Counted binaries, with the following exceptions:

- The low order bit of the flags field (`boxed[2]`) is `1`, to indicate that the reference binary is constant;
- The ptr field (`boxed[3]`) points directly to the constant storage (e.g., literal data stored in a memory-mapped BEAM file);
- The trailing cons cell elements are unused, as dynamic memory management for static storage is unnecessary. These values are initialized to `nil`.

This heap structure has the following representation:

```
                          |< 6  >|
+=========================+======+
|    boxed-size (5)       |100000| boxed[0]
+-------------------------+------+
|         size (in bytes)        | boxed[1]
+--------------------------------+
|             flags             1| boxed[2]
+--------------------------------+                    static storage
|              ptr >-------------- boxed[3] -------> +----------------------+
+--------------------------------+                   |         data         |
|             unused             | boxed[4]          |                      |
+--------------------------------+                          ...
|             unused             | boxed[5]          |                      |
+================================+                   +----------------------+
|<---------- word-size --------->|
```

Match Binaries

Match binaries are represented as boxed terms containing a boxed header (`boxed[0]`), a type tag of `0x04` (`000100b`), and the following elements:

- a reference to either a binary or another match binary that refers to a binary;
- an offset in the referenced binary used by the match opcodes;
- a saved state used for backtracking unmatched clause heads;

Like a reference counted binary, a match binary includes a trailing cons cell, whose `car` element points to the actual referenced binary (if the referenced binary is a reference-counted binary), and whose `cdr` points to the "previous" cons cell associated with a reference counted binary in the heap.

Note If the referenced binary is not reference-counted, the trailing cons cell elements are unused and are initialized to `nil`.

```
some
binary                            |< 6  >|
^       +=========================+======+
|       |    boxed-size (5)       |100100| boxed[0]
|       +-------------------------+------+
|       |      match-or-binary-ref       | boxed[1]
|       +--------------------------------+
|       |             offset             | boxed[2]
|       +--------------------------------+
|       |             saved              | boxed[3]
|       +--------------------------------+
|       |              cdr               | boxed[4]
|       +--------------------------------+
+--------------------< car               | boxed[5]
        +================================+
        |<---------- word-size --------->|
```

A reference to a reference-counted binary counts as a reference, in which case the creation or copying of a match binary results in the increment of the reference-counted binary's reference count, and the garbage collection of a match binary results in a decrement (and possible `free`ing) of a reference-counted binary. The trailing cons cell becomes an element of the context (or message) MSO list, and plays a critical role in garbage collection. See the garbage collection section below for more information about the role of this structure.

Sub-Binaries

Sub-binaries are represented as boxed terms containing a boxed header (`boxed[0]`), a type tag of `0x28` (`001000b`)

A sub-binary is a boxed term that points to a reference-counted binary, recording the offset into the binary and the length (in bytes) of the sub-binary. An invariant for this term is that the `offset + length` is always less than or equal to the length of the referenced binary.

```
    some
    refc
    binary                              |< 6   >|
    ^       +========================+======+
    |       |    boxed-size (3)      |001000| boxed[0]
    |       +------------------------+------+
    |       |              len              | boxed[1]
    |       +-------------------------------+
    |       |            offset             | boxed[2]
    |       +-------------------------------+
    +----------------< binary-ref           | boxed[3]
            +===============================+
            |<---------- word-size --------->|
```

Note than when a sub-binary is copied between processes (e.g., via `erlang:send`, or `!`), the sub-binary boxed term, as well as the boxed-term that manages the reference-counted binary is copied, as well. Thus, sending a sub-binary to another process will result in an increment of the reference count on the referenced binary, and similarly, garbage collection of the sub-binary will result in a decrement of the referenced binary's reference count.

A sub-binary may be created from both const (literal) and non-const reference-counted binaries. For performance reasons, sub-binaries do not reference heap binaries.

Sub-binaries are created via the `binary:part/3` and `binary:split/2` Nifs, as well as via `/binary` bit syntax specifier.

10.4 Lists

A list is, very simply, a cons cell, i.e., a sequence of two words, whose first word is a term (single word or term pointer) representing the tail (`cdr`) of the list, and the second of which represents the head (`car`) of the list.

```
+===============================+
|                tail           | list_elem[0]
+-------------------------------+
|                head           | list_elem[1]
+===============================+
|                               |
|<---------- word-size --------->|
```

Note Lists are typically terminated with the empty list (`[]`), represented by the nil term, described above. However, nothing in Erlang requires that a sequence of cons cells is `nil`-terminated.

Unlike boxed terms, the low-order two bits of list pointers are `0x1` (`01b`):

```
+==============================+==+
|       term address           |01| <- list pointer type (2 bits)
+==============================+==+
|                                 |
|<---------- word-size --------->|
```

10.4.1 Strings

Strings are just lists of integers, but they are efficiently allocated at creation time so that a contiguous block of cons cells are created in the heap. They otherwise have the same properties of a list described above.

```
+================================+
|    address-of-next-cons     |01| elem[1]
+--------------------------------+
|            int-value           |
+--------------------------------+
|    address-of-next-cons     |01| elem[2]
+--------------------------------+
|            int-value           |
+--------------------------------+
|               ...           |01| elem[i]
+--------------------------------+
|               ...              |
+--------------------------------+
|               nil              | elem[n]
+--------------------------------+
|            int-value           |
+================================+
|                                |
|<---------- word-size --------->|
```

Note String elements may not remain contiguous after a garbage collection event.

10.4.2 Functions

Functions are represented as boxed terms containing a boxed header (`boxed[0]`), a type tag of `0x14` (`010100b`), followed by the raw memory address of the Module data structure in which the function is defined, and the function index (so that the function can be located).

In addition, if there are any terms that are used outside of the scope of the function (i.e., closures), these terms are copied from registers into the function objects

```
                         |< 6  >|
+========================+======+
|    boxed-size (n)      |010100| boxed[0]
+------------------------+------+
|         module address         | boxed[1]
+--------------------------------+
|         function index         | boxed[2]
+--------------------------------+
|            closure_1           | boxed[3]
+- - - - - - - - - - - - - - - - +
|              ...               |
+- - - - - - - - - - - - - - - - +
|            closure_k           | boxed[n-1]
+= = = = = = = = = = = = = = = = +
|                                |
|<---------- word-size --------->|
```

10.5 Special Stack Types

Some terms are only used in the stack.

10.5.1 Continuation Pointer

A continuation pointer is a raw address. Because words are aligned on word boundaries, the low order two bits of a continuation pointer are always `0x0` (`(00000000)b`):

```
+================================+
|             raw address      |00|
+================================+
```

10.5.2 Catch Labels

A catch label is used to indicate a position in code to which to jump in a try-catch expression. The term occupies a single term, with the low order 6 bits having the value `0x1B`, the high order 8 bits holding the module index (`m_i`), and the middle 18 bits holding the catch label index (`l_i`):

```
|<   8  >|<      18        >|< 6  >|
+========+================+======+
|   m_i  |      l_i        |011011|
+========+================+======+
|                                 |
|<---------- word-size --------->|
```

Module and catch label indices are stored outside of the process heap and are outside of the scope of this document.

10.6 Garbage Collection

Garbage collection refers to the process of removing no-longer referenced term data stored in the heap, making room for new storage, as the program requires. AtomVM implements Tracing Garbage Collection, as does Erlang Garbage Collection. Unlike some garbage collection systems (e.g., as implemented by the Java Virtual Machine), garbage collection in Erlang-based systems, is performed independently on the heap allocated for each active Erlang process; there is no single shared heap for all running Erlang processes.

A given process heap and stack occupy a single region of malloc'd memory, and it is the job of the Erlang VM to manage memory within the allocated regions. Because this region is fixed, every allocation in the heap or stack results in less free space for the Erlang process. When free space reaches a limit, AtomVM will run a garbage collection event, which will allocate a new block of memory to hold the new heap and stack (the actual allocation depends on the heap growth strategy as explained above), and then copy terms from the old heap and stack to the new heap and stack. Any terms that no longer have references from term pointers in the old stack or registers are not copied to the new stack, and are therefore "collected" as garbage. In addition, any objects in the old heap that reference objects in shared memory (see reference counted binaries, above) are also managed as part of this process, in a manner described below.

```
                              +---------+ ------
                              |   new   |    ^
                              |   heap  |    |
                              +---------+    |
       ---- +----------+      |         |    |
        ^   |          |      |         |    |
        |   |   old    |      |         |    |  new
```

```
old     |   |   heap    |      |         |    |  malloc'd
malloc'd|   |          | ===> |         |    |  region
region  |   +==========+  gc  |  free   |    |
        |   |   old    |      |         |    |
        v   |   stack  |      |         |    |
       ---- +----------+      |         |    |
                              +---------+    |
                              |   new   |    |
                              |   stack |    v
                              +---------+ -----


+---+---+---+--------------------+---+
| 0 | 1 | 2 |                    | 15|
+---+---+---+--------------------+---+
registers


process dictionary
+--------+--------+
|   k1   |   v1   |
+--------+--------+
|   k2   |   v2   |
+--------+--------+
|       ...       |
```

Terms stored in the stack, registers, and process dictionary are either single-word terms (like atoms or pids) or term references, i.e., single-word terms that point to boxed terms or list cells in the heap. These terms constitute the "roots" of the memory graph of all "reachable" terms in the process.

10.6.1 When does garbage collection happen?

Garbage collection typically occurs as the result of a request for an allocation of a multi-word term in the heap (e.g., a tuple, list, or binary, among other types), and when there is currently insufficient space in the free space between the current heap and the current stack to accommodate the allocation.

Garbage collection is a *synchronous* operation in each Context (Erlang process), but conceptually no other execution contexts are impacted (i.e., no global locks, other than those required for memory allocation in the OS process heap).

10.6.2 Garbage Collection Steps

Garbage collection in AtomVM can be broken down into the following phases:

- Allocation of a new block of memory to store the new heap and stack;
- A "shallow copy" of all root terms (from the stack, registers, and process dictionary) into the heap, as well as updates to the references in the stack, registers, and process dictionary;
- An iterative "scan and copy" of the new heap, until all "live" terms are copied to the new heap;
- A sweep of the "Mark Sweep Object" list;
- Deletion of the old heap.

The following subsections describe these phases in more detail.

Allocation

Garbage collection typically occurs as the result of a request for space on an Erlang process's heap. The amount of space requested is dependent on the kind of term being allocated, but in general, AtomVM will check the amount of free space in the heap, and if it is below the amount of requested space plus some extra (currently, 16 words), then a garbage collection will occur, with the requested allocation space being the current size of the heap, plus the requested size, plus an extra 16 words.

Allocation is a straightforward `malloc` in the (operating system) process heap of the requested set of words. This block of storage will become the "new heap", as opposed to the existing, or "old heap".

Shallow Copy

The garbage collector starts by traversing the current root set, i.e., the terms contained in the stack, registers, and keys and values in the process dictionary, and performs a "shallow copy" of the terms that are in or referenced from these root terms from the old heap to the new heap, while at the same time updating the values in the root set, as some of these values may be pointers into the old heap, and therefore need to be updated to pointers in the new heap.

A shallow copy of a term depends on the type of the term being copied. If the term is a single-word term, like an atom or pid, then the term only resides in the root set, itself, and nothing needs to be copied from the old heap to the new heap. (The term *may* occur in the heap elsewhere, but as an element of another term, like a tuple, for example.)

On the other hand, if the term in the root set points to a boxed term in the old heap, then three things happen:

- The boxed term is copied from the old heap to the new heap. Note that if the term being copied contains pointers to other boxed terms in the old heap, the pointers are *not* updated (yet); they will be as part of the iterative scan and copy (see below);
- The first word of the existing boxed term that was copied is *over-written* with a marker value (`0x2b`) in the old heap, and the second word is over-written with the address of the copied boxed term in the new heap.
- The term in the root set is updated with the address of the copied boxed term in the new heap.

This process is best illustrated with a motivating example:

```
{foo, <<1,2,3,4,...,1024>>}
```

Suppose this term resides in the old heap, and some `register[i]` is a root term pointer to this tuple in the heap:

```
+-> |              |        |             |
|   |              |        |             |
|   |              |        |             |  USED
|   |              |        |             |
          ...                     ...
|   |              |        |             |
|   +--------------+        +=============+ <-- heap
|   |     tuple    |<---+   |             |     addr
|   +--------------+    |   |             |
|   |   atom foo   |    |   |             |
|   +--------------+    |   |             |
+----< refc binary |    |   |             |  FREE
    +--------------+    |   |             |
    |              |    |   |             |
    |     ...      |    |   |     ...     |
    |              |    |   |             |
        old heap        |       new heap
                        |
            ---+----------------+---
           ... |     old-ptr    | ...
            ---+----------------+---
                  register[i]
```

The boxed term is copied to the new heap, overwritten with the marked header `0x2b`, along with a pointer to the new term, and the root term is updated with the same address:

```
+-> |              | <--------+   |             |
|   |              |          |   |             |
|   |              |          |   |             |
|   |              |          |   |             |
           ...                          ...
|   |              |          |   |             |
|   +--------------+ >>>>>>>>>|>> +-------------+
|   |     0x2b     |   +--+---|-> |   tuple     | USED
|   +--------------+   |  |   |   +-------------+
|   |     ptr  --------+  |   |   |  atom foo   |
|   +--------------+      |   |   +-------------+
+----< refc binary |      |   +----< refc binary|
    +--------------+ >>>>>|>>>>>>>+=============+ <-- new
    |              |      | COPY  |             |   heap
    |     ...      |      |       |             |   addr
         old heap         |       |             |
                          |       |             | FREE
                          |
                          |
              ---+----------------+---
             ... |     new-ptr    | ...
              ---+----------------+---
                    register[i]
```

Note that the first term of the tuple (atom `foo`) is copied to the new heap, but the pointer to the refc binary is out of date – it still points to a value in the old heap. This will be corrected in the iterative scan and copy phase, below.

After a shallow copy of the root set, all terms immediately reachable from the root set have been copied to the new heap, and any boxed terms they reference have been marked as being moved. The new heap consists of a set of contiguous copied boxed terms from the old heap, starting from the base address of the heap, to some higher address in the heap, but less than or equal to the maximum heap size on the new heap.

Iterative Scan and Copy

The iterative scan and copy phase works as follows:

- Starting with the newly created region used in the shallow copy phase in the new heap, iterate over every term in the region (call this the "scan&copy" region);
- If any term in this region is a reference to a term on the old heap that has *not* been marked as copied, perform a shallow copy of it (as described above) to the new heap, but starting at the next free address below the region being iterated over;
- Note that after iterating over all such terms in the scan and copy region, all terms are "complete", in that there are no references to boxed terms in the old heap in that region. We have, however, created a new region which may have references to boxed terms in the old heap;
- So we repeat the process on the new region, which will complete the current scan&copy region, but which in turn may create a new region of copied terms;
- The process is repeated until no new regions have been introduced.

The following sequence of iterative additions to the new heap illustrates this process:

```
+---------------+ ===> +---------------+ ===> +---------------+ ...
|   scan&copy   |      |    complete   |      |    complete   |
|   region      |      |    region     |      |    region     |
|               |      |               |      |               |
|               |      |               |      |               |
|               |      |               |      |               |
|               |      |               |      |               |
|               |      |               |      |               |
+---------------+      +---------------+      +---------------+
                       |     newly     |      |   scan&copy   |
                       |    copied     |       |   region      |
                       |     terms     |       |               |
                       +---------------+       +---------------+
                                              |newlycpiedterms|
                                              +---------------+



 ... ===> +---------------+ ===> +---------------+
          |    complete   |      |    complete   |
          |    region     |      |    region     |
          |               |      |               |
          |               |      |               |
          |               |      |               |
          |               |      |               |
          |               |      |               |
          |               |      |               |
          |               |      |               |
          |               |      |               |
          |               |      |               |
          +---------------+      |               |
          |  scan&copy    |      |               |
          +---------------+      +---------------+
```

At the end of the iterative scan and copy, all reachable terms in the old heap will be copied to the new heap, and no boxed terms in the old heap will contain pointers to terms in the old heap. Any terms that have not been copied to the new heap are "garbage", as there are no longer any paths to them from the root set, and can therefore be destroyed,

MSO Sweep

As mentioned in the section above on binaries, AtomVM supports reference-counted binaries, whereby binaries of a sufficiently large size (>64 bytes) are allocated outside of the process heap, and are instead referenced from boxed terms in the heap. This way, binaries, which are immutable objects, can be shared between processes without incurring the time and space cost of a large data copy.

In order to manage the memory associated with such binaries, AtomVM tracks references to these off-heap binaries via the "Mark and Sweep Object" list, a list that keeps track of which boxed terms in the process heap have a reference to an off-heap binary. When such a boxed term is copied (e.g., from a heap to a mailbox on a `send`, or from a mailbox to a heap on a `receive`), the reference count is incremented on the off-heap binary.

The MSO list is formed via the cons cells that are appended to reference counted binary boxed terms in the process heap. The list is initially empty (nil), but as reference counted binaries are added to the process heap, they are pre-pended to the MSO list for the process (on the mailbox message, as reference-counted binaries in the mailbox need to be managed, as well).

The following diagram illustrates a set of two reference counted binaries in a process heap:

```
        |                   |
+-----> +-------------------+
|       |       refc        |
|       |      binary       |
|       |                   |
|       |                   |
|       +-------------------+ <----+
|       |        nil        |      |
|       +-------------------+      |
+---------------< car       |      |
        +-------------------+      |
        |                   |      |
                ...
        |                   |      |
+-----> +-------------------+      |
|       |       refc        |      |
|       |      binary       |      |
|       |                   |      |
|       |                   |      |
|       +-------------------+ <----|-------+
|       |        cdr >-------------+       |
|       +-------------------+              |
+--------------< car        |              |
        +-------------------+              |
        |                   |              |
        |                   |              |
                ...                        |
                                           |
                                   +-------^--------+
                                   |   mso_list     |
                                   +----------------+
```

After the new heap has been scanned and copied, as described above, the MSO list is traversed to determine if any reference-counted binaries are no longer referenced from the process heap. If any reference counted binaries in the heap have not been marked as moved from the old heap, they are, effectively, no longer referenced from the root set, and the reference count on the corresponding off-heap binary can be decremented. Furthermore, when the reference count reaches 0, the binaries can then be deleted.

Note Const binaries, while they have slots for entry into the MSO list, nonetheless are never "stitched" into the MSO list, as the binary data they point to is const, endures for the lifecycle of the program, and is never deleted. Match binaries, on the other hand, do count as references, and can therefore be stitched into the MSO list. However, when they are, the reference counted binaries they point to are the actual binaries in the process heap, not the match binaries, as with the case of refc binaries on the process heap.

Deletion

Once all terms have been copied from the old heap to the new heap, and once the MSO list has been swept for unreachable references, the old heap is simply discarded via the `free` function.

11 Packbeam Format

AtomVM makes use of the packbeam format for aggregating beam and other file types into a single file that is used as the code base for an AtomVM application. Typically, on an embedded device, packbeam files are uploaded (e.g., via serial connection) to a specific location on flash media. The AtomVM runtime will locate the entrypoint into the application, and use the beam and other files flashed to the local media to run the uploaded application.

AtomVM provides a simple tool for generating packbeam files, but other tools have emerged for manipulation packbeam files using standard Erlang and Elixir tool chains, notably `Mix` and `rebar3`.

This document describes the packbeam format, so that both AtomVM and upstream/downstream tooling have a reference document on which to base implementations.

11.1 Overview

Packbeam files are binary-encoded aggregations of BEAM and plain data files. At a high level, a packbeam file consists of a packbeam header, followed by a sequence of files (beam or otherwise), each of which is prefixed with a header, including data about the file (name, size, flags, etc).

All binary integer values are 32-bit, in network order (big-endian). Headers and encoded files are padded when necessary and aligned on 4-byte boundaries.

At present, the AtomVM runtime treats data in packbeam files as *read-only* data. There is no support for modifying the contents on an AtomVM file by the runtime.

11.2 Packbeam Header

All AtomVM files begin with the packbeam header, a fixed 24-byte sequence of octets:

```
0x23, 0x21, 0x2f, 0x75,
0x73, 0x72, 0x2f, 0x62,
0x69, 0x6e, 0x2f, 0x65,
0x6e, 0x76, 0x20, 0x41,
0x74, 0x6f, 0x6d, 0x56,
0x4d, 0x0a, 0x00, 0x00
```

The ASCII encoding of this sequence is

    #!/usr/bin/env AtomVM\n

followed by two nil (`0x00`) bytes.

The packbeam header is followed by a sequence of 0 or more encoded files. The number of files in a packbeam file is not indicated in the packbeam header; however, packbeam files do contain a special `end` file header, marking the end of the sequence of encoded files.

11.3 File encodings

Each embedded file in a packbeam file contains a file header, followed by the file contents.

11.3.1 File Header

The file header consists of the following 4 fields:

- `size` (32 bit, big-endian)

- `flags` (32-bit, big endian)
- `reserved` (32-bit, big-endian, currently unused)
- `module_name` (null-terminated sequence of bytes)

The `size` field indicates the size (in bytes) of the encoded file following the header. This size includes the file content length, in addition to any padding that may have been added to the file, in order for it to align on a 4-byte boundary.

Currently, the two low-order bits of the `flags` field are used. `0x02` indicates the file is a BEAM file, and `0x01` indicates that the file contains a `start/0` function, and is therefore suitable as an entry-point to start code execution.

When AtomVM starts, it will scan the BEAM files in the AtomVM file, from start to finish, with which it is initialized to find the entrypoint to start code execution. It will start execution on the first BEAM file with a `start/0` function, i.e., whose flags mask against `0x03`. It is conventional, but not required, for the first file in an AtomVM file to be a BEAM file that has a `start/0` entrypoint.

The `reserved` field is currently unused.

The `module_name` is variable length, null terminated sequence of characters. Because the module name is variable-length, the header may be padded with null characters (`0x00`), in order to align the start of the file contents on a 4-byte boundary.

11.3.2 Example

The following BEAM header indicates a BEAM file with a length of 308 bytes (`0x00000134`), with a `start/0` entrypoint (`0x00000003`), and named `mylib.beam` (`0x6D796C69 622E6265 616D00`). The header has a 1-byte padding of null (`0x00`) characters.

```
00000134 00000003 00000000 6D796C69 622E6265 616D0000
```

11.3.3 BEAM files

BEAM files obey IFF encoding as detailed here, but certain information in BEAM files is stripped out in order to minimize the amount of data stored on flash.

The following BEAM chunks are included in BEAM files:

- `AtU8`
- `Code`
- `ExpT`
- `LocT`
- `ImpT`
- `LitU`
- `FunT`
- `StrT`
- `LitT`

Any other chunks are stripped out of the BEAM files before insertion into AVM files.

In addition, data in the literals table (`LitT`) are uncompressed before insertion into AVM files, as the AtomVM runtime does not include support for `zlib` decompression.

BEAM files may be padded at the end with a sequence of 1-3 null (`0x00`) characters, in order to align on 4-byte boundaries.

Note The `module_name` field in the file header will only contain the "base" name of the BEAM file, i.e., the file name stripped of any path information.

11.3.4 Normal Files

Normal files (e.g., text files, data files, etc.) can be stored in packbeam AVM files, as well as BEAM files. For example, a normal file might contain static configuration information, or data that is interpreted at runtime.

Normal files contain a 32-bit big-endian size prefix, indicating the size of the file data (without padding). Note that the `size` field in the file header includes the size of the data with padding, if applicable.

The AtomVM runtime provides access to data files via the `atomvm:read_priv/2` NIF. This function will create a path name formed by the `App` (atom) and `Path` (string) terms provided by this function, separated by `"/priv/"`. For example, the expression

```
atomvm:read_priv(mylib, "sample.txt")
```

yields a `binary` containing the contents of `mylib/priv/sample.txt`, if it exists, in the AtomVM packbeam file.

As a consequence, normal files should be included in packbeam files using module names that obey the above patterns.

Tip Normal file names may encode virtual directory names, such as `mylib/priv/another/sample/text/file`. There is no requirement that the `Path` component of a normal file be a simple file name.

11.3.5 `end` file

Packbeam files end with a special `end` header. The `size` field of the `end` header is 0 bytes.

Example `end` header

The following sequence of bytes encodes the `end` header:

```
00000000 00000000 00000000 656E6400
```

Chapter 12

API Reference Documentation

12.1 Erlang Libraries

12.1.1 estdlib

The estdlib library

Modules

Module application

- Data Types
- Function Index
- Function Details

Data Types

start_type()

Function Index

Function Details

get_env/2

`Application`: application to get the parameter value of`Parameter`: parameter to get the value of

returns: undefined

Retrieve the value of the configuration parameter `Parameter` for application `Application` or `undefined` if not found.

get_env/3

`Application`: application to get the parameter value of`Parameter`: parameter to get the value of`Default`: default value if parameter is not found

returns: default value

Retrieve the value of the configuration parameter `Parameter` for application `Application` or `Default` if not found.

Module base64

- Description
- Function Index
- Function Details

An implementation of a subset of the Erlang/OTP base64 interface.

Description

This module is designed to be API-compatible with the Erlang/OTP base64 module, with the following exceptions:

- No support for decoding data with whitespace in base64 data
- No support for mime decoding functions

Function Index

Function Details

decode/1

`Data`: the data to decode

returns: the base-64 data decoded, as a binary

Base-64 decode a binary or string, outputting a binary.

This function will raise a badarg exception if the supplied data is not valid base64-encoded data.

decode_to_string/1

`Data`: the data to decode

returns: the base-64 data decoded, as a string

Base-64 decode a binary or string, outputting a string.

This function will raise a badarg exception if the supplied data is not valid base64-encoded data.

encode/1

`Data`: the data to encode

returns: the base-64 data encoded, as a binary

Base-64 encode a binary or string, outputting a binary.

encode_to_string/1

`Data`: the data to encode

returns: the base-64 data encoded, as a string

Base-64 encode a binary or string, outputting a string.

Module binary

- Description
- Function Index
- Function Details

An implementation of a subset of the Erlang/OTP binary interface.

Function Index

Function Details

at/2

`Binary`: binary to get a byte from`Index`: 0-based index of the byte to return

returns: value of the byte from the binary

Get a byte from a binary by index.

decode_hex/1

`Data`: hex encoded binary to decode

returns: decoded binary

Decodes a hex encoded binary into a binary.

encode_hex/1

`Data`: binary data to convert into hex encoded binary

returns: hex encoded binary

Encodes a binary into a hex encoded binary using the specified case for the hexadecimal digits "a" to "f".

encode_hex/2

`Data`: binary data to convert into hex encoded binary`Case`: which case to encode into

returns: hex encoded binary

Encodes a binary into a hex encoded binary using the specified case for the hexadecimal digits "a" to "f".

part/3

`Binary`: binary to extract a subbinary from`Pos`: 0-based index of the subbinary to extract`Len`: length, in bytes, of the subbinary to extract.

returns: a subbinary from Binary

Get the part of a given binary. A negative length can be passed to count bytes backwards.

split/2

`Binary`: binary to split`Pattern`: pattern to perform the split

returns: a list composed of one or two binaries

Equivalent to `split(Binary, Pattern, [])`.

Split a binary according to pattern. If pattern is not found, returns a singleton list with the passed binary. Unlike Erlang/OTP, pattern must be a binary.

split/3

`Binary`: binary to split`Pattern`: pattern to perform the split

returns: a list composed of one or two binaries

Split a binary according to pattern. If pattern is not found, returns a singleton list with the passed binary. Unlike Erlang/OTP, pattern must be a binary. Only implemented option is `global`

Module calendar

- Description
- Data Types
- Function Index
- Function Details

A partial implementation of the Erlang/OTP calendar functions.

Description

This module provides an implementation of a subset of the functionality of the Erlang/OTP calendar functions.

All dates conform to the Gregorian calendar. This calendar was introduced by Pope Gregory XIII in 1582 and was used in all Catholic countries from this year. Protestant parts of Germany and the Netherlands adopted it in 1698, England followed in 1752, and Russia in 1918 (the October revolution of 1917 took place in November according to the Gregorian calendar).

The Gregorian calendar in this module is extended back to year 0. For a given date, the gregorian day is the number of days up to and including the date specified.

Data Types

date()

datetime()

day()

day_of_week()

gregorian_days()

hour()

minute()

month()

second()

time()

year()

Function Index

Function Details

date_to_gregorian_days/1

`Date`: the date to get the gregorian day count of

returns: Days number of days

Equivalent to `date_to_gregorian_days(Year, M, D)`.

Year cannot be abbreviated.

For example, 93 denotes year 93, not 1993. The valid range depends on the underlying operating system. The date tuple must denote a valid date.

date_to_gregorian_days/3

`Year`: ending year`M`: ending month`D`: ending day

returns: Days number of days

Computes the number of gregorian days starting with year 0 and ending at the specified date.

datetime_to_gregorian_seconds/1

`DateTime`: the date and time to convert to seconds

returns: Seconds number of seconds

Computes the number of gregorian seconds starting with year 0 and ending at the specified date and time.

day_of_the_week/1

`Date`: the date for which to retrieve the weekday

returns: Weekday day of the week

Equivalent to `day_of_the_week(Y, M, D)`.

Computes the day of the week from the specified date tuple {Year, Month, Day}. Returns the day of the week as 1: Monday, 2: Tuesday, and so on.

day_of_the_week/3

`Y`: year of the desired day`M`: month of the desired day`D`: year of the desired day

returns: Weekday day of the week

Computes the day of the week from the specified Year, Month, and Day. Returns the day of the week as 1: Monday, 2: Tuesday, and so on.

system_time_to_universal_time/2

`Time`: the time, as an integer, in the specified unit`TimeUnit`: the time unit

returns: DateTime The date and time (in UTC) converted from the specified time and time unit

Convert an integer time value to a date and time in UTC.

Module code

- Description
- Function Index
- Function Details

An implementation of a subset of the Erlang/OTP code interface.

Function Index

Function Details

all_available/0

returns: A list of available modules, including loaded modules

Return all modules available from loaded avm packs, in addition to loaded modules. List of available modules may be incomplete if this function is called while a module is loaded. Result type differs from Erlang/OTP: names of modules is a binary (and not a string), and second term of tuples is currently unspecified

all_loaded/0

returns: A list of all loaded modules

Return a list of all loaded modules. Result type differs from Erlang/OTP: second term of tuples is currently unspecified

ensure_loaded/1

`Module`: module to load

returns: Tuple `{module, Module}` if module is loaded or `{error, embedded}`

Try to load a module if it's not already loaded. AtomVM works in an embedded-like mode where modules are loaded at start-up but modules can be loaded explicitely as well (especially from a binary with `load_binary/3`). So this function can be used to determine if a module is loaded. It is called by Elixir Code module.

is_loaded/1

`is_loaded(Module) -> any()`

Determine if a module is loaded. AtomVM works in an embedded-like mode where modules are loaded at start-up but modules can be loaded explicitely as well (especially from a binary with `load_binary/3`).

load_abs/1

`Filename`: path to the beam to open, without .beam suffix

returns: A tuple with the name of the module

Load a module from a path. Error return result type is different from Erlang/OTP.

load_binary/3

`Module`: name of the module to load`Filename`: path to the beam (unused)`Binary`: binary of the module to load

returns: A tuple with the name of the module

Load a module from a binary. Error return result type is different from Erlang/OTP. Also unlike

Erlang/OTP, no check is performed to verify that `Module` matches the name of the loaded module.

which/1

`which(Module) -> any()`

Determine if a module is loaded. There currently is no way to distinguish a module that was loaded with `load_binary/3` or that was preloaded at startup.

Module crypto

- Data Types
- Function Index
- Function Details

Data Types

cipher_iv()

cipher_no_iv()

crypto_opt()

crypto_opts()

digest()

hash_algorithm()

padding()

Function Index

Function Details

crypto_one_time/4

`Cipher`: a supported cipher`Key`: the encryption / decryption key`Data`: to be crypted or encrypted`FlagOrOptions`: either just true for encryption (or false for decryption), or a proplist for any additional option

returns: Returns crypted or encrypted data.

Encrypted/decrypt data using given cipher and key

crypto_one_time/5

`Cipher`: a supported cipher that makes use of IV`Key`: the encryption / decryption key`IV`: an initialization vector`Data`: to be crypted or encrypted`FlagOrOptions`: either just true for encryption (or false for decryption), or a proplist for any additional option such as padding.

returns: Returns crypted or encrypted data.

Encrypted/decrypt data using given cipher, key, IV.

hash/2

`Type`: the hash algorithm`Data`: the data to hash

returns: Returns the result of hashing the supplied data using the supplied hash algorithm.

Hash data using a specified hash algorithm.

strong_rand_bytes/1

`N`: desired length of cryptographically secure random data

returns: Returns Cryptographically secure random data of length `N`

Generate N cryptographically secure random octets and return the result in a binary.

Module dist_util

- Function Index

- Function Details

Function Index

Function Details

cancel_timer/1

handshake_other_started/1

handshake_we_started/1

net_ticker_spawn_options/0

reset_timer/1

shutdown/3

shutdown/4

start_timer/1

Module erl_epmd

- Description
- Function Index
- Function Details

An implementation of the Erlang/OTP erl_epmd interface.

Behaviours: `gen_server`.

Description

This module implements a strict subset of the Erlang/OTP erl_epmd interface.

Function Index

Function Details

names/1

`Host`: the host to connect to

returns: a list of names and ports of registered nodes

Get the names and ports of all registered nodes This function will connect to epmd on localhost.

port_please/2

`Name`: name of the node to get the port of`Host`: host of the node to get the port of

Get the port and version of a node on a given host. This function will connect to epmd on the host.

register_node/2

`Name`: name to register`Port`: port to register

Register to local epmd and get a creation number

start_link/0

Start EPMD client

stop/0

Stop EPMD client-

Module erlang

- Description

An implementation of the Erlang/OTP erlang module, for functions that are not already defined as NIFs.

Data Types

atom_encoding()

demonitor_option()

float_format_option()

heap_growth_strategy()

mem_type()

send_destination()

spawn_option()

time_unit()

timestamp()

Function Index

Function Details

apply/2

`Function`: Function to call`Args`: Parameters to pass to function (max 6)

returns: Returns the result of Function(Args).

Returns the result of applying Function to Args. The arity of the function is the length of Args. Example:

```
    > apply(fun(R) -> lists:reverse(R) end, [[a, b, c]]).
  [c,b,a]
  > apply(fun erlang:atom_to_list/1, ['AtomVM']).
  "AtomVM"
```

If the number of arguments are known at compile time, the call is better written as Function(Arg1, Arg2, …, ArgN).

apply/3

`Module`: Name of module`Function`: Exported function name`Args`: Parameters to pass to function (max 6)

returns: Returns the result of Module:Function(Args).

Returns the result of applying Function in Module to Args. The applied function must be exported from Module. The arity of the function is the length of Args. Example:

```
    > apply(lists, reverse, [[a, b, c]]).
  [c,b,a]
  > apply(erlang, atom_to_list, ['AtomVM']).
  "AtomVM"
```

If the number of arguments are known at compile time, the call is better written as Module:Function(Arg1, Arg2, …, ArgN).

atom_to_binary/1

`Atom`: Atom to convert

returns: a binary with the atom's name

Convert an atom to a binary, defaults to utf8. Only latin1 encoding is supported.

atom_to_binary/2

`Atom`: Atom to convert`Encoding`: Encoding for conversion (any of latin1, utf8 or unicode)

returns: a binary with the atom's name

Convert an atom to a binary. Only latin1 encoding is supported.

atom_to_list/1

`Atom`: Atom to convert

returns: a string with the atom's name

Convert an atom to a string.

binary_to_atom/1

`Binary`: Binary to convert to atom

returns: an atom from passed binary

Convert a binary to atom, defaults to utf8.

binary_to_atom/2

`Binary`: Binary to convert to atom`Encoding`: encoding for conversion (any of latin1, utf8 or unicode)

returns: an atom from passed binary

Convert a binary to atom.

binary_to_integer/1

`Binary`: Binary to parse for integer

returns: the integer represented by the binary

Parse the text in a given binary as an integer.

binary_to_integer/2

`Binary`: Binary to parse for integer

returns: the integer represented by the binary

Parse the text in a given binary as an integer.

binary_to_list/1

`Binary`: Binary to convert to list

returns: a list of bytes from the binary

Convert a binary to a list of bytes.

binary_to_term/1

`Binary`: binary to decode

returns: A term decoded from passed binary

Decode a term that was previously encodes with `term_to_binary/1` This function should be mostly compatible with its Erlang/OTP counterpart. Unlike modern Erlang/OTP, resources are currently serialized as empty binaries and cannot be unserialized.

demonitor/1

`Monitor`: reference of monitor to remove

returns: `true`

Remove a monitor

demonitor/2

`Monitor`: reference of monitor to remove`Options`: options list returns: `true`

Remove a monitor, with options. If `flush`, monitor messages are flushed and guaranteed to not be received. If `info`, return `true` if monitor was removed, `false` if it was not found. If both options are provivded, return `false` if flush was needed.

display/1

`Term`: term to print

returns: `true`

Print a term to stdout.

erase/0

returns: the previous process dictionary.

Erase all keys from the process dictionary.

erase/1

`Key`: key to erase from the process dictionary

returns: the previous value associated with this key or undefined

Erase a key from the process dictionary.

exit/1

`Reason`: reason for exit

Raises an exception of class `exit` with reason `Reason`. The exception can be caught. If it is not, the process exits. If the exception is not caught the signal is sent to linked processes. In this case, if `Reason` is `kill`, it is not transformed into `killed` and linked processes can trap it (unlike `exit/2`).

exit/2

`Process`: target process`Reason`: reason for exit

returns: `true`

Send an exit signal to target process. The consequences of the exit signal depends on `Reason`, on whether `Process` is self() or another process and whether target process is trapping exit. If `Reason` is not `kill` nor `normal`:

- If target process is not trapping exits, it exits with `Reason`
- If traget process is trapping exits, it receives a message `{'EXIT', From, Reason}` where `From` is the caller of `exit/2`.

If `Reason` is `kill`, the target process exits with `Reason` changed to `killed`. If `Reason` is `normal` and `Process` is not `self()`:

- If target process is not trapping exits, nothing happens.
- If traget process is trapping exits, it receives a message `{'EXIT', From, normal}` where `From` is the caller of `exit/2`.

If `Reason` is `normal` and `Process` is `self()`:

- If target process is not trapping exits, it exits with `normal`.
- If traget process is trapping exits, it receives a message `{'EXIT', From, normal}` where `From` is the caller of `exit/2`.

float_to_binary/1

`Float`: Float to convert

returns: a binary with a text representation of the float

Convert a float to a binary.

float_to_binary/2

`Float`: Float to convert`Options`: Options for conversion

returns: a binary with a text representation of the float

Convert a float to a binary.

float_to_list/1

`Float`: Float to convert

returns: a string with a text representation of the float

Convert a float to a string.

float_to_list/2

`Float`: Float to convert`Options`: Options for conversion

returns: a string with a text representation of the float

Convert a float to a string.

fun_info/1

`fun_info(Fun) -> any()`

```
        Returns information about the function `Fun` in unspecified order.
```

fun_info/2

`fun_info(Fun, Info) -> any()`

```
        Returns information about the function `Fun` in unspecified order.
```

fun_to_list/1

`Fun`: function to convert to a string

returns: a string representation of the function

Create a string representing a function.

function_exported/3

`Module`: module to test`Function`: function to test`Arity`: arity to test

returns: `true` if Module exports a Function with this Arity

Determine if a function is exported

garbage_collect/0

returns: `true`

Run a garbage collect in current process

garbage_collect/1

`Pid`: pid of the process to garbage collect

returns: `true` or `false` if the process no longer exists

Run a garbage collect in a given process. The function returns before the garbage collect actually happens.

get/0

returns: the process directory

Return all values from the process dictionary *get/1*

`Key`: key in the process dictionary

returns: value associated with this key or undefined

Return a value associated with a given key in the process dictionary

get_cookie/0

returns: The cookie used by this node

Return the cookie used by this node or `nocookie` if distribution was not started

get_cookie/1

returns: The cookie used to connect to another node

Return the cookie used to connect to another node or `nocookie` if distribution was not started

get_module_info/1

`Module`: module to get info for

returns: A list of module info tuples

Get info for a given module. This function is not meant to be called directly but through `Module:module_info/0` exported function.

get_module_info/2

`Module`: module to get info for`InfoKey`: info to get

returns: A term representing info for given module

Get specific info for a given module. This function is not meant to be called directly but through `Module:module_info/1` exported function. Supported info keys are `module`, `exports`, `compile` and `attributes`.

group_leader/0

returns: Pid of group leader or self() if no group leader is set.

Return the pid of the group leader of caller.

group_leader/2

`Leader`: pid of process to set as leader`Pid`: pid of process to set a Leader

returns: `true`

Set the group leader for a given process.

integer_to_binary/1

`Integer`: integer to convert to a binary

returns: a binary with a text representation of the integer

Convert an integer to a binary.

integer_to_binary/2

`Integer`: integer to convert to a binary`Base`: base for representation

returns: a binary with a text representation of the integer

Convert an integer to a binary.

integer_to_list/1

`Integer`: integer to convert to a string

returns: a string representation of the integer

Convert an integer to a string.

integer_to_list/2

`Integer`: integer to convert to a string`Base`: base for representation

returns: a string representation of the integer

Convert an integer to a string.

iolist_size/1

`IOList`: IO list to compute the binary size of

returns: the number of bytes of IOList if it was convered to `binary()`

Equivalent to `byte_size(iolist_to_binary(IOList))`.

Compute the length in bytes of the IO list or `binary()`

iolist_to_binary/1

`IOList`: IO list to convert to binary

returns: a binary with the bytes of the IO list

Convert an IO list to binary.

is_alive/0

returns: true if current node is part of distribution

Determine if current node is part of distribution

is_map/1

`Map`: the map to test

returns: `true` if `Map` is a map; `false`, otherwise.

Return `true` if `Map` is a map; `false`, otherwise.

This function may be used in a guard expression.

is_map_key/2

`Key`: the key`Map`: the map

returns: `true` if `Key` is associated with a value in `Map`; `false`, otherwise.

Return `true` if `Key` is associated with a value in `Map`; `false`, otherwise.

This function raises a `{badmap, Map}` error if `Map` is not a map.

This function may be used in a guard expression.

is_process_alive/1

`Pid`: pid of the process to test

returns: `true` if the process is alive, `false` otherwise

Determine if a process is alive

is_record/2

`RecordTag`: atom representing tuple tag

Returns true if Term is a tuple and its first element is RecordTag, false otherwise.

link/1

`Pid`: process to link to

returns: `true`

Link current process with a given process.

list_to_atom/1

`String`: string to convert to an atom

returns: an atom from the string

Convert a string into an atom. Unlike Erlang/OTP 20+, atoms are limited to ISO-8859-1 characters. The VM currently aborts if passed unicode characters. Atoms are also limited to 255 characters. Errors with system_limit_atom if the passed string is longer.

See also: list_to_existing_atom/1.

list_to_binary/1

`IOList`: iolist to convert to binary

returns: a binary composed of bytes and binaries from the list

Convert a list into a binary. Errors with `badarg` if the list is not an iolist.

list_to_existing_atom/1

`String`: string to convert to an atom

returns: an atom from the string

Convert a string into an atom. This function will error with badarg if the atom does not exist

See also: list_to_atom/1.

list_to_integer/1

`String`: string to convert to integer

returns: an integer value from its string representation

Convert a string (list of characters) to integer. Errors with `badarg` if the string is not a representation of an integer.

list_to_integer/2

`String`: string to convert to integer`Base`: string to convert to integer

returns: an integer value from its string representation

Convert a string (list of characters) to integer in specified base. Errors with `badarg` if the string is not a representation of an integer or the base is out of bounds.

list_to_tuple/1

`List`: list to convert to tuple

returns: a tuple with elements of the list

Convert a list to a tuple with the same size.

localtime/0

returns: A tuple representing the current local time.

Return the current time and day for system local timezone.

See also: universaltime/0.

make_ref/0

returns: a new reference

Create a new reference

map_get/2

`Key`: the key to get`Map`: the map from which to get the value

returns: the value in `Map` associated with `Key`, if it exists.

Get the value in `Map` associated with `Key`, if it exists.

This function raises a `{badkey, Key}` error if 'Key' does not occur in `Map` or a `{badmap, Map}` if `Map` is not a map.

This function may be used in a guard expression.

map_size/1

`Map`: the map

returns: the size of the map

Returns the size of (i.e., the number of entries in) the map

This function raises a `{badmap, Map}` error if `Map` is not a map.

This function may be used in a guard expression.

max/2

`A`: any term`B`: any term

returns: `A` if `A > B`; `B`, otherwise.

Return the maximum value of two terms

Terms are compared using `>` and follow the ordering principles defined in https://www.erlang.org/doc/reference_manual/expressions.html#term-comparisons

md5/1

`Data`: data to compute hash of, as a binary.

returns: the md5 hash of the input Data, as a 16-byte binary.

Computes the MD5 hash of an input binary, as defined by https://www.ietf.org/rfc/rfc1321.txt

memory/1

`Type`: the type of memory to request

returns: the amount of memory (in bytes) used of the specified type

Return the amount of memory (in bytes) used of the specified type

min/2

`A`: any term`B`: any term

returns: `A` if `A < B`; `B`, otherwise.

Return the minimum value of two terms

Terms are compared using `<` and follow the ordering principles defined in https://www.erlang.org/doc/reference_manual/expressions.html#term-comparisons

monitor/2

`Type`: type of monitor to create

`Type`: type of monitor to create

returns: a monitor reference

Create a monitor on a process or on a port. When the process or the port terminates, the following message is sent to the caller of this function:

```
{'DOWN', MonitorRef, Type, Pid, Reason}
```

Unlike Erlang/OTP, monitors are only supported for processes and ports.

monotonic_time/1

`Unit`: time unit

returns: monotonic time in the specified units

Return the monotonic time in the specified units.

Monotonic time varies from system to system, and should not be used to determine, for example the wall clock time.

Instead, monotonic time should be used to compute time differences, where the function is guaranteed to return a (not necessarily strictly) monotonically increasing value.

For example, on ESP32 system, monotonic time is reported as the difference from the current time and the time the ESP32 device was started, whereas on UNIX systems the value may vary among UNIX systems (e.g., Linux, macOS, FreeBSD).

open_port/2

`PortName`: Tuple {spawn, Name} identifying the port`Options`: Options, meaningful for the port

returns: A pid identifying the open port

Open a port. Unlike Erlang/OTP, ports are identified by pids.

pid_to_list/1

`Pid`: pid to convert to a string

returns: a string representation of the pid

Create a string representing a pid.

port_to_list/1

`Port`: port to convert to a string

returns: a string representation of the port

Create a string representing a port.

process_flag/2

`Flag`: flag to change`Value`: new value of the flag

returns: Previous value of the flag

Set a flag for the current process. When `trap_exit` is true, exit signals are converted to messages

```
{'EXIT', From, Reason}
```

and the process does not exit if `Reason` is not `normal`.

process_info/2

`Pid`: the process pid.`Key`: key used to find process information.

`Pid`: the process pid.`Key`: key used to find process information.

`Pid`: the process pid.`Key`: key used to find process information.

`Pid`: the process pid.`Key`: key used to find process information.

`Pid`: the process pid.`Key`: key used to find process information.

`Pid`: the process pid.`Key`: key used to find process information.

`Pid`: the process pid.`Key`: key used to find process information.

returns: process information for the specified pid defined by the specified key.

Return process information.

This function returns information about the specified process. The type of information returned is dependent on the specified key.

The following keys are supported:

- **heap_size** the number of words used in the heap (integer), including the stack but excluding fragments
- **total_heap_size** the number of words used in the heap (integer) including fragments
- **registered_name** - returns `{registered_name, RegisteredName}` where `RegisteredName` is the registered name of the port or process. If the port/process has no registered name, `[]` is returned

- **stack_size** the number of words used in the stack (integer)
- **message_queue_len** the number of messages enqueued for the process (integer)
- **memory** the estimated total number of bytes in use by the process (integer)
- **links** the list of linked processes

Specifying an unsupported term or atom raises a bad_arg error.

processes/0

returns: A list of pids of all processes

Return a list of all current processes. Compared to Erlang/OTP, this function also returns native processes (ports).

put/2

`Key`: key to add to the process dictionary`Value`: value to store in the process dictionary

returns: the previous value associated with this key or undefined

Store a value with a given key in the process dictionary.

ref_to_list/1

`Ref`: reference to convert to a string

returns: a string representation of the reference

Create a string representing a reference.

register/2

`Name`: name of the process to register`PidOrPort`: pid or port of the process to register

returns: `true`

Register a name for a given process. Errors with `badarg` if the name is already registered or if the process is already registered.

send/2

`Message`: message to send

returns: the sent message

Send a message to a given process

send_after/3

`Time`: time in milliseconds after which to send the message.`Dest`: Pid or server name to which to send the message.`Msg`: Message to send to Dest after Time ms.

returns: a reference that can be used to cancel the timer, if desired.

Send Msg to Dest after Time ms.

set_cookie/1

`Cookie`: the cookie to use to connect to this node

Set the cookie of this node. Fails if distribution was not started.

set_cookie/2

`Node`: the node to set the cookie for`Cookie`: the cookie to use to connect to Node

Set the cookie of a given node. Fails if distribution was not started.

spawn/1

`Function`: function to create a process from

returns: pid of the new process

Create a new process

spawn/3

`Module`: module of the function to create a process from`Function`: name of the function to create a process from`Args`: arguments to pass to the function to create a process from

returns: pid of the new process

Create a new process by calling exported Function from Module with Args.

spawn_link/1

`Function`: function to create a process from

returns: pid of the new process

Create a new process and link it.

spawn_link/3

`Module`: module of the function to create a process from`Function`: name of the function to create a process from`Args`: arguments to pass to the function to create a process from

returns: pid of the new process

Create a new process by calling exported Function from Module with Args and link it.

spawn_monitor/1

`Function`: function to create a process from

returns: pid of the new process

Create a new process and monitor it.

spawn_monitor/3

`Module`: module of the function to create a process from`Function`: name of the function to create a process from`Args`: arguments to pass to the function to create a process from

returns: pid of the new process

Create a new process by calling exported Function from Module with Args and monitor it.

spawn_opt/2

`Function`: function to create a process from`Options`: additional options.

returns: pid of the new process

Create a new process.

spawn_opt/4

`Module`: module of the function to create a process from`Function`: name of the function to create a process from`Args`: arguments to pass to the function to create a process from`Options`: additional options.

returns: pid of the new process

Create a new process by calling exported Function from Module with Args.

split_binary/2

`Bin`: binary to split`Pos`: position to split the binary, from 0 to `byte_size(Bin)`

returns: A tuple with two subbinaries

Split a binary into two sub-binaries. This operation is not destructive and will create two new binaries.

start_timer/3

`Time`: time in milliseconds after which to send the timeout message.`Dest`: Pid or server name to which to send the timeout message.`Msg`: Message to send to Dest after Time ms.

returns: a reference that can be used to cancel the timer, if desired.

Start a timer, and send {timeout, TimerRef, Msg} to Dest after Time ms, where TimerRef is the reference returned from this function.

system_flag/2

`Key`: key used to change system flag.`Value`: value to change

returns: previous value of the flag.

Update system flags.

This function allows to modify system flags at runtime.

The following key is supported on SMP builds:

- **schedulers_online** the number of schedulers online

Specifying an unsupported atom key will result in a bad_arg error. Specifying a term that is not an atom will result in a bad_arg error.

system_info/1

`Key`: key used to find system information.

returns: system information defined by the specified key.

Return system information.

This function returns information about the system on which AtomVM is running. The type of information returned is dependent on the specified key.

The following keys are supported on all platforms:

- **process_count** the number of processes running in the node (integer)
- **port_count** the number of ports running in the node (integer)
- **atom_count** the number of atoms currently allocated (integer)
- **system_architecture** the processor and OS architecture (binary)
- **version** the version of the AtomVM executable image (binary)
- **wordsize** the number of bytes in a machine word on the current platform (integer)
- **schedulers** the number of schedulers, equal to the number of online processors (integer)
- **schedulers_online** the current number of schedulers (integer)

The following keys are supported on the ESP32 platform:

- **esp32_free_heap_size** the number of (noncontiguous) free bytes in the ESP32 heap (integer)
- **esp32_largest_free_block** the number of the largest contiguous free bytes in the ESP32 heap (integer)
- **esp32_minimum_free_size** the smallest number of free bytes in the ESP32 heap since boot (integer)
- **esp32_chip_info** Details about the model and capabilities of the ESP32 device (map)

Additional keys may be supported on some platforms that are not documented here.

Specifying an unsupported atom key will results in returning the atom 'undefined'.

Specifying a term that is not an atom will result in a bad_arg error.

system_time/1

`Unit`: Unit to return system time in

returns: An integer representing system time

Get the current system time in provided unit.

term_to_binary/1

`Term`: term to encode

returns: A binary encoding passed term.

Encode a term to a binary that can later be decoded with `binary_to_term/1`. This function should be mostly compatible with its Erlang/OTP counterpart. Unlike modern Erlang/OTP, resources are currently serialized as empty binaries.

timestamp/0

returns: A tuple representing the current timestamp.

Return the timestamp in `{MegaSec, Sec, MicroSec}` format. This the old format returned by `erlang:now/0`. Please note that the latter which is deprecated in Erlang/OTP is not implemented by AtomVM.

See also: monotonic_time/1, system_time/1.

unique_integer/0

returns: A unique integer

Same as erlang:unique_integer([]).

unique_integer/1

`Options`: list of options.

returns: a unique integer

Return a unique integer. If positive is passed, returned integer is positive. If monotonic is passed, returned integer is monotonically increasing across all processes.

universaltime/0

returns: A tuple representing the current universal time.

Return the current time and day for UTC.

See also: localtime/0.

unlink/1

`Pid`: process to unlink from

returns: `true`

Unlink current process from a given process.

unregister/1

`Name`: name to unregister

returns: `true`

Lookup a process by name. Unlike Erlang/OTP, ports are not distinguished from processes. Errors with `badarg` if the name is not registered.

whereis/1

`Name`: name of the process to locate

returns: `undefined` or the pid of the registered process

Lookup a process by name.

Module erpc

- Description
- Function Index

- Function Details

An implementation of the Erlang/OTP erpc interface.

Description

This module implements a strict subset of the Erlang/OTP erpc interface.

Function Index

Function Details

execute_call/4

`Reference`: reference of the request, passed in exit tuple`Module`: module to call`Func`: function to call`Args`: argument of the call

Execute a call locally, exiting with the result. This function is called from rpc on other nodes using spawn_request BIF.

Module erts_debug

- Description
- Function Index
- Function Details

An implementation of a subset of the Erlang/OTP erts_debug interface.

Function Index

Function Details

flat_size/1

`Term`: term to get the size of

returns: A size

Return the size, in terms, of a given term.

Module ets

- Description
- Data Types
- Function Index
- Function Details

A limited implementation of the Erlang/OTP `ets` module.

Data Types

access_type()

option()

options()

table()

abstract datatype: `table()`

table_type()

Function Index

Function Details

delete/1

`Table`: a reference to the ets table

returns: true;

Delete an ets table.

delete/2

`Table`: a reference to the ets table`Key`: the key used to lookup one or more entries to delete

returns: true; otherwise, an error is raised if arguments are bad

Delete an entry from an ets table.

insert/2

`Table`: a reference to the ets table`Entry`: the entry to insert

returns: true; otherwise, an error is raised if arguments are bad

Insert an entry into an ets table.

lookup/2

`Table`: a reference to the ets table`Key`: the key used to lookup one or more entries

returns: the entry in a set, or a list of entries, if the table permits

Look up an entry in an ets table.

lookup_element/3

`Table`: a reference to the ets table`Key`: the key used to lookup one or more entries`Pos`: index of the element to retrieve (1-based)

returns: the Pos:nth element of entry in a set, or a list of entries, if the table permits

Look up an element from an entry in an ets table.

new/2

`Name`: the ets table name`Options`: the options used to create the table

returns: A new ets table

Create a new ets table.

update_counter/3

`Table`: a reference to the ets table`Key`: the key used to look up the entry expecting to contain a tuple of integers or a single integer`Params`: the increment value or a tuple {Pos, Increment} or {Pos, Increment, Treshold, SetValue}, where Pos is an integer (1-based index) specifying the position in the tuple to increment. Value is clamped to SetValue if it exceeds Threshold after update.

returns: the updated element's value after performing the increment, or the default value if applicable

Updates a counter value at Key in the table. If Params is a single integer, it increments the direct integer value at Key or the first integer in a tuple. If Params is a tuple {Pos, Increment}, it increments the integer at the specified position Pos in the tuple stored at Key.

update_counter/4

`Table`: a reference to the ets table`Key`: the key used to look up the entry expecting to contain a tuple of integers or a single integer`Params`: the increment value or a tuple {Pos, Increment} or {Pos, Increment, Treshold, SetValue}, where Pos is an integer (1-based index) specifying the position in the tuple to increment. If after incrementation value exceeds the Treshold, it is set to SetValue.`Default`: the default value used if the entry at Key doesn't exist or doesn't contain a valid tuple with a sufficient size or integer at Pos

returns: the updated element's value after performing the increment, or the default value if applicable

Updates a counter value at Key in the table. If Params is a single integer, it increments the direct integer value at Key or the first integer in a tuple. If Params is a tuple {Pos, Increment}, it increments the integer at the specified position Pos in the tuple stored at Key. If the needed element does not exist, uses Default value as a fallback.

Module gen_event

- Function Index
- Function Details

Function Index

Function Details

add_handler/3

`add_handler(EventMgrRef, Handler, Args) -> any()`

delete_handler/3

`delete_handler(EventMgrRef, Handler, Args) -> any()`

notify/2

`notify(EventMgrRef, Event) -> any()`

start/0

`start() -> any()`

start/2

`start(EventMgrName, Options) -> any()`

start_link/0

`start_link() -> any()`

start_link/2

`start_link(EventMgrName, Options) -> any()`

stop/1

`stop(EventManagerRef) -> any()`

sync_notify/2

`sync_notify(EventMgrRef, Event) -> any()`

Module gen_server

- Description
- Data Types
- Function Index
- Function Details

An implementation of the Erlang/OTP gen_server interface.

This module defines the `gen_server` behaviour. Required callback functions: `init/1`, `handle_call/3`, `handle_cast/2`.

Description

This module implements a strict subset of the Erlang/OTP gen_server interface, supporting operations for local creation and management of gen_server instances.

This module is designed to be API-compatible with gen_server, with exceptions noted below.

Caveats:

- Support only for locally named procs
- No support for abcast

- No support for enter_loop
- No support for format_status
- No support for multi_call

Data Types

from()

options()

server_ref()

Function Index

Function Details

call/2

Equivalent to `call(ServerRef, Request, 5000)`.

Send a request to a gen_server instance, and wait for a reply.

call/3

`ServerRef`: a reference to the gen_server acquired via start`Request`: the request to send to the gen_server`TimeoutMs`: the amount of time in milliseconds to wait for a reply

returns: the reply sent back from the gen_server; {error, Reason}, otherwise.

Send a request to a gen_server instance, and wait for a reply.

This function will send the specified request to the specified gen_server instance, and wait at least Timeout milliseconds for a reply from the gen_server.

cast/2

`ServerRef`: a reference to the gen_server acquired via start`Request`: the request to send to the gen_server

returns: ok | {error, Reason}

Send a request to a gen_server instance.

This function will send the specified request to the specified gen_server instance, but will not wait for a reply.

init_it/4

`init_it(Starter, Module, Args, Options) -> any()`

init_it/5

`init_it(Starter, Name, Module, Args, Options) -> any()`

reply/2

`From`: the client to whom to send the reply`Reply`: the reply to send to the client

returns: an arbitrary term, that should be ignored

Send a reply to a calling client.

This function will send the specified reply back to the specified gen_server client (e.g, via call/3). The return value of this function can be safely ignored.

start/3

`Module`: the module in which the gen_server callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_server

returns: the gen_server pid, if successful; {error, Reason}, otherwise.

Start an un-named gen_server.

This function will start a gen_server instance.

Note. *The Options argument is currently ignored.*

start/4

`ServerName`: the name with which to register the gen_server`Module`: the module in which the gen_server callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_server

returns: the gen_server pid, if successful; {error, Reason}, otherwise.

Start a named gen_server.

This function will start a gen_server instance and register the newly created process with the process registry. Subsequent calls may use the gen_server name, in lieu of the process id.

Note. *The Options argument is currently ignored.*

start_link/3

`Module`: the module in which the gen_server callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_server

returns: the gen_server pid, if successful; {error, Reason}, otherwise.

Start and link an un-named gen_server.

This function will start a gen_server instance.

Note. *The Options argument is currently ignored.*

start_link/4

`ServerName`: the name with which to register the gen_server`Module`: the module in which the gen_server callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_server

returns: the gen_server pid, if successful; {error, Reason}, otherwise.

Start and link a named gen_server.

This function will start a gen_server instance and register the newly created process with the process registry. Subsequent calls may use the gen_server name, in lieu of the process id.

Note. *The Options argument is currently ignored.*

start_monitor/3

`Module`: the module in which the gen_server callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_server

returns: the gen_server pid and monitor reference tuple if successful; {error, Reason}, otherwise.

Start and monitor an un-named gen_server.

This function will start a gen_server instance.

Note. *The Options argument is currently ignored.*

start_monitor/4

`ServerName`: the name with which to register the gen_server`Module`: the module in which the gen_server callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_server

returns: the gen_server pid and monitor reference tuple if successful; {error, Reason}, otherwise.

Start and monitor a named gen_server.

This function will start a gen_server instance and register the newly created process with the process registry. Subsequent calls may use the gen_server name, in lieu of the process id.

Note. *The Options argument is currently ignored.*

stop/1

Equivalent to `stop(ServerRef, normal, infinity)`.

Stop a previously started gen_server instance.

stop/3

`ServerRef`: a reference to the gen_server acquired via start`Reason`: reason to be supplied to callback function`Timeout`: ms to wait for successful stop

returns: ok, if the gen_server stopped; {error, Reason}, otherwise.

Stop a previously started gen_server instance.

This function will stop a gen_server instance, providing the supplied Reason to the gen_server's terminate/2 callback function. If the gen_server is named, then the gen_server name may be used to stop the gen_server.

Module gen_statem

- Description
- Data Types
- Function Index
- Function Details

An implementation of the Erlang/OTP gen_statem interface.

This module defines the `gen_statem` behaviour. Required callback functions: `init/1`, `callback_mode/0`.

Description

This module implements a strict subset of the Erlang/OTP gen_statem interface, supporting operations for local creation and management of gen_statem instances.

This module is designed to be API-compatible with gen_statem, with exceptions noted below.

Caveats:

- No support for start_link
- Support only for locally named gen_statem instances
- Support only for state function event handlers
- No support for keep_state or repeat_state return values from Module:StateName/3 callbacks
- No support for postpone or hibernate state transition actions
- No support for state enter calls
- No support for multi_call

Data Types

options()

server_ref()

Function Index

Function Details

call/2

Equivalent to `call(ServerRef, Request, infinity)`.

Send a request to a gen_statem instance, and wait for a reply.

call/3

`ServerRef`: a reference to the gen_statem acquired via start`Request`: the request to send to the gen_statem`Timeout`: the amount of time in milliseconds to wait for a reply

returns: the reply sent back from the gen_statem; {error, Reason}, otherwise.

Send a request to a gen_statem instance, and wait for a reply..

This function will send the specified request to the specified gen_statem instance, and wait at least Timeout milliseconds for a reply from the gen_statem.

cast/2

`ServerRef`: a reference to the gen_statem acquired via start`Request`: the request to send to the gen_statem

returns: ok | {error, Reason}

Send a request to a gen_statem instance.

This function will send the specified request to the specified gen_statem instance, but will not wait for a reply.

reply/2

`Client`: the client to whom to send the reply`Reply`: the reply to send to the client

returns: an arbitrary term, that should be ignored

Send a reply to a calling client.

This function will send the specified reply back to the specified gen_statem client (e.g, via call/3). The return value of this function can be safely ignored.

start/3

`Module`: the module in which the gen_statem callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_statem

returns: the gen_statem pid, if successful; {error, Reason}, otherwise.

Start an un-named gen_statem.

This function will start a gen_statem instance.

Note. *The Options argument is currently ignored.*

start/4

`ServerName`: the name with which to register the gen_statem`Module`: the module in which the gen_statem callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_statem

returns: the gen_statem pid, if successful; {error, Reason}, otherwise.

Start a named gen_statem.

This function will start a gen_statem instance and register the newly created process with the process registry. Subsequent calls may use the gen_statem name, in lieu of the process id.

Note. *The Options argument is currently ignored.*

start_link/3

`Module`: the module in which the gen_statem callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_statem

returns: the gen_statem pid, if successful; {error, Reason}, otherwise.

Start an un-named gen_statem.

This function will start a gen_statem instance.

This version of the start function will link the started gen_statem process to the calling process.

Note. *The Options argument is currently ignored.*

start_link/4

`ServerName`: the name with which to register the gen_statem`Module`: the module in which the gen_statem callbacks are defined`Args`: the arguments to pass to the module's init callback`Options`: the options used to create the gen_statem

returns: the gen_statem pid, if successful; {error, Reason}, otherwise.

Start a named gen_statem.

This function will start a gen_statem instance and register the newly created process with the process registry. Subsequent calls may use the gen_statem name, in lieu of the process id.

This version of the start function will link the started gen_statem process to the calling process.

Note. *The Options argument is currently ignored.*

stop/1

Equivalent to `stop(ServerRef, normal, infinity)`.

Stop a previously started gen_statem.

stop/3

`ServerRef`: a reference to the gen_statem acquired via start`Reason`: the reason to supply for stopping`Timeout`: maximum time to wait for shutdown

returns: ok, if the gen_statem stopped; {error, Reason}, otherwise.

Stop a previously started gen_statem instance.

This function will stop a gen_statem instance, providing the supplied Reason to the . If the gen_s-tatem is a named gen_statem, then the gen_statem name may be used to stop the gen_statem.

Module gen_tcp

- Description
- Data Types
- Function Index
- Function Details

A partial implementation of the Erlang/OTP gen_tcp interface.

Description

This module provides an implementation of a subset of the functionality of the Erlang/OTP gen_tcp interface. It is designed to be API-compatible with gen_tcp, with exceptions noted below.

This interface may be used to send and receive TCP packets, as either binaries or strings. Active and passive modes are supported for receiving data.

Caveats:

- Limited support for socket tuning parameters
- No support for **controlling_process/2**

Note. *Port drivers for this interface are not supportedon all AtomVM platforms.*

Data Types

connect_option()

listen_option()

option()

packet()

reason()

Function Index

Function Details

accept/1

`Socket`: the listening socket.

returns: a connection-based (tcp) socket that can be used for reading and writing

Accept a connection on a listening socket.

accept/2

`Socket`: the listening socket.`Timeout`: amount of time in milliseconds to wait for a connection

returns: a connection-based (tcp) socket that can be used for reading and writing

Accept a connection on a listening socket.

close/1

`Socket`: the socket to close

returns: ok.

Close the socket.

connect/3

`Address`: the address to which to connect`Port`: the port to which to connect`Options`: options for controlling the behavior of the socket (see below)

returns: {ok, Socket} | {error, Reason}

Connect to a TCP endpoint on the specified address and port.

If successful, this function will return a Socket which can be used with the send/2 and recv/2 and recv/3 functions in this module.

The following options are supported:

- **active** Active mode (default: true)
- **buffer** Size of the receive buffer to use in active mode (default: 512)
- **binary** data is received as binaries (as opposed to lists)
- **list** data is received as lists (default)

If the socket is connected in active mode, then the calling process will receive messages of the form {tcp, Socket, Packet} when data is received on the socket. If active mode is set to false, then applications need to explicitly call one of the recv operations in order to receive data on the socket.

controlling_process/2

`Socket`: the socket to which to assign the pid`Pid`: Pid to which to send messages

returns: ok | {error, Reason}.

Assign a controlling process to the socket. The controlling process will receive messages from the socket.

This function will return `{error, not_owner}` if the calling process is not the current controlling process.

By default, the controlling process is the process associated with the creation of the Socket.

listen/2

`Port`: the port number on which to listen. Specify 0 to use an OS-assigned port number, which can then be retrieved via the inet:port/1 function.`Options`: A list of configuration parameters.

returns: a listening socket, which is appropriate for use in accept/1

Create a server-side listening socket.

This function is currently unimplemented

recv/2

Equivalent to `recv(Socket, Length, infinity)`.

Receive a packet over a TCP socket from a source address/port.

recv/3

`Socket`: the socket over which to receive a packet`Length`: the maximum length to read of the received packet`Timeout`: the amount of time to wait for a packet to arrive

returns: {ok, Packet} | {error, Reason}

Receive a packet over a TCP socket from a source address/port.

This function is used when the socket is not created in active mode. The received packet data returned from this call, and should be of length no greater than the specified length. This function will return {error, closed} if the server gracefully terminates the server side of the connection.

This call will block until data is received or a timeout occurs.

Note. *Currently, the Timeout parameter isignored.*

send/2

`Socket`: The Socket obtained via connect/3`Packet`: the data to send

returns: ok | {error, Reason}

Send data over the specified socket to a TCP endpoint.

If successful, this function will return the atom ok; otherwise, an error with a reason.

Module gen_udp

- Description
- Data Types
- Function Index
- Function Details

An implementation of the Erlang/OTP gen_udp interface.

Description

This module provides an implementation of a subset of the functionality of the Erlang/OTP gen_udp interface. It is designed to be API-compatible with gen_udp, with exceptions noted below.

This interface may be used to send and receive UDP packets, as either binaries or strings. Active and passive modes are supported for receiving data.

Caveats:

- Currently no support for IPv6
- Currently limited support for socket tuning parameters
- Currently no support for closing sockets

Note. *Port drivers for this interface are not supportedon all AtomVM platforms.*

Data Types

option()

packet()

reason()

Function Index

Function Details

close/1

`Socket`: the socket to close

returns: ok

Close the socket.

controlling_process/2

`Socket`: the socket to which to assign the pid`Pid`: Pid to which to send messages

returns: ok | {error, Reason}.

Assign a controlling process to the socket. The controlling process will receive messages from the socket.

This function will return `{error, not_owner}` if the calling process is not the current controlling process.

By default, the controlling process is the process associated with the creation of the Socket.

open/1

Equivalent to `open(PortNum, [])`.

Create a UDP socket. This function will instantiate a UDP socket that may be used to send or receive UDP messages.

open/2

`PortNum`: the port number to bind to. Specify 0 to use an OS-assigned port number, which can then be retrieved via the inet:port/1 function.`Options`: A list of configuration parameters.

returns: an opaque reference to the socket instance, used in subsequent commands.

throws `bad_arg`

Create a UDP socket. This function will instantiate a UDP socket that may be used to send or receive UDP messages. This function will raise an exception with the bad_arg atom if there is no socket driver supported for the target platform.

Note. *The Params argument is currently ignored.*

recv/2

Equivalent to `recv(Socket, Length, infinity)`.

Receive a packet over a UDP socket from a source address/port.

recv/3

`Socket`: the socket over which to receive a packet`Length`: the maximum length to read of the received packet`Timeout`: the amount of time to wait for a packet to arrive

returns: {ok, {Address, Port, Packet}} | {error, Reason}

Receive a packet over a UDP socket from a source address/port. The address and port of the received packet, as well as the received packet data, are returned from this call. This call will block until data is received or a timeout occurs.

Note. *Currently Length and Timeout parameters areignored.*

Note. *Currently the length of the received packetis limited to 128 bytes.*

send/4

`Socket`: the socket over which to send a packet`Address`: the target address to which to send the packet`PortNum`: the port on target address to which to send the packet`Packet`: the packet of data to send

returns: ok | {error, Reason}

Send a packet over a UDP socket to a target address/port.

Note. *Currently only ipv4 addresses are supported.*

Module inet

- Data Types
- Function Index
- Function Details

Data Types

address_family()

hostname()

ip4_address()

ip_address()

moniker()

port_number()

socket()

socket_impl()

Function Index

Function Details

close/1

`Socket`: the socket to close

returns: ok.

Close the socket.

getaddr/2

`Name`: the name to resolve`Family`: the family to resolve it to

returns: The address or an error tuple.

Get the IP address associated with a given name.

peername/1

`Socket`: the socket

returns: The address and port of the remote end of an established connection.

The address and port representing the "remote" end of a connection. This function should be called on a running socket instance.

port/1

`Socket`: the socket from which to obtain the port number

returns: the port number associated with the local socket

Retrieve the actual port number to which the socket is bound. This function is useful if the port assignment is done by the operating system.

sockname/1

`Socket`: the socket

returns: The address and port of the local end of an established connection.

The address and port representing the "local" end of a connection. This function should be called on a running socket instance.

Module init

- Description
- Function Index
- Function Details

An implementation of the Erlang/OTP init interface.

Description

This module implements a strict subset of the Erlang/OTP init interface.

Function Index

Function Details

get_argument/1

`Flag`: flag to get values for

returns: `error` if no value is associated with provided flag or values in order of the command line

Returns values associated with a given command-line user flag. Currently always returns `error` on AtomVM.

get_plain_arguments/0

returns: plain command-line arguments as a list of strings.

Gets plain command-line arguments. Currently always returns `[]` on AtomVM.

Module io

- Description
- Data Types
- Function Index
- Function Details

An implementation of the Erlang/OTP io interface.

Description

This module implements a strict subset of the Erlang/OTP io interface.

Data Types

device()

format()

getopt()

standard_error()

standard_io()

user()

Function Index

Function Details

columns/0

Equivalent to `columns(standard_io)`.

Returns the number of columns for standard I/O.

columns/1

`IODevice`: IO device to get columns of

Returns the number of columns for passed I/O device.

format/1

Equivalent to `fwrite(Format)`.

format/2

Equivalent to `fwrite(Format, Data)`.

format/3

Equivalent to `fwrite(IODevice, Format, Data)`.

fwrite/1

Equivalent to `fwrite(Format, [])`.

fwrite/2

Equivalent to `fwrite(standard_io, Format, Data)`.

fwrite/3

`IODevice`: device to write to`Format`: format string

returns: ok

Format string and data to IO device. See io_lib:format/2 for information about formatting capabilities.

get_line/1

`Prompt`: prompt for user input

returns: string

Read string from console with prompt.

getopts/0

Equivalent to `getopts(standard_io)`.

Get all options for standard I/O

getopts/1

`IODevice`: IO device to get options for

Get all options for a given IODevice

printable_range/0

Returns the user-requested range of printable Unicode characters. Currently always returns `unicode`

put_chars/1

`Chars`: character(s) to write to console

returns: ok

Equivalent to `put_chars(standard_io, Chars)`.

Writes the given character(s) to the console.

put_chars/2

`Device`: device to send characters to`Chars`: character(s) to write to device

returns: ok

Writes the given character(s) to the given device.

requests/1

`requests(Requests) -> any()`

scan_erl_exprs/4

`IODevice`: device to read from`Prompt`: prompt to print`StartLocation`: start location for reading`Options`: options for tokenizing

returns: read tokens or an error

Reads data from IODevice with a given prompt

Module io_lib

- Description
- Data Types
- Function Index
- Function Details

An implementation of the Erlang/OTP io_lib interface.

Description

This module implements a strict subset of the Erlang/OTP io_lib interface.

Data Types

chars()

Function Index

Function Details

format/2

`Format`: format string`Args`: format argument

returns: string

Format string and data to a string. Features most of OTP `io_lib:format/2`. Raises `badarg` error if the number of format specifiers does not match the length of the Args.

fwrite/2

`fwrite(Format, Args) -> any()`

Equivalent to `format(Format, Args)`.

Format string and data to a string. Features most of OTP `io_lib:format/2`. Raises `badarg` error if the number of format specifiers does not match the length of the Args.

latin1_char_list/1

`Term`: term to test

returns: true if Term is a list of latin1 characters, false otherwise.

Determine if passed term is a list of ISO-8859-1 characters (0-255).

printable_list/1

`List`: term to test

Determine if `List` is a flat list of printable characters

write/1

`Term`: term to represent

Equivalent to `forma("~w", [Term])`.

Returns a character list that represents Term

write_atom/1

write_string/1

`String`: string to print

Equivalent to `write_string(String, $")`.

Returns the list of characters needed to print String as a string.

Module kernel

- Description
- Function Index
- Function Details

Root supervisor for the kernel application.

Behaviours: `application`, `supervisor`.

Function Index

Function Details

start/2

Start kernel application

Module lists

- Description
- Function Index
- Function Details

An implementation of the Erlang/OTP lists interface.

Description

This module implements a strict subset of the Erlang/OTP lists interface.

Function Index

Function Details

all/2

`Fun`: the predicate to evaluate`List`: the list over which to evaluate elements

returns: true if Fun(E) evaluates to true, for all elements in List

Evaluates to true iff Fun(E) =:= true, for all E in List

any/2

`Fun`: the predicate to evaluate`List`: the list over which to evaluate elements

returns: true if Fun(E) evaluates to true, for at least one in List

Evaluates to true iff Fun(E) =:= true, for some E in List

append/1

`ListOfLists`: a list of lists to make the general list from

returns: a list made of the sublists of `ListOfLists`.

Returns a list in which all the sublists of `ListOfLists` have been appended.

append/2

`List1`: a list`List2`: a list

returns: a list made of the elements of `List1` and `List2`.

Returns a new list `List3`, which is made from the elements of `List1` followed by the elements of `List2`.

delete/2

`E`: the member to delete`L`: the list from which to delete the value

returns: the result of removing E from L, if it exists in L; otherwise, L.

Remove E from L

dropwhile/2

`Pred`: the predicate to check against elements in List1

returns: List List1 tail at the point predicate returned false for a element

Drops elements Elem from List1 while Pred(Elem) returns true and returns the remaining list. The Pred function must return a boolean.

duplicate/2

`Count`: the number of times to duplicate the element`Elem`: the element to duplicate

returns: a list made of Elem duplicate Count times

Duplicate an element

filter/2

`Pred`: the predicate to apply to elements in List`List`: list

returns: all values in L for which Pred is true.

Filter a list by a predicate, returning the list of elements for which the predicate is true.

filtermap/2

`Fun`: the filter/map fun`List1`: the list where given fun will be applied

returns: Returns the result of application of given fun over given list items

Calls `Fun(Elem)` on successive elements `Elem` of `List1` in order to update or remove elements from `List1`.

`Fun/1` must return either a Boolean or a tuple `{true, Value}`. The function returns the list of elements for which `Fun` returns a new value, where a value of `true` is synonymous with `{true, Elem}`.

Example: `1> lists:filtermap(fun(X) -> case X rem 2 of 0 -> {true, X div 2}; _ -> false end end, [1,2,3,4,5]). [1,2]`

flatmap/2

`F`: the function to apply to elements of L

returns: List of mapped elements, appended

Map elements of list L by applying F, and append the results. F should return a list.

flatten/1

`L`: the list to flatten

returns: flattened list

recursively flattens elements of L into a single list

foldl/3

`Fun`: the function to apply`Acc0`: the initial accumulator`List`: the list over which to fold

returns: the result of folding Fun over L

Fold over a list of terms, from left to right, applying Fun(E, Accum) to each successive element in List

foldr/3

Equivalent to `foldl(Fun, Acc0, reverse(List))`.

Fold over a list of terms, from right to left, applying Fun(E, Accum) to each successive element in List

foreach/2

`Fun`: the predicate to evaluate`List`: the list over which to evaluate elements

returns: ok

Applies given fun to each list element

join/2

`Sep`: the separator`List`: list

returns: the result of inserting Sep between every element of List.

Inserts Sep between every element of List.

keydelete/3

`K`: the key to match`I`: the position in the tuple to compare (1..tuple_size)`L`: the list from which to delete the element

returns: the result of deleting any element in L who's Ith element matches K

Delete the entry in L whose Ith element matches K.

keyfind/3

`K`: the key to match`I`: the position in the tuple to compare (1..tuple_size)`L`: the list from which to find the element

returns: the tuple in L who's Ith element matches K; the atom false, otherwise

Find the entry in L whose Ith element matches K.

keymember/3

`K`: the key to match`I`: the position in the tuple to compare (1..tuple_size)`L`: the list from which to find the element

returns: true if there is a tuple in L who's Ith element matches K; the atom false, otherwise

Returns true if a Ith element matches K.

keyreplace/4

`NewTuple`: tuple containing the new key to replace param `K`

returns: result of replacing the first element in L who's Ith element matches K with the contents of NewTuple.

Returns the result of replacing NewTuple for the first element in L with who's Ith element matches K.

keystore/4

`Key`: the key to match`N`: the position in the tuple to compare (1..tuple_size)`TupleList`: the list of tuples from which to find the element`NewTuple`: the tuple to add to the list

returns: An updated TupleList where the first occurrence of `Key` has been replaced with `NewTuple`.

Searches the list of tuples `TupleList` for a tuple whose `N`th element compares equal to `Key`, replaces it with `NewTuple` if found. If not found, append `NewTuple` to `TupleList`.

keytake/3

`Key`: the key to match`N`: the position in the tuple to compare (1..tuple_size)`TupleList1`: the list of tuples from which to find the element

returns: `{value, Tuple, TupleList2}` if such a tuple is found, otherwise `false`. `TupleList2` is a copy of `TupleList1` where the first occurrence of `Tuple` has been removed.

Searches the list of tuples `TupleList1` for a tuple whose `N`th element compares equal to `Key`.

last/1

`L`: the proper list from which to get the last item

returns: the last item of the list.

Get the last item of a list.

map/2

`Fun`: the function to apply`List`: the list over which to map

returns: the result of mapping over L

Map a list of terms, applying Fun(E)

mapfoldl/3

`Fun`: the function to apply

returns: the result of mapping and folding Fun over L

Combine `map/2` and `foldl/3` in one pass.

member/2

`E`: the member to search for`L`: the list from which to get the value

returns: true if E is a member of L; false, otherwise.

Determine whether a term is a member of a list.

nth/2

`N`: the index in the list to get`L`: the list from which to get the value

returns: the value in the list at position N.

Get the value in a list at position N.

Returns the value at the specified position in the list. The behavior of this function is undefined if N is outside of the {1..length(L)}.

nthtail/2

`N`: the index to start the sublist from`L`: the list from which to extract a tail

returns: a sublist of list starting from position N.

Get the sublist of list L starting after the element N.

The behavior of this function is undefined if N is outside of the {0..length(L)}.

reverse/1

`L`: the list to reverse

returns: the elements of L in reverse order

Equivalent to `lists:reverse(L, [])`.

Erlang/OTP implementation of this function actually handles few simple cases and calls `lists:reverse/2` for the more generic case. Consequently, calling `lists:reverse/1` without a list or with an improper list of two elements will fail with a function clause exception on Erlang /OTP and with a badarg exception with this implementation.

reverse/2

`L`: the list to reverse`T`: the tail to append to the reversed list

`L`: the list to reverse`T`: the tail to append to the reversed list

`L`: the list to reverse`T`: the tail to append to the reversed list

returns: the elements of L in reverse order followed by T

Reverse the elements of L, followed by T. If T is not a list or not a proper list, it is appended anyway and the result will be an improper list.

If L is not a proper list, the function fails with badarg.

Following Erlang/OTP tradition, `lists:reverse/1,2` is a nif. It computes the length and then allocates memory for the list at once (2 * n terms).

While this is much faster with AtomVM as allocations are expensive with default heap growth strategy, it can consume more memory until the list passed is garbage collected, as opposed to a recursive implementation where the process garbage collect part of the input list during the reversal.

Consequently, tail-recursive implementations calling `lists:reverse/2` can be as expensive or more expensive in memory than list comprehensions or non-tail recursive versions depending on the number of terms saved on the stack between calls.

For example, a non-tail recursive join/2 implementation requires two terms on stack for each iteration, so when it returns it will use `n * 3` (stack) + `n * 4` (result list) a tail recursive version will use, on last iteration: `n * 4` (reversed list) + n * 4' (result list)

search/2

`Pred`: the predicate to apply to elements in List`List`: search

returns: the first {value, Val}, if Pred(Val); false, otherwise.

If there is a Value in List such that Pred(Value) returns true, returns {value, Value} for the first such Value, otherwise returns false.

seq/2

`From`: from integer`To`: to Integer

returns: list of integers from [From..To]

Returns a sequence of integers in a specified range.

This function is equivalent to `lists:seq(From, To, 1).`

seq/3

`From`: from integer`To`: to Integer`Incr`: increment value

returns: list of integers `[From, From+Incr, ..., N]`, where `N` is the largest integer `<=` `To` incremented by `Incr`

Returns a sequence of integers in a specified range incremented by a specified value.

sort/1

`List`: a list

returns: Sorted list, ordered by `<`

Returns a sorted list, using `<` operator to determine sort order.

sort/2

`Fun`: sort function`List`: a list

returns: Sorted list, ordered by Fun(A, B) : boolean() such that A "less than" B.

Returns a sorted list, using Fun(A, B) to determine sort order.

split/2

`N`: elements non negative Integer`List1`: list to split

returns: Tuple with the two lists

Splits List1 into List2 and List3. List2 contains the first N elements and List3 the remaining elements (the Nth tail).

sublist/2

`List`: list to take the sublist from`Len`: the number of elements to get from List returns: a list made of the first `Len` elements of `List`

Return a sublist made of the first `Len` elements of `List`. It is not an error for `Len` to be larger than the length of `List`.

usort/1

`List`: a list

returns: Sorted list with duplicates removed, ordered by `<`

Returns a unique, sorted list, using `<` operator to determine sort order.

See also: sort/1.

usort/2

`Fun`: sort function`List`: a list

returns: Sorted list with duplicates removed, ordered by Fun.

Returns a unique, sorted list.

See also: sort/2.

Module logger

- Description
- Data Types
- Function Index
- Function Details

A naive implementation of the Erlang/OTP logger interface.

Description

This module implements a strict subset of the Erlang/OTP logger interface, supporting operations for logging messages to various log handlers. A default handler (`logger_std_h`) supports logging to the console.

This module is designed to be API-compatible with the Erlang/OTP logger API, with exceptions noted below. Users can use macros defined in the Erlang/OTP `logger.hrl` header for logging messages.

Limitations include but are not limited to:

- No support for logging filters
- No support for logging formatters
- No API support for logger configuration; all configuration must be done at initialization of the `logger_manager`
- No support for throttling or compacting sequences of repeated log messages

Data Types

level()

string_or_report()

Function Index

Function Details

alert/1

`StringOrReport`: string or report

returns: `ok`

Log a string at alert log level.

alert/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at alert log level.

alert/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata

returns: `ok`

Log a format string with args at alert log level with the specified metadata.

allow/2

`Level`: the log level`Module`: the module

returns: `true` if logging should be permitted at the specified level for the specified module; `false`, otherwise.

Determine whether logging should be permitted at the specified level for the specified module

compare/2

`Level1`: a level`Level2`: a level

returns: `lt | eq | gt`

Return comparison between levels

`lt` if `Level1 < Level2` `eq` if `Level1 == Level2` `gt` if `Level1 > Level2`

critical/1

`StringOrReport`: string or report

returns: `ok`

Log a string at critical log level.

critical/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at critical log level.

critical/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata

returns: `ok`

Log a format string with args at critical log level with the specified metadata.

debug/1

`StringOrReport`: string or report

returns: `ok`

Log a string at debug log level.

debug/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at debug log level.

debug/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata returns: `ok`

Log a format string with args at debug log level with the specified metadata.

emergency/1

`StringOrReport`: string or report

returns: `ok`

Log a string at emergency log level.

emergency/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at emergency log level.

emergency/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata

returns: `ok`

Log a format string with args at emergency log level with the specified metadata.

error/1

`StringOrReport`: string or report

returns: `ok`

Log a string at error log level.

error/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at error log level.

error/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata

returns: `ok`

Log a format string with args at error log level with the specified metadata.

info/1

`StringOrReport`: string or report

returns: `ok`

Log a string at info log level.

info/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at info log level.

info/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata

returns: `ok`

Log a format string with args at info log level with the specified metadata.

log/2

`Level`: log level`StringOrReport`: string or report map

returns: `ok`

Log a string at the specified log level.

log/3

`Level`: log level`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at the specified log level.

log/4

`Level`: log level`Format`: format string`Args`: format string arguments`Meta`: log metadata

returns: `ok`

Log a format string with args at the specified log level with the specified metadata.

notice/1

`StringOrReport`: string or report

returns: `ok`

Log a string at notice log level.

notice/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at notice log level.

notice/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata

returns: `ok`

Log a format string with args at notice log level with the specified metadata.

warning/1

`StringOrReport`: string or report

returns: `ok`

Log a string at warning log level.

warning/2

`FormatOrReport`: format string or report`ArgsOrMeta`: format string arguments or metadata

returns: `ok`

Log a format string with args at warning log level.

warning/3

`Format`: format string`Args`: format string arguments`MetaData`: log metadata

returns: `ok`

Log a format string with args at warning log level with the specified metadata.

Module maps

- Description
- Data Types

A *naive* implementation of the Erlang/OTP `maps` interface.

Description

The `maps` module provides several convenience operations for interfacing with the Erlang map type, which associates (unique) keys with values.

Note that the ordering of entries in a map is implementation-defined. While many operations in this module present entries in lexical order, users should in general make no assumptions about the ordering of entries in a map.

This module implements a subset of the Erlang/OTP `maps` interface. Some OTP functions are not implemented, and the approach favors correctness and readability over speed and performance.

Data Types

iterator()

iterator()

abstract datatype: `iterator(Key, Value)`

iterator_order()

iterator_order()

map_or_iterator()

Function Index

Function Details

filter/2

`Pred`: a function used to filter entries from the map`MapOrIterator`: the map or map iterator to filter

returns: a map containing all elements in `MapOrIterator` that satisfy `Pred`

Return a map who's entries are filtered by the supplied predicate.

This function returns a new map containing all elements from the input `MapOrIterator` that satisfy the input `Pred`.

The supplied predicate is a function from key-value inputs to a boolean value.

This function raises a `{badmap, Map}` error if `Map` is not a map or map iterator, and a `badarg` error if the input predicate is not a function.

find/2

`Key`: the key to find`Map`: the map in which to search

returns: `{ok, Value}` if `Key` is in `Map`; `error`, otherwise.

Returns `{ok, Value}` if `Key` is in `Map`; `error`, otherwise.

This function raises a `{badmap, Map}` error if `Map` is not a map.

fold/3

`Fun`: function over which to fold values`MapOrIterator`: the map or map iterator over which to fold

returns: the result of folding over all elements of the supplied map.

Fold over the entries in a map.

This function takes a function used to fold over all entries in a map and an initial accumulator value to use as the value supplied to the first entry in the map.

This function raises a `badmap` error if `Map` is not a map or map iterator, and a `badarg` error if the input function is not a function.

foreach/2

`Fun`: function to call with every key-value pair`MapOrIterator`: the map or map iterator over which to iterate

returns: `ok`

Iterate over the entries in a map.

This function takes a function used to iterate over all entries in a map.

This function raises a `badmap` error if `Map` is not a map or map iterator, and a `badarg` error if the input function is not a function.

from_keys/2

`List`: the list of keys of the map that will be created`Value`: the value that will be used as value for all map items

returns: a map having all provided keys having provided value as value

Creates a map with specified keys intialized to given value

from_list/1

`List`: a list of `[{Key, Value}]` pairs

returns: the map containing the entries from the list of supplied key-value pairs.

This function constructs a map from the supplied list of key-value pairs.

If the input list contains duplicate keys, the returned map will contain the right-most entry.

This function will raise a `badarg` error if the input is not a proper list or contains an element that is not a key-value pair.

get/2

`Key`: the key to get`Map`: the map from which to get the value

returns: the value in `Map` associated with `Key`, if it exists.

Get the value in `Map` associated with `Key`, if it exists.

This function raises a `{badkey, Key}` error if 'Key' does not occur in `Map` or a `{badmap, Map}` error if `Map` is not a map.

get/3

`Key`: the key`Map`: the map`Default`: default value

returns: the value in `Map` associated with `Key`, or `Default`, if the key is not associated with a value in `Map`.

Get the value in `Map` associated with `Key`, or `Default`, if the key is not associated with a value in `Map`.

This function raises a `{badmap, Map}` error if `Map` is not a map.

is_key/2

`Key`: the key`Map`: the map

returns: `true` if `Key` is associated with a value in `Map`; `false`, otherwise.

Return `true` if `Key` is associated with a value in `Map`; `false`, otherwise.

This function raises a `{badmap, Map}` error if `Map` is not a map.

iterator/1

Equivalent to `iterator(Map, undefined)`.

iterator/2

`Map`: the map`Order`: the iterator order, or undefined for default (unspecified) order.

returns: an iterator structure that can be used to iterate over associations in a map.

Return an iterator structure that can be used to iterate over associations in a map.

This function raises a `{badmap, Map}` error if `Map` is not a map.

See also: next/1.

keys/1

`Map`: the map

returns: the list of keys that occur in this map.

Returns the list of keys that occur in this map.

No guarantees are provided about the order of keys returned from this function.

This function raises a `{badmap, Map}` error if `Map` is not a map.

map/2

`Fun`: the function to apply to every entry in the map`Map`: the map to which to apply the map function

returns: the result of applying `Fun` to every entry in `Map`

Returns the result of applying a function to every element of a map.

This function raises a `badmap` error if `Map` is not a map or map iterator, and a `badarg` error if the input function is not a function.

merge/2

`Map1`: a map`Map2`: a map

returns: the result of merging entries from `Map1` and `Map2`.

Merge two maps to yield a new map.

If `Map1` and `Map2` contain the same key, then the value from `Map2` will be used.

This function raises a `badmap` error if neither `Map1` nor `Map2` is a map.

merge_with/3

`Combiner`: a function to merge values from Map1 and Map2 if a key exists in both maps`Map1`: a map`Map2`: a map

returns: the result of merging entries from `Map1` and `Map2`.

Merge two maps to yield a new map.

If `Map1` and `Map2` contain the same key, then the value from `Combiner(Key, Value1, Value2)` will be used.

This function raises a `badmap` error if neither `Map1` nor `Map2` is a map.

new/0

returns: a new map

Return a new (empty) map.

next/1

`Iterator`: a map iterator

returns: the key and value, along with the next iterator in the map, or the atom `none` if there are no more items over which to iterate.

Returns the next key and value in the map, along with a new iterator that can be used to iterate over the remainder of the map.

This function raises a `badarg` error if the supplied iterator is not of the expected type. Only use iterators that are returned from functions in this module.

put/3

`Key`: the key`Value`: the value`Map`: the map

returns: A copy of `Map` containing the `{Key, Value}` association.

Return the map containing the `{Key, Value}` association.

If `Key` occurs in `Map` then it will be over-written. Otherwise, the returned map will contain the new association.

This function raises a `{badmap, Map}` error if `Map` is not a map.

remove/2

`Key`: the key to remove`MapOrIterator`: the map or map iterator from which to remove the key

returns: a new map without `Key` as an entry.

Remove an entry from a map using a key.

If `Key` does not occur in `Map`, then the returned Map has the same entries as the input map or map iterator.

Note. This function extends the functionality of the OTP `remove/2` function, since the OTP interface only takes a map as input.

This function raises a `badmap` error if `Map` is not a map or map iterator.

size/1

`Map`: the map

returns: the size of the map

Returns the size of (i.e., the number of entries in) the map

This function raises a `{badmap, Map}` error if `Map` is not a map.

to_list/1

returns: a list of `[{Key, Value}]` tuples

Return the list of entries, expressed as `{Key, Value}` pairs, in the supplied map.

If provided with a map, no guarantees are provided about the order of entries returned from this function. Order can be controlled with `iterator/2`

This function raises a `{badmap, Map}` error if `Map` is not a map and not an iterator.

update/3

`Key`: the key to update`Value`: the value to update`Map`: the map to update

returns: a new map, with `Key` updated with `Value`

Returns a new map with an updated key-value association.

This function raises a `badmap` error if `Map` is not a map and `{badkey, Key}` if key doesn`t exist

values/1

`Map`: the map

returns: the list of values that occur in this map.

Returns the list of values that occur in this map.

No guarantees are provided about the order of values returned from this function.

This function raises a `{badmap, Map}` error if `Map` is not a map.

Module math

Function Index

Function Details

acos/1

acosh/1

asin/1

asinh/1

atan/1

atan2/2

atanh/1

ceil/1

cos/1

cosh/1

exp/1

floor/1

fmod/2

log/1

log10/1

log2/1

pi/0

pow/2

sin/1

sinh/1

sqrt/1

tan/1

tanh/1

Module net

- Data Types
- Function Index
- Function Details

Data Types
addrinfo()
service()

Function Index

Function Details

getaddrinfo/1

`Host`: the host string for which to find address information

returns: Address info for the specified host

Equivalent to `getaddrinfo(Host, undefined)`.

Retrieve address information for a given hostname.

getaddrinfo/2

`Host`: the host string for which to find address information`Service`: the service string for which to find address information

returns: Address info for the specified host and service

Retrieve address information for a given hostname and service.

The `Host` parameter may be a fully qualified host name or a string containing a valid dotted pair IP address. (Currently, only IPv4 is supported).

The `Service` parameter may be the name of a service (as defined via `services(3)` or a string containing a decimal value of the same.

Note that the `Host` or `String` parameter may be `undefined`, but not both.

gethostname/0

returns: The (usually short) name of the host.

Get the hostname

Module net_kernel

- Description
- Function Index
- Function Details

An implementation of the Erlang/OTP net_kernel interface.

Behaviours: `gen_server`.

Description

This module implements a strict subset of the Erlang/OTP net_kernel interface.

Function Index

Function Details

epmd_module/0

returns: the module to use to connect to epmd daemon

Get the epmd client module implementation.

get_net_ticktime/0

returns: the net tick time in seconds

Get the current net tick time

get_state/0

returns: a map describing state of distribution

Get state of erlang distribution

start/2

`Name`: name of the node

Start erlang distribution

stop/0

```
stop() -> any()
```

Stop erlang distribution

Module net_kernel_sup

- Description
- Function Index
- Function Details

Supervisor for erlang distribution.

Behaviours: `supervisor`.

Description

This module matches OTP's undocumented erl_distribution module and serves as the supervisor for everything erlang distribution related.

Function Index

Function Details

init/1

`init(Options) -> any()`

Module proplists

- Description
- Data Types
- Function Index
- Function Details

An implementation of the Erlang/OTP proplists interface.

Description

This module implements a strict subset of the Erlang/OTP proplists interface.

Data Types

property()

proplist()

Function Index

Function Details

compact/1

`ListIn`: the list will be compacted, such as `[{key, true}]`

returns: the compacted list, such as `[key]`

Minimizes the representation of all entries in the list. This is equivalent to `[property(P) || P <- ListIn]`. See also `property/1`, `unfold/1`.

delete/2

`Key`: the item key that will be deleted`List`: the property list from which items will be deleted

returns: A list without items having key `Key`

Deletes all entries associated with `Key` from `List`.

from_map/1

`Map`: the map that will be converted, such as `#{key => true}`

returns: the map converted to list, such as `[{key, true}]`

Converts the map `Map` to a property list.

get_all_values/2

`Key`: the key with which to find the values`List`: the property list from which to get the value

returns: a list of values for all entries having `Key` as key

Similar to `get_value/2`, but returns the list of values for *all* entries `{Key, Value}` in `List`. If no such entry exists, the result is the empty list.

get_bool/2

`Key`: the key that will be searched`List`: the list where key is searched

returns: `true` when exists an option with given key that is `true`, otherwise `false`

Returns the value of a boolean key/value option. If `lookup(Key, List)` would yield `{Key, true}`, this function returns `true`, otherwise `false`. See also `get_value/2`, `lookup/2`.

get_value/2

Equivalent to `get_value(Key, List, undefined)`.

Get a value from a property list.

get_value/3

`Key`: the key with which to find the value`List`: the property list from which to get the value`Default`: the default value to return, if Key is not in the property list.

returns: the value in the property list under the key, or Default, if Key is not in List.

Get a value from a property list.

Returns the value under the specified key, or the specified Default, if the Key is not in the supplied List. If the Key corresponds to an entry in the property list that is just a single atom, this function returns the atom true.

is_defined/2

`Key`: the key that will be searched`List`: the list where key is searched

returns: `true` if `Key` is defined, otherwise false

Returns `true` if `List` contains at least one entry associated with `Key`, otherwise `false`.

lookup/2

`Key`: the key with which to find the entry`List`: the property list from which to get the entry

returns: Either the found entry (always as a tuple) or `none`

Returns the first entry associated with `Key` in `List`, if one exists, otherwise returns `none`. For an atom `A` in the list, the tuple `{A, true}` is the entry associated with `A`. See also `get_bool/2`, `get_value/2`, `lookup_all/2`.

lookup_all/2

`Key`: the key with which to find the entries`List`: the property list from which to get the entries

returns: all entries having `Key` as key

Returns the list of all entries associated with `Key` in `List`. If no such entry exists, the result is the empty list. See also `lookup/2`.

property/1

`PropertyIn`: a property

returns: the same property in normal form

Creates a normal form (minimal) representation of a property. If `PropertyIn` is `{Key, true}`, where `Key` is an atom, `Key` is returned, otherwise the whole term `PropertyIn` is returned. See also `property/2`.

property/2

`Key`: the property key`Value`: the property value

returns: Creates a property in normal form

Creates a normal form (minimal) representation of a simple key/value property. Returns `Key` if `Value` is `true` and `Key` is an atom, otherwise a tuple `{Key, Value}` is returned. See also `property/1`.

to_map/1

`List`: the list will be converted to a map, such as `[key, {one, 1}]`

returns: the list converted as a map, such as `#{key => true, one => 1}`

Converts the property list `List` to a map

Shorthand atom values in `List` will be expanded to an association of the form `Atom => true`. Tuples of the form `{Key, Value}` in `List` will be converted to an association of the form `Key => Value`. Anything else will be silently ignored.

If the same key appears in `List` multiple times, the value of the one appearing nearest to the head of `List` will be in the result map, that is the value that would be returned by a call to `get_value(Key, List)`.

unfold/1

`ListIn`: the list that will be unfolded, such as `[key]`

returns: the unfolded list, such as `{key, true}`

Unfolds all occurrences of atoms in `ListIn` to tuples `{Atom, true}`.

Module queue

- Data Types
- Function Index
- Function Details

Data Types

queue()

queue()

abstract datatype: `queue(Item)`

Function Index

Function Details

all/2

`Pred`: the predicate function to apply to each item`Q`: the queue to check against the predicate

returns: Returns `true` if `Pred(Item)` returns `true` for all items `Item` in `Q`, otherwise `false`.

Returns `true` if `Pred(Item)` returns `true` for all items `Item` in `Q`, otherwise `false`.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). 2> queue:all(fun (E) -> E > 3 end, Queue). false 3> queue:all(fun (E) -> E > 0 end, Queue). true`

any/2

`Pred`: the predicate function to apply to each item`Q`: the queue to check against the predicate

returns: Returns `true` if `Pred(Item)` returns `true` for at least one item `Item` in `Q`, otherwise `false`.

Returns `true` if `Pred(Item)` returns `true` for at least one item `Item` in `Q`, otherwise `false`.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). 2> queue:any(fun (E) -> E > 10 end, Queue). false 3> queue:any(fun (E) -> E > 3 end, Queue). true` *delete/2*

`Item`: the item to delete from the queue`Q1`: the queue from which the item will be deleted

returns: Returns a copy of `Q1` where the first item matching `Item` is deleted, if there is such an item.

Returns a copy of `Q1` where the first item matching `Item` is deleted, if there is such an item.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). 2> Queue1 = queue:delete(3, Queue). 3> queue:member(3, Queue1). false`

delete_r/2

`Item`: the item to delete from the queue`Q1`: the queue from which the item will be deleted

returns: Returns a copy of `Q1` where the last item matching `Item` is deleted, if there is such an item.

Returns a copy of `Q1` where the last item matching `Item` is deleted, if there is such an item.

Example: `1> Queue = queue:from_list([1,2,3,4,3,5]). 2> Queue1 = queue:delete_r(3, Queue). 3> queue:to_list(Queue1). [1,2,3,4,5]`

delete_with/2

`Pred`: the predicate function to apply to each item`Q1`: the queue from which the item will be deleted

returns: Returns a copy of `Q1` where the first item for which `Pred` returns `true` is deleted, if there is such an item.

Returns a copy of `Q1` where the first item for which `Pred` returns `true` is deleted, if there is such an item.

Example: `1> Queue = queue:from_list([100,1,2,3,4,5]). 2> Queue1 = queue:delete_with(fun (E) -> E > 0, Queue). 3> queue:to_list(Queue1). [1,2,3,4,5]`

delete_with_r/2

`Pred`: the predicate function to apply to each item`Q1`: the queue from which the item will be deleted

returns: Returns a copy of `Q1` where the last item for which `Pred` returns `true` is deleted, if there is such an item.

Returns a copy of `Q1` where the last item for which `Pred` returns `true` is deleted, if there is such an item.

Example: `1> Queue = queue:from_list([1,2,3,4,5,100]). 2> Queue1 = queue:delete_with_r(fun (E) -> E > 10, Queue). 3> queue:to_list(Queue1). [1,2,3,4,5]`

drop/1

`Q1`: the queue from which the first element will be removed

returns: Returns a queue `Q2` that is the result of removing the front item from `Q1`. Fails with reason `empty` if `Q1` is empty

Returns a queue `Q2` that is the result of removing the front item from `Q1`. Fails with reason `empty` if `Q1` is empty.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue = queue:drop(Queue). {[5,4,3],[2]} 3> queue:to_list(Queue1). [2,3,4,5]`

drop_r/1

`Q1`: the queue from which the last element will be removed

returns: Returns a queue `Q2` that is the result of removing the rear item from `Q1`. Fails with reason `empty` if `Q1` is empty

Returns a queue `Q2` that is the result of removing the rear item from `Q1`. Fails with reason `empty` if `Q1` is empty.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue =`

`queue:drop_r(Queue). {[4,3],[1,2]} 3> queue:to_list(Queue1). [1,2,3,4]`

filter/2

`Fun`: the function to be applied to each item in the queue`Q1`: the queue where the function will be applied

returns: Returns a queue `Q2` that is the result of calling `Fun(Item)` on all items in `Q1`

Returns a queue `Q2` that is the result of calling `Fun(Item)` on all items in `Q1`. If `Fun(Item)` returns `true`, `Item` is copied to the result queue. If it returns `false`, `Item` is not copied. If it returns a list, the list elements are inserted instead of `Item` in the result queue.

Example 1: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue1 = queue:filter(fun (E) -> E > 2 end, Queue). {[5],[3,4]} 3> queue:to_list(Queue1). [3,4,5]`

Example 2: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue1 = queue:filter(fun (E) -> [E, E+1] end, Queue). {[6,5,5,4,4,3],[1,2,2,3]} 3> queue:to_list(Queue1). [1,2,2,3,3,4,4,5,5,6]`

filtermap/2

`Fun`: the function to be applied to each item in the queue`Q1`: the queue where the function will be applied

returns: Returns a queue `Q2` that is the result of calling `Fun(Item)` on all items in `Q1`

Returns a queue `Q2` that is the result of calling `Fun(Item)` on all items in `Q1`. If `Fun(Item)` returns `true`, `Item` is copied to the result queue. If it returns `false`, `Item` is not copied. If it returns `{true, NewItem}`, the queue element at this position is replaced with `NewItem` in the result queue.

Example 1: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue1 = queue:filtermap(fun (E) -> E > 2 end, Queue). {[5],[3,4]} 3> queue:to_list(Queue1). [3,4,5] 4> Queue1 = queue:filtermap(fun (E) -> {true, E+100} end, Queue). {[105,104,103],[101,102]} 5> queue:to_list(Queue1). [101,102,103,104,105]`

fold/3

`Fun`: the function to be applied to each item and accumulator`Acc0`: the initial accumulator value`Q`: the queue over which the function will be folded

returns: Returns the final value of the accumulator after folding over the queue

Calls `Fun(Item, AccIn)` on successive items `Item` of `Queue`, starting with `AccIn == Acc0`. The queue is traversed in queue order, that is, from front to rear. `Fun/2` must return a new accumulator, which is passed to the next call. The function returns the final value of the accumulator. `Acc0` is returned if the queue is empty.

Example: `1> queue:fold(fun(X, Sum) -> X + Sum end, 0, queue:from_list([1,2,3,4,5])). 15 2> queue:fold(fun(X, Prod) -> X * Prod end, 1, queue:from_list([1,2,3,4,5])). 120`

from_list/1

`L`: the list to be converted to a queue

returns: Returns a queue containing the items in `L` in the same order

Returns a queue containing the items in `L` in the same order; the head item of the list becomes the front item of the queue. This function is part of the Original API.

get/1

`Q`: the queue from which the first element will be returned

returns: Returns `Item` at the front of queue `Q`. Fails with reason `empty` if `Q` is empty

Returns `Item` at the front of queue `Q`. Fails with reason `empty` if `Q` is empty.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> 1 ==`

`queue:get(Queue). true`

get_r/1

`Q`: the queue from which the last element will be returned

returns: Returns `Item` at the rear of queue `Q`. Fails with reason `empty` if `Q` is empty

Returns `Item` at the rear of queue `Q`. Fails with reason `empty` if `Q` is empty.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> 5 == queue:get_r(Queue). true`

in/2

`Item`: the item that will be enqueued (inserted at the rear of the queue)`Q1`: the queue where the item will be inserted in

returns: Returns the queue with `Item` inserted at the rear of the queue

Inserts `Item` at the rear of queue `Q1`. Returns the resulting queue `Q2`. This function is part of the Original API

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue1 = queue:in(100, Queue). {[100,5,4,3],[1,2]} 3> queue:to_list(Queue1). [1,2,3,4,5,100]`

in_r/2

`Item`: the item that will be enqueued (inserted at the front of the queue)`Q1`: the queue where the item will be inserted in

returns: Returns the queue with `Item` inserted at the front of the queue

Inserts `Item` at the front of queue `Q1`. Returns the resulting queue `Q2`. This function is part of the Original API

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue1 = queue:in_r(100, Queue). {[5,4,3],[100,1,2]} 3> queue:to_list(Queue1). [100,1,2,3,4,5]`

is_empty/1

`Q`: the queue to be tested

returns: Returns `true` if `Q` is empty, otherwise `false`

Tests if `Q` is empty and returns `true` if so, otherwise `false`. This function is part of the Original API.

is_queue/1

`Term`: the term to be tested

returns: Returns `true` if `Term` is a queue, otherwise `false`

Tests if `Term` is a queue and returns `true` if so, otherwise `false`. Note that the test will return `true` for a term coinciding with the representation of a queue, even when not constructed by this module. This function is part of the Original API.

join/2

`Q1`: the first queue to be joined`Q2`: the second queue to be joined

returns: Returns a queue `Q3` that is the result of joining `Q1` and `Q2` with `Q1` in front of `Q2`

Returns a queue `Q3` that is the result of joining `Q1` and `Q2` with `Q1` in front of `Q2`. This function is part of the Original API

Example: `1> Queue1 = queue:from_list([1,3]). {[3],[1]} 2> Queue2 = queue:from_list([2,4]). {[4],[2]} 3> queue:to_list(queue:join(Queue1, Queue2)). [1,3,2,4]`

len/1

`Q`: the queue whose length is to be calculated

returns: Returns the length of queue `Q`

Calculates and returns the length of queue `Q`. This function is part of the Original API.

member/2

`Item`: the item to be searched in the queue`Q`: the queue to be searched

returns: Returns `true` if `Item` matches some element in `Q`, otherwise `false`

Returns `true` if `Item` matches some element in `Q`, otherwise `false`. This function is part of the Original API.

new/0

returns: Returns an empty queue.

This function returns an empty queue. This function is part of the Original API.

out/1

`Q1`: the queue from which the item will be dequeued (removed from the front)

returns: Returns a tuple `{{value, Item}, Q2}` where `Item` is the item removed and `Q2` is the resulting queue. If `Q1` is empty, tuple `{empty, Q1}` is returned

Removes the item at the front of queue `Q1`. Returns tuple `{{value, Item}, Q2}`, where `Item` is the item removed and `Q2` is the resulting queue. If `Q1` is empty, tuple `{empty, Q1}` is returned. This function is part of the Original API

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> {{value, 1=Item}, Queue1} = queue:out(Queue). {{value,1},{[5,4,3],[2]}} 3> queue:to_list(Queue1). [2,3,4,5]`

out_r/1

`Q1`: the queue from which the item will be dequeued (removed from the rear)

returns: Returns a tuple `{{value, Item}, Q2}` where `Item` is the item removed and `Q2` is the resulting queue. If `Q1` is empty, tuple `{empty, Q1}` is returned

Removes the item at the rear of queue `Q1`. Returns tuple `{{value, Item}, Q2}`, where `Item` is the item removed and `Q2` is the new queue. If `Q1` is empty, tuple `{empty, Q1}` is returned. This function is part of the Original API

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> {{value, 5=Item}, Queue1} = queue:out_r(Queue). {{value,5},{[4,3],[1,2]}} 3> queue:to_list(Queue1). [1,2,3,4]`

peek/1

`Q`: the queue from which the first element will be returned

returns: Returns tuple `{value, Item}`, where `Item` is the front item of `Q`, or `empty` if `Q` is empty

Returns tuple `{value, Item}`, where `Item` is the front item of `Q`, or `empty` if `Q` is empty.

Example: `1> queue:peek(queue:new()). empty 2> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 3> queue:peek(Queue). {value, 1}`

peek_r/1

`Q`: the queue from which the last element will be returned

returns: Returns tuple `{value, Item}`, where `Item` is the rear item of `Q`, or `empty` if `Q` is empty

Returns tuple `{value, Item}`, where `Item` is the rear item of `Q`, or `empty` if `Q` is empty.

Example: `1> queue:peek_r(queue:new()). empty 2> Queue =`

`queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 3> queue:peek_r(Queue). {value, 5}`

reverse/1

`Q1`: the queue to be reversed

returns: Returns a queue `Q2` containing the items of `Q1` in reverse order

Returns a queue `Q2` containing the items of `Q1` in reverse order. This function is part of the Original API

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> Queue1 = queue:reverse(Queue). {[2,1],[3,4,5]}`

split/2

`N`: the number of items to be put in the first resulting queue `Q2Q1`: the queue to be split

returns: Returns a tuple `{Q2, Q3}` where `Q2` contains the first `N` items of `Q1` and `Q3` contains the remaining items

Splits `Q1` in two. The `N` front items are put in `Q2` and the rest in `Q3`. This function is part of the Original API

to_list/1

`Q`: the queue to be converted to a list

returns: Returns a list of the items in the queue

Returns a list of the items in the queue in the same order; the front item of the queue becomes the head of the list.

Example: `1> Queue = queue:from_list([1,2,3,4,5]). {[5,4,3],[1,2]} 2> List == queue:to_list(Queue). true` This function is part of the Original API.

Module sets

- Data Types
- Function Index
- Function Details

Data Types

set()

abstract datatype: `set(Element)`

set()

Function Index

Function Details

add_element/2

`Element`: the element to add`Set1`: the set to add the element to

returns: Returns a new set formed from `Set1` with `Element` inserted.

Return `Set1` with `Element` inserted in it.

del_element/2

`Element`: the element to remove`Set1`: the set to remove the element from

returns: Returns `Set1`, but with `Element` removed.

Return `Set1` but with `Element` removed.

filter/2

`Pred`: the boolean function to filter elements with`Set1`: the set to filter

returns: Returns a set containing elements of `Set1` that satisfy the boolean function `Fun`. The evaluation order is undefined.

Filter Set with Fun.

filtermap/2

`Fun`: the filter and map fun`Set1`: the set to filter and map over

returns: Returns a set containing elements of `Set1` that are filtered and mapped using `Fun`.

Filters and maps elements in `Set1` with function `Fun`.

fold/3

`Set`: the set to fold over

returns: Returns the final value of the accumulator after folding the function over every element in the set.

Fold function Fun over all elements in Set and return Accumulator.

from_list/1

`List`: the list of items that is used for building the set

returns: Returns a set of the elements in `List`.

Builds a set from the elements in `List`.

from_list/2

`List`: the list to be converted to a set`Version`: only version 2 is supported

returns: Returns a set of the elements in `List` at the given version.

Builds a set from the elements in `List` using the specified version. Only `v2` format is supported.

intersection/1

`SetList`: the non-empty list of sets

returns: Returns the intersection of the non-empty list of sets.

Return the intersection of the list of sets.

intersection/2

`Set1`: the first set`Set2`: the second set

returns: Returns the intersection of `Set1` and `Set2`.

Return the intersection of `Set1` and `Set2`.

is_disjoint/2

`Set1`: the first set`Set2`: the second set

returns: Returns `true` if `Set1` and `Set2` are disjoint (have no elements in common), otherwise `false`.

Check whether `Set1` and `Set2` are disjoint.

is_element/2

`Element`: the element to check`Set`: the set to check against

returns: Returns `true` if `Element` is an element of `Set`, otherwise `false`.

Return `true` if `Element` is an element of `Set`, else `false`.

is_empty/1

`Set`: the set to be checked for emptiness returns: Returns `true` if `Set` is an empty set, otherwise `false`.

Returns `true` if `Set` is an empty set, otherwise `false`.

is_equal/2

`Set1`: first set to be checked for equality`Set2`: second set to be checked for equality

returns: Return `true` if `Set1` and `Set2` contain the same elements, otherwise `false`.

Returns `true` if `Set1` and `Set2` are equal, that is when every element of one set is also a member of the respective other set, otherwise `false`.

is_set/1

`Set`: the term that will be checked

returns: Return `true` if `Set` is a set of elements, else `false`.

Returns `true` if `Set` appears to be a set of elements, otherwise `false`.

Note that the test is shallow and will return `true` for any term that coincides with the possible representations of a set.

is_subset/2

`Set1`: the first set`Set2`: the second set

returns: Returns `true` when every element of `Set1` is also a member of `Set2`, otherwise `false`.

Return 'true' when every element of Set1 is also a member of Set2, else 'false'.

map/2

`Fun`: the mapping function`Set1`: the set to map over

returns: Returns a set containing elements of `Set1` that are mapped using `Fun`.

Map Set with Fun.

new/0

returns: Returns a new empty set.

Returns a new empty set using the v2 format (a map).

new/1

`Version`: must be `{version, 2}`

returns: Returns a new empty set.

Returns a new empty set (only v2 format is supported).

size/1

`Set`: the set for which size will be returned

returns: Return the number of elements in `Set`.

Returns the number of elements in `Set`.

subtract/2

`Set1`: the first set`Set2`: the second set

returns: Returns only the elements of `Set1` that are not also elements of `Set2`.

Return all and only the elements of Set1 which are not also in Set2.

to_list/1

`Set`: the set to be converted to a list

returns: Return the elements in Set as a list.

Returns the elements of `Set` as a list. The order of the returned elements is undefined.

union/1

`SetList`: the list of sets

returns: Returns the merged (union) set of the list of sets.

Return the union of the list of sets.

union/2

`Set1`: the first set`Set2`: the second set

returns: Returns the merged (union) set of `Set1` and `Set2`.

Return the union of `Set1` and `Set2`.

Module socket

- Data Types
- Function Index
- Function Details

Data Types

domain()

in_addr()

ip_mreq()

port_number()

protocol()

sockaddr()

sockaddr_in()

socket()

abstract datatype: `socket()`

socket_option()

type()

Function Index

Function Details

accept/1

Equivalent to `socket:accept(ListeningSocket, infinity)`.

accept/2

`Socket`: the socket`Timeout`: timeout (in milliseconds)

returns: `{ok, Connection}` if successful; `{error, Reason}`, otherwise.

Wait for the socket to accept a connection.

Wait for the socket to accept a connection. The socket should be set to listen for connections.

Note that this function will block until a connection is made from a client, unless `nowait` or a reference is passed as `Timeout`. Typically, users will spawn a call to `accept` in a separate process.

Example:

```
{ok, ConnectedSocket} = socket:accept(ListeningSocket)
```

bind/2

`Socket`: the socket`Address`: the address to which to bind the socket

returns: `ok` if successful; `{error, Reason}`, otherwise.

Bind a socket to an interface.

Bind a socket to an interface, via a socket address. Use `any` to bind to all interfaces. Use `loopback` to bind to the loopback address. To specify a port, use a map containing the network family, address, and port.

Example:

```
ok = socket:bind(ListeningSocket, #{family => inet, addr => any, port =>
44404})
```

close/1

`Socket`: the socket

returns: `ok` if successful; `{error, Reason}`, otherwise.

Close a socket.

Close a previously opened socket.

Example:

```
ok = socket:close(Socket)
```

connect/2

`Socket`: the socket`Address`: the address to which to connect the socket

returns: `ok` if successful; `{error, Reason}`, otherwise.

Wait for the socket to connect to an address.

Wait for the socket to connect to an address. The socket should be a connection-based socket.

Note that this function will block until a connection is made to a server.

Example:

```
ok = socket:connect(Socket, #{family => inet, addr => loopback, port =>
44404})
```

getopt/2

`Socket`: the socket`SocketOption`: the option

returns: `{ok, Value}` if successful; `{error, Reason}`, otherwise.

Get a socket option.

Currently, the following options are supported:

Example:

```
{ok, stream} = socket:getopt(ListeningSocket, {socket, type})
```

listen/1

`Socket`: the socket

returns: `ok` if successful; `{error, Reason}`, otherwise.

Set the socket to listen for connections.

Listen for connections. The socket should be a connection-based socket and should be bound to an address and port.

Example:

```
ok = socket:listen(ListeningSocket)
```

listen/2

`Socket`: the socket`Backlog`: the maximum length for the queue of pending connections

returns: `ok` if successful; `{error, Reason}`, otherwise.

Set the socket to listen for connections.

Listen for connections. The socket should be a connection-based socket and should be bound to an address and port.

Use the Backlog to specify the maximum length for the queue of pending connections

Example:

`ok = socket:listen(ListeningSocket, 4)`

open/3

`Domain`: the network domain`Type`: the network type`Protocol`: the network protocol

returns: `{ok, Socket}` if successful; `{error, Reason}`, otherwise.

Create a socket.

Create a socket with a specified domain, type, and protocol. Use the returned socket for communications.

Example:

`{ok, ListeningSocket} = socket:open(inet, stream, tcp)`

peername/1

`Socket`: the socket

returns: `{ok, Address}` if successful; `{error, Reason}`, otherwise.

Return the address of the peer connected to the specified socket.

Example:

`{ok, Address} = socket:peername(ConnectedSocket)`

recv/1

Equivalent to `socket:recv(Socket, 0)`.

recv/2

Equivalent to `socket:recv(Socket, Length, infinity)`.

recv/3

`Socket`: the socket`Length`: number of bytes to receive`Timeout`: timeout (in milliseconds)

returns: `{ok, Data}` if successful; `{error, Reason}`, otherwise.

Receive data on the specified socket.

This function is equivalent to `recvfrom/3` except for the return type.

Example:

`{ok, Data} = socket:recv(ConnectedSocket)`

recvfrom/1

Equivalent to `socket:recvfrom(Socket, 0)`.

recvfrom/2

Equivalent to `socket:recvfrom(Socket, Length, infinity)`.

recvfrom/3

`Socket`: the socket`Length`: number of bytes to receive`Timeout`: timeout (in milliseconds)

returns: `{ok, {Address, Data}}` if successful; `{error, Reason}`, otherwise.

Receive data on the specified socket, returning the from address.

Note that this function will block until data is received on the socket.

Example:

`{ok, {Address, Data}} = socket:recvfrom(ConnectedSocket)`

If socket is UDP, the function retrieves the first available packet and truncate it to Length bytes, unless Length is 0 in which case it returns the whole packet ("all available").

If socket is TCP and Length is 0, this function retrieves all available data without waiting (using peek if the platform allows it). If socket is TCP and Length is not 0, this function waits until Length bytes are available and return these bytes.

send/2

`Socket`: the socket`Data`: the data to send

returns: `{ok, Rest}` if successful; `{error, Reason}`, otherwise.

Send data on the specified socket.

Note that this function will block until data is sent on the socket. The data may not have been received by the intended recipient, and the data may not even have been sent over the network.

Example:

`ok = socket:send(ConnectedSocket, Data)`

sendto/3

`Socket`: the socket`Data`: the data to send`Dest`: the destination to which to send the data

returns: `{ok, Rest}` if successful; `{error, Reason}`, otherwise.

Send data on the specified socket to the specified destination.

Note that this function will block until data is sent on the socket. The data may not have been received by the intended recipient, and the data may not even have been sent over the network.

Example:

`ok = socket:sendto(ConnectedSocket, Data, Dest)`

setopt/3

`Socket`: the socket`SocketOption`: the option`Value`: the option value

returns: `ok` if successful; `{error, Reason}`, otherwise.

Set a socket option.

Set an option on a socket.

Currently, the following options are supported:

Example:

`ok = socket:setopt(ListeningSocket, {socket, reuseaddr}, true) ok = socket:setopt(ListeningSocket, {socket, linger}, #{onoff => true, linger => 0})`

shutdown/2

`Socket`: the socket`How`: how to shut the socket down

returns: `ok` if successful; `{error, Reason}`, otherwise.

Shut down one or both ends of a full-duplex socket connection.

Example:

`ok = socket:shutdown(Socket, read_write)`

sockname/1

`Socket`: the socket

returns: `{ok, Address}` if successful; `{error, Reason}`, otherwise.

Return the current address for the specified socket.

Example:

```
{ok, Address} = socket:sockname(ConnectedSocket)
```

Module socket_dist

- Function Index
- Function Details

Function Index

Function Details

accept/1

accept_connection/5

`accept_connection(AcceptPid, DistCtrl, MyNode, Allowed, SetupTime) -> any()`

address/0

close/1

`close(Listen) -> any()`

listen/1

select/1

`select(Node) -> any()`

setup/5

`setup(Node, Type, MyNode, LongOrShortNames, SetupTime) -> any()`

Module socket_dist_controller

- Function Index
- Function Details

Behaviours: `gen_server`.

Function Index

Function Details

address/2

`address(Controller, Node) -> any()`

code_change/3

`code_change(OldVsn, State, Extra) -> any()`

getll/1

`getll(Controller) -> any()`

getstat/1

`getstat(Controller) -> any()`

handle_call/3

`handle_call(X1, From, State) -> any()`

handle_cast/2

`handle_cast(X1, State) -> any()`

handle_info/2

`handle_info(X1, State0) -> any()`

handshake_complete/3

`handshake_complete(Controller, Node, DHandle) -> any()`

init/1

`init(Socket) -> any()`

recv/3

`recv(Controller, Length, Timeout) -> any()`

send/2

`send(Controller, Data) -> any()`

setopts_post_nodeup/1

`setopts_post_nodeup(Controller) -> any()`

setopts_pre_nodeup/1

`setopts_pre_nodeup(Controller) -> any()`

start/1

`start(Socket) -> any()`

supervisor/2

`supervisor(Controller, Pid) -> any()`

terminate/2

`terminate(Reason, State) -> any()`

tick/1

`tick(Controller) -> any()`

Module ssl

- Data Types
- Function Index
- Function Details

Behaviours: `gen_server`.

Data Types

client_option()

host()

hostname()

ip_address()

reason()

sni()

socket_option()

sslsocket()

abstract datatype: `sslsocket()`

tls_client_option()

Function Index

Function Details

close/1

connect/3

handle_call/3

`handle_call(X1, From, State) -> any()`

handle_cast/2

`handle_cast(Msg, State) -> any()`

handle_info/2

`handle_info(Msg, State) -> any()`

init/1

`init(X1) -> any()`

recv/2

send/2

start/0

stop/0

terminate/2

`terminate(Reason, State) -> any()`

Module string

- Description
- Function Index
- Function Details

An implementation of the Erlang/OTP string interface.

Description

This module implements a strict subset of the Erlang/OTP string interface.

Function Index

Function Details

find/2

`String`: string to search in`SearchPattern`: pattern to search

returns: remainder of String starting from first occurrence of SearchPattern or `nomatch` if SearchPattern cannot be found in String

Equivalent to `find(String, SearchPattern, leading)`.

find/3

`String`: string to search in`SearchPattern`: pattern to search`Direction`: direction to search, `leading` or `trailing`

returns: remainder of String starting from first or last occurrence of SearchPattern or `nomatch` if SearchPattern cannot be found in String

length/1

`String`: string to compute the length of

Return the length of the string in characters.

split/2

`String`: a string to split`Pattern`: the search pattern to split at

returns: chardata

Equivalent to `split(String, Pattern, leading)`.

split/3

`String`: a string to split`Pattern`: the search pattern to split at`Where`: position to split (leading, trailing, or all)

returns: chardata

Splits String where SearchPattern is encountered and return the remaining parts.

Where, default leading, indicates whether the leading, the trailing or all encounters of SearchPattern will split String.

Example:

```
   0> string:split("ab..bc..cd", "..").
["ab","bc..cd"]
1> string:split(<<"ab..bc..cd">>, "..", trailing).
[<<"ab..bc">>,<<"cd">>]
2> string:split(<<"ab..bc....cd">>, "..", all).
[<<"ab">>,<<"bc">>,<<>>,<<"cd">>]
```

to_lower/1

`Input`: a string or character to convert

returns: a Character or string

Convert string or character to uppercase.

The specified string or character is case-converted. Notice that the supported character set is ISO/IEC 8859-1 (also called Latin 1); all values outside this set are unchanged

to_upper/1

`Input`: a string or character to convert

returns: a Character or string

Convert string or character to uppercase.

The specified string or character is case-converted. Notice that the supported character set is ISO/IEC 8859-1 (also called Latin 1); all values outside this set are unchanged

trim/1

`String`: a string or character to trim whitespace

returns: a Character or string

Equivalent to `trim(String, both)`.

trim/2

`String`: a string or character to trim`Direction`: an atom indicating the direction from which to remove whitespace

returns: a Character or string

Returns a string, where leading or trailing, or both, whitespace has been removed.

If omitted, Direction is both.

Example:

```
   1> string:trim("\t  Hello  \n").
"Hello"
2> string:trim(<<"\t  Hello  \n">>, leading).
<<"Hello  \n">>
3> string:trim(<<".Hello.\n">>, trailing, "\n.").
<<".Hello">>
```

Module supervisor

Data Types

child_spec()

child_type()

restart()

shutdown()

strategy()

sup_flags()

Function Index

Function Details

delete_child/2

```
delete_child(Supervisor, ChildId) -> any()
```

handle_call/3

```
handle_call(X1, From, State) -> any()
```

handle_cast/2

```
handle_cast(Msg, State) -> any()
```

handle_info/2

```
handle_info(Msg, State) -> any()
```

init/1

```
init(X1) -> any()
```

restart_child/2

```
restart_child(Supervisor, ChildId) -> any()
```

start_child/2

```
start_child(Supervisor, ChildSpec) -> any()
```

start_link/2

```
start_link(Module, Args) -> any()
```

start_link/3

```
start_link(SupName, Module, Args) -> any()
```

terminate_child/2

```
terminate_child(Supervisor, ChildId) -> any()
```

Module timer

An implementation of the Erlang/OTP timer interface.

Description

This module implements a strict subset of the Erlang/OTP timer interface.

Function Index

Function Details

sleep/1

`Timeout`: number of milliseconds to sleep or `infinity`

returns: `ok`

Pauses the execution of the current process for a given number of milliseconds, or forever, using `infinity` as the parameter.

Module unicode

- Description
- Data Types
- Function Index
- Function Details

An implementation of the Erlang/OTP unicode interface.

Description

This module implements a strict subset of the Erlang/OTP unicode interface.

Data Types

chardata()

charlist()

encoding()

latin1_chardata()

unicode_binary()

Function Index

Function Details

characters_to_binary/1

`Data`: data to convert to UTF8

returns: an utf8 binary or a tuple if conversion failed.

Equivalent to `characters_to_binary(Data, utf8, utf8)`.

Convert character data to an UTF8 binary

characters_to_binary/2

`Data`: data to convert to UTF8`InEncoding`: encoding of data

returns: an utf8 binary or a tuple if conversion failed.

Equivalent to `characters_to_binary(Data, InEncoding, utf8)`.

Convert character data in a given encoding to an UTF8 binary

characters_to_binary/3

`Data`: data to convert to UTF8`InEncoding`: encoding of input data`OutEncoding`: output encoding

returns: an encoded binary or a tuple if conversion failed.

Convert character data in a given encoding to a binary in a given encoding.

If conversion fails, the function returns a tuple with three elements:

- First element is `error` or `incomplete`. `incomplete` means the conversion failed because of an incomplete unicode transform at the very end of data.

- Second element is what has been converted so far.
- Third element is the remaining data to be converted, for debugging purposes. This remaining data can differ with what Erlang/OTP returns.

Also, Erlang/OTP's implementation may error with `badarg` for parameters for which this function merely returns an error tuple.

characters_to_list/1

`Data`: data to convert to Unicode

returns: a list of characters or a tuple if conversion failed.

Convert UTF-8 data to a list of Unicode characters.

If conversion fails, the function returns a tuple with three elements:

- First element is `error` or `incomplete`. `incomplete` means the conversion failed because of an incomplete unicode transform at the very end of data.
- Second element is what has been converted so far.
- Third element is the remaining data to be converted, for debugging purposes. This remaining data can differ with what Erlang/OTP returns.

characters_to_list/2

`Data`: data to convert`Encoding`: encoding of data to convert

returns: a list of characters or a tuple if conversion failed.

Convert UTF-8 or Latin1 data to a list of Unicode characters. Following Erlang/OTP, if input encoding is latin1, this function returns an error tuple if a character > 255 is passed (in a list). Otherwise, it will accept any character within Unicode range (0-0x10FFFF).

See also: characters_to_list/1.

12.1.2 eavmlib

The eavmlib library

Modules

Module ahttp_client

- Data Types
- Function Index
- Function Details

Data Types

backend()

connection()

abstract datatype: `connection()`

data_response()

done_response()

error_tuple()

header_continuation_response()

header_response()

host()

option()

close/1

`Conn`: the connection

returns: Either `ok` or an error tuple.

Closes the connection.

connect/4

`Protocol`: the protocol, either http or https`Host`: the server hostname`Port`: the server port number, usually 80 (http) or 443 (https)`Options`: the property list with http client and connection options

returns: Either a http connection result tuple or an error tuple

Connects to the http(s) server.

The `Host` parameter may be a fully qualified host name or a string containing a valid dotted pair IP address. (Currently, only IPv4 is supported). `Host` can be also a binary.

The `Options` can be used for providing connection options such as SSL `{verify, verify_none}`, `gen_tcp` options such as `{active, false}` and `ahttp_client {parse_headers, [<<"HeaderName">>]}`.

recv/2

`Conn`: the connection`Len`: the number of bytes will be received, when using 0 all pending bytes are received

returns: Either an ok tuple with the updated connection and a list of responses or an error tuple.

Receive and parse a number of bytes from the http connection.

This function should be used when the connection has been opened using `{active, false}`.

See also `stream/2` for more information about the responses list.

request/5

`Conn`: the connection`Method`: a http method such as "GET", "POST", "PUT", etc…`Path`: the path to the http resource, such as "/"`Headers`: a list of headers`Body`: the body that is sent to the server, may be `undefined` or `nil` when there is no body

returns: Either a result tuple with the updated http connection and a reference to the http request, or an error tuple.

Makes a http request using given method on provided path.

When using methods such as `GET` the body should be omited using either `undefined` or `nil` (they are both equivalent).

When uploading a smaller body (a single binary that fits in memory) the body binary can be provided.

`stream` option can be used with `stream_request_body/3` in order to upload a bigger binary in streaming mode. This option should be combined with `Content-Length` header.

As soon as the request is sent to the server, a tuple such as `{ok, Conn, Ref}` is returned, otherwise an error tuple is returned, such as `{error, {gen_tcp, econnrefused}}`.

The returned connection should be used for the next call, such as to `stream/2` (when using active mode) or `recv/3` (when using passive mode). `Ref` is meant to identify a single request, so any response to a specific request will be identified from the same reference.

stream/2

`Conn`: the connection`Msg`: a received message

returns: Either a list of responses, `unknown` or an error tuple.

This function should be used when in active mode in order to process socket messages.

If a socket message is streamed using this function, a tuple with a list of http responses is returned (e.g. `{ok, UpdatedConn, Responses}``, or an error tuple.

Otherwise `unknown` is returned, that means that the message is not a socket message tied to the open connection, and it should be handled in some other way.

The first returned response to a new request is a status `{status, Ref, 200}` for a successful response. After that headers and data may follow.

Since the response might span multiple socket messages, `stream/2` may be called multiple times. Each time the latest `UpdatedConn` must be used.

stream_request_body/3

`Conn`: the connection`Ref`: the reference to the pending request`BodyChunk`: a chunk of the body that will be sent

returns: Either `ok` or an error tuple.

Uploads a chunk of request body.

This function should be used when `stream` has been used as Body parameter.

Module atomvm

- Description
- Data Types
- Function Index
- Function Details

AtomVM-specific APIs.

Description

This module contains functions that are specific to the AtomVM platform.

Data Types

avm_path()

platform_name()

posix_dir()

abstract datatype: `posix_dir()`

posix_error()

posix_fd()

abstract datatype: `posix_fd()`

posix_open_flag()

Function Index

Function Details

add_avm_pack_binary/2

`AVMData`: AVM data.`Options`: Options, as a property list.

returns: `ok`

Add code from an AVM binary to your application.

This function will add the data in the `AVMData` parameter to your application. The data is assumed to be valid AVM data (e.g, as generated by packbeam tooling).

Failure to properly load AVM data is result in a runtime `error`

add_avm_pack_file/2

`AVMPath`: Path to AVM data.`Options`: Options, as a property list.

returns: `ok`

Add code from an AVM binary to your application.

This function will add the data located in the `AVMPath` parameter to your application. The data is assumed to be valid AVM data (e.g, as generated by packbeam tooling).

On `generic_unix` platforms, the `AVMPath` may be a valid file system path to an AVM file.

On `esp32` platforms, the `AVMPath` should be the name of an ESP32 flash partition, prefixed with the string `/dev/partition/by-name/`. Thus, for example, if you specify `/dev/partition/by-name/main2.app` as the `AVMPath`, the ESP32 flash should contain a data partition with the name `main2.app`

Failure to properly load AVM path is result in a runtime `error`

close_avm_pack/2

`Name`: the AVM name.`Options`: Options, as a property list.

returns: `ok | error`

Close previously opened AVM binary from your application.

This function will close the data referenced by the `Name` parameter from your application. The `Name` parameter must reference previously opened AVM data.

Failure to close AVM data is result in a runtime `error`

get_start_beam/1

`AVM`: Name of avm (atom)

returns: the name of the start module (with suffix)

Get the start beam for a given avm

platform/0

returns: The platform name.

Return the platform moniker. You may use this function to uniquely identify the platform type on which your application is running.

posix_clock_settime/2

`ClockId`: The clock id`ValueSinceUnixEpoch`: The value, in specified seconds and nanoseconds, since the UNIX epoch (Jan 1, 1970)

returns: `ok` or an error tuple

Set the system time.

This function sets the system time to the specified value, expressed as a tuple containing seconds and nanoseconds since the UNIX epoch (Jan 1, 1970). Coordinates are all in UTC.

Note. Some systems may require special permissions to call this function.

posix_close/1

`File`: Descriptor to a file to close

returns: `ok` or an error tuple

Close a file that was opened with `posix_open/2,3` *posix_closedir/1*

`Dir`: Descriptor to a directory to close

returns: `ok` or an error tuple

Close a directory that was opened with `posix_opendir/1`

posix_open/2

`Path`: Path to the file to open`Flags`: List of flags passed to `open(3)`.

returns: A tuple with a file descriptor or an error tuple.

Open a file (on platforms that have `open(3)`). The file is automatically closed when the file descriptor is garbage collected.

Files are automatically opened with `O_NONBLOCK`. Other flags can be passed.

posix_open/3

`Path`: Path to the file to open`Flags`: List of flags passed to `open(3)`.`Mode`: Mode passed to `open(3)` for created file.

returns: A tuple with a file descriptor or an error tuple.

Open a file (on platforms that have `open(3)`). This variant can be used to specify the mode for new file.

posix_opendir/1

`Path`: Path to the directory to open

returns: A tuple with a directory descriptor or an error tuple.

Open a file (on platforms that have `opendir(3)`).

posix_read/2

`File`: Descriptor to an open file`Count`: Maximum number of bytes to read

returns: a tuple with read bytes, `eof` or an error tuple

Read at most `Count` bytes from a file. Files are open non-blocking. `atomvm:posix_select_read/3' can non-blocking. `atomvm:posix_select_read/3' can be used to determine if the file can be read. `eof` is returned if no more data can be read because the file cursor reached the end.

posix_readdir/1

`Dir`: Descriptor to an open directory

returns: a `{dirent, InodeNo, Name}` tuple, `eof` or an error tuple

Read a directory entry `eof` is returned if no more data can be read because the directory cursor reached the end.

posix_write/2

`File`: Descriptor to an open file`Data`: Data to write

returns: a tuple with the number of written bytes or an error tuple

Write data to a file. Files are open non-blocking. `atomvm:posix_select_write/3' can be used to determine if the file can be written.

rand_bytes/1

`Len`: non-negative integer

returns: Binary containing random sequence of bytes of length Len.

This function is deprecated: Use crypto:strong_rand_bytes/1 instead.

Returns a binary containing random sequence of bytes of length Len. Supplying a negative value will result in a badarg error. This function will use a cryptographically strong RNG if available. Otherwise, the random value is generated using a PRNG.

random/0

returns: random 32-bit integer.

Returns a random 32-bit integer value. This function will use a cryptographically strong RNG if available. Otherwise, the random value is generated using a PRNG.

read_priv/2

`App`: application name.`Path`: path to the resource.

returns: Binary containing the resource content.

This function allows to fetch priv/ resources content.

subprocess/4

`Path`: path to the command to execute`Args`: arguments to pass to the command. First item is the name of the command`Options`: options to run execve. Should be `[stdout]`

returns: a tuple with the process id and a fd to the stdout of the process.

Fork and execute a program using fork(2) and execve(2). Pipe stdout so output of the program can be read with `atomvm:posix_read/2`.

Module avm_pubsub

- Function Index
- Function Details

Function Index

Function Details

handle_call/3

`handle_call(X1, From, Table) -> any()`

handle_info/2

`handle_info(Info, Table) -> any()`

init/1

`init(X1) -> any()`

pub/3

`pub(PubSub, Topic, Term) -> any()`

start/0

`start() -> any()`

start/1

`start(LocalName) -> any()`

sub/2

`sub(PubSub, Topic) -> any()`

sub/3

`sub(PubSub, Topic, Pid) -> any()`

terminate/2

`terminate(Reason, State) -> any()`

unsub/2

`unsub(PubSub, Topic) -> any()`

unsub/3

`unsub(PubSub, Topic, Pid) -> any()`

Module console

- Description
- Function Index
- Function Details

This modules supports output of string data to the console.

Function Index

Function Details

flush/0

returns: ok if the data was written, or {error, Reason}, if there was an error.

Flush any previously written data to the console.

print/1

`Text`: the data to write to the console

returns: ok if the data was written, or {error, Reason}, if there was an error.

Write a string to the console.

See also: erlang:display/1.

puts/1

`Text`: the string data to write to the console

returns: ok if the data was written, or {error, Reason}, if there was an error.

Write a string to the console.

Note. *This operation will only write string data.The output is not suffixed with a newline character or sequence.To print an erlang term, use erlang:display/1.*

See also: erlang:display/1.

Module emscripten

- Description
- Data Types
- Function Index
- Function Details

emscripten API.

Description

The functions in this module broadly reflect emscripten's API and obviously are only implemented for the emscripten platform.

See Emscripten's API documentation for more information about these APIs.

The counterpart of functions defined in this module are two main Javascript functions that can be used to send messages to Erlang.

```
Module.cast('some_proc', 'message')
await Module.call('some_proc', 'message')
```

These respectively send the following messages to Erlang process registered as `some_proc`:

```
{emscripten, {cast, <<"message">>}}
{emscripten, {call, Promise, <<"message">>}}
```

Promise should be passed to `promise_resolve/1,2` or `promise_reject/1,2` as documented below.

Data Types

focus_event()

html5_target()

keyboard_event()

listener_handle()

abstract datatype: `listener_handle()`
mouse_event()

promise()

abstract datatype: `promise()`
register_error_reason()

register_option()

register_options()

register_result()

run_script_opt()

touch_event()

touch_point()

ui_event()

wheel_event()

Function Index

Function Details

promise_reject/1

Equivalent to `promise_reject(_Promise, 0).`

promise_reject/2

`_Promise`: Opaque promise resource`_Value`: Value to send to Javascript, must be an integer or a string.

Reject a promise with a given result. This is similar to `promise_resolve` except the promise is rejected.

promise_resolve/1

Equivalent to `promise_resolve(_Promise, 0).`

promise_resolve/2

`_Promise`: Opaque promise resource`_Value`: Value to send to Javascript, must be an integer or a string.

Successfully resolve a promise with a given result. A promise is currently only obtained through synchronous calls using `Module.call()` javascript function. If Javascript calls:

```
await Module.call('some_proc', 'message')
```

and if an Erlang process is registered as `some_proc`, then the process will receive a message:

and the Javascript caller will wait until `promise_resolve` or `promise_reject` is called. If the process doesn't exist, the promise will be rejected with 'no_proc'. Likewise if the Promise is garbage collected by the Erlang VM.

register_blur_callback/1

Equivalent to `register_blur_callback(_Target, [])`.

register_blur_callback/2

Register for blur events. Events are sent as:

```
{emscripten, {blur, focus_event()}}
```

See also: register_keypress_callback/2.

register_blur_callback/3

Register for blur events. Events are sent as:

```
{emscripten, {blur, focus_event()}, UserData}
```

See also: register_keypress_callback/2.

register_click_callback/1

Equivalent to `register_click_callback(_Target, [])`.

register_click_callback/2

Register for click events. Events are sent as:

```
{emscripten, {click, mouse_event()}}
```

See also: register_keypress_callback/2.

register_click_callback/3

Register for click events. Events are sent as:

```
{emscripten, {click, mouse_event()}, UserData}
```

See also: register_keypress_callback/2.

register_dblclick_callback/1

Equivalent to `register_dblclick_callback(_Target, [])`.

register_dblclick_callback/2

Register for dblclick events.

See also: register_click_callback/2.

register_dblclick_callback/3

Register for dblclick events.

See also: register_click_callback/2.

register_focus_callback/1

Equivalent to `register_focus_callback(_Target, [])`.

register_focus_callback/2

Register for focus events.

See also: register_blur_callback/2.

register_focus_callback/3

Register for focus events.

See also: register_blur_callback/2.

register_focusin_callback/1

Equivalent to `register_focusin_callback(_Target, [])`.

register_focusin_callback/2

Register for focusin events.

See also: register_blur_callback/2.

register_focusin_callback/3

Register for focusin events.

See also: register_blur_callback/2.

register_focusout_callback/1

Equivalent to `register_focusout_callback(_Target, [])`.

register_focusout_callback/2

Register for focusout events.

See also: register_blur_callback/2.

register_focusout_callback/3

Register for focusout events.

See also: register_blur_callback/2.

register_keydown_callback/1

Equivalent to `register_keydown_callback(_Target, [])`.

register_keydown_callback/2

Register for keydown events.

See also: register_keypress_callback/2.

register_keydown_callback/3

Register for keydown events.

See also: register_keypress_callback/2.

register_keypress_callback/1

Equivalent to `register_keypress_callback(_Target, [])`.

register_keypress_callback/2

Register for keypress events. This function registers with no user data and events are sent as:

```
{emscripten, {keypress, keyboard_event()}}
```

See also: register_keypress_callback/3.

register_keypress_callback/3

`_Target`: target to register keypress on_`Options`: options for event handling_`UserData`: user data passed back

Register for keypress events. This function registers keypress events on a given target. Target can be specified as special atoms `window` for Javascript's window, `document` for window.document. `screen` is also supported, but it doesn't seem to work, see `https://github.com/emscripten-core/emscripten/issues/19865`

Second parameter specifies options which can be a `boolean()` to match `useCapture` in Emscripten's API. Alternatively, the option can be a proplist() with `use_capture` and `prevent_default` keys. `prevent_default` determines what the handler should return to Javascript, `true` meaning that the default should be prevented.

Third parameter is user data that is passed back. Indeed, when an event occurs, the following message is sent to the process that registered the event:

```
{emscripten, {keypress, keyboard_event()}, UserData}
```

The function eventually returns a `listener_handle()` or an error. The handler is an opaque resource that actually contains a copy of `UserData`. Please note that if the calling process dies, the callback and any callback for the same event on the same target are unregistered.

register_keyup_callback/1

Equivalent to `register_keyup_callback(_Target, [])`.

register_keyup_callback/2

Register for keyup events.

See also: register_keypress_callback/2.

register_keyup_callback/3

Register for keyup events.

See also: register_keypress_callback/2.

register_mousedown_callback/1

Equivalent to `register_mousedown_callback(_Target, [])`.

register_mousedown_callback/2

Register for mousedown events.

See also: register_click_callback/2.

register_mousedown_callback/3

Register for mousedown events.

See also: register_click_callback/2.

register_mouseenter_callback/1

Equivalent to `register_mouseenter_callback(_Target, [])`.

register_mouseenter_callback/2

Register for mouseenter events.

See also: register_click_callback/2.

register_mouseenter_callback/3

Register for mouseenter events.

See also: register_click_callback/2.

register_mouseleave_callback/1

Equivalent to `register_mouseleave_callback(_Target, [])`.

register_mouseleave_callback/2

Register for mouseleave events.

See also: register_click_callback/2.

register_mouseleave_callback/3

Register for mouseleave events.

See also: register_click_callback/2.

register_mousemove_callback/1

Equivalent to `register_mousemove_callback(_Target, [])`.

register_mousemove_callback/2

Register for mousemove events.

See also: register_click_callback/2.

register_mousemove_callback/3

Register for mousemove events.

See also: register_click_callback/2.

register_mouseout_callback/1

Equivalent to `register_mouseout_callback(_Target, [])`.

register_mouseout_callback/2

Register for mouseout events.

See also: register_click_callback/2.

register_mouseout_callback/3

Register for mouseout events.

See also: register_click_callback/2.

register_mouseover_callback/1

Equivalent to `register_mouseover_callback(_Target, [])`.

register_mouseover_callback/2

Register for mouseover events.

See also: register_click_callback/2.

register_mouseover_callback/3

Register for mouseover events.

See also: register_click_callback/2.

register_mouseup_callback/1

Equivalent to `register_mouseup_callback(_Target, [])`.

register_mouseup_callback/2

Register for mouseup events.

See also: register_click_callback/2.

register_mouseup_callback/3

Register for mouseup events.

See also: register_click_callback/2.

register_resize_callback/1

Equivalent to `register_resize_callback(_Target, [])`.

register_resize_callback/2

Register for resize events. Events are sent as:

```
{emscripten, {resize, ui_event()}}
```

See also: register_keypress_callback/2.

register_resize_callback/3

Register for resize events. Events are sent as:

```
{emscripten, {resize, ui_event()}, UserData}
```

See also: register_keypress_callback/2.

register_scroll_callback/1

Equivalent to `register_scroll_callback(_Target, [])`.

register_scroll_callback/2

Register for scroll events.

See also: register_resize_callback/2.

register_scroll_callback/3

Register for scroll events.

See also: register_resize_callback/2.

register_touchcancel_callback/1

Equivalent to `register_touchcancel_callback(_Target, [])`.

register_touchcancel_callback/2

Register for touchcancel events.

See also: register_touchstart_callback/2.

register_touchcancel_callback/3

Register for touchcancel events.

See also: register_touchstart_callback/2.

register_touchend_callback/1

Equivalent to `register_touchend_callback(_Target, [])`.

register_touchend_callback/2

Register for touchend events.

See also: register_touchstart_callback/2.

register_touchend_callback/3

Register for touchend events.

See also: register_touchstart_callback/2.

register_touchmove_callback/1

Equivalent to `register_touchmove_callback(_Target, [])`.

register_touchmove_callback/2

Register for touchmove events.

See also: register_touchstart_callback/2.

register_touchmove_callback/3

Register for touchmove events.

See also: register_touchstart_callback/2.

register_touchstart_callback/1

Equivalent to `register_touchstart_callback(_Target, [])`.

register_touchstart_callback/2

Register for touchstart events. Events are sent as:

```
{emscripten, {touchstart, touch_event()}}
```

See also: register_keypress_callback/2.

register_touchstart_callback/3

Register for touchstart events. Events are sent as:

```
{emscripten, {touchstart, touch_event()}, UserData}
```

See also: register_keypress_callback/2.

register_wheel_callback/1

Equivalent to `register_wheel_callback(_Target, [])`.

register_wheel_callback/2

Register for wheel events. Events are sent as:

```
{emscripten, {wheel, wheel_event()}}
```

See also: register_keypress_callback/2.

register_wheel_callback/3

Register for wheel events. Events are sent as:

```
{emscripten, {wheel, mouse_event()}, UserData}
```

See also: register_keypress_callback/2.

run_script/1

Equivalent to `run_script(_Script, [])`.

run_script/2

`_Script`: Script to run`_Options`: List of options. If `main_thread` is specified, the script is run on the main thread. If `async`, the script is run asynchronously, i.e. the caller does not wait for completion. Only applies if `main_thread` is specified.

returns: ok

Run a script. By default, the script is run in the current worker thread, which arguably may not be very useful. Please note that exception handling is disabled, so the script should not throw and should compile, otherwise this will crash the VM.

unregister_blur_callback/1

Unregister a blur event handler.

See also: unregister_keypress_callback/1.

unregister_click_callback/1

Unregister a click event handler.

See also: unregister_keypress_callback/1.

unregister_dblclick_callback/1

Unregister a dblclick event handler.

See also: unregister_click_callback/1.

unregister_focus_callback/1

Unregister a focus event handler.

See also: unregister_blur_callback/1.

unregister_focusin_callback/1

Unregister a focusin event handler.

See also: unregister_blur_callback/1.

unregister_focusout_callback/1

Unregister a focusout event handler.

See also: unregister_blur_callback/1.

unregister_keydown_callback/1

Unregister a keydown event handler.

See also: unregister_keypress_callback/1.

unregister_keypress_callback/1

`_TargetOrHandle`: Target or handle

returns: ok or an error

Unregister a keypress listener.

To match Emscripten's API, this function can take a target. This function unregisters every keypress listeners on the specified target.

Alternatively, this function can take a listener_handle() returned by `register_keypress_callback/1,2,3`. This will still unregister every keypress listeners on the same target.

Passing a `listener_handle()` is recommended to avoid surprises and for memory efficiency. If a handle is passed, it can then be garbage collected. If a handle is not passed, but the process dies, every keypress listener on the same target will be unregistered, including listeners that were later registered. This is a known limitation of the implementation that favored avoiding memory leaks and crashes.

unregister_keyup_callback/1

Unregister a keyup event handler.

See also: unregister_keypress_callback/1.

unregister_mousedown_callback/1

Unregister a mousedown event handler.

See also: unregister_click_callback/1.

unregister_mouseenter_callback/1

Unregister a mouseenter event handler.

See also: unregister_click_callback/1.

unregister_mouseleave_callback/1

Unregister a mouseleave event handler.

See also: unregister_click_callback/1.

unregister_mousemove_callback/1

Unregister a mousemove event handler.

See also: unregister_click_callback/1.

unregister_mouseout_callback/1

Unregister a mouseout event handler.

See also: unregister_click_callback/1.

unregister_mouseover_callback/1

Unregister a mouseover event handler.

See also: unregister_click_callback/1.

unregister_mouseup_callback/1

Unregister a mouseup event handler.

See also: unregister_click_callback/1.

unregister_resize_callback/1

Unregister a resize event handler.

See also: unregister_keypress_callback/1.

unregister_scroll_callback/1

Unregister a scroll event handler.

See also: unregister_resize_callback/1.

unregister_touchcancel_callback/1

Unregister a touchcancel event handler.

See also: unregister_touchstart_callback/1.

unregister_touchend_callback/1

Unregister a touchend event handler.

See also: unregister_touchstart_callback/1.

unregister_touchmove_callback/1

Unregister a touchmove event handler.

See also: unregister_touchstart_callback/1.

unregister_touchstart_callback/1

Unregister a touchstart event handler.

See also: unregister_keypress_callback/1.

unregister_wheel_callback/1

Unregister a wheel event handler.

See also: unregister_keypress_callback/1.

Module epmd

- Data Types
- Function Index
- Function Details

Behaviours: `gen_server`.

Data Types

epmd_config()

epmd_config_option()

Function Index

Function Details

start_link/1

Module esp

- Description
- Data Types
- Function Index
- Function Details

ESP32-specific APIs.

Description

This module contains functions that are specific to the ESP32 platform.

Data Types

esp_partition()

esp_partition_address()

esp_partition_props()

esp_partition_size()

esp_partition_subtype()

esp_partition_type()

esp_reset_reason()

esp_wakeup_cause()

interface()

mac()

mounted_fs()

abstract datatype: `mounted_fs()`

task_wdt_config()

task_wdt_user_handle()

abstract datatype: `task_wdt_user_handle()`

Function Index

Function Details

deep_sleep/0

Put the esp32 into deep sleep. This function never returns. Program is restarted and wake up reason can be inspected to determine how the esp32 was woken up.

deep_sleep/1

`SleepMS`: time to deep sleep in milliseconds

Put the esp32 into deep sleep. This function never returns. Program is restarted and wake up reason can be inspected to determine if the esp32 was woken by the timeout or by another cause.

deep_sleep_enable_gpio_wakeup/2

`Mask`: bit mask of GPIO numbers which will cause wakeup`Mode`:

level that will trigger the wakeup.

The available modes are:

returns: `ok | error`

Configure multiple gpio pins for wakeup from deep sleep. Implemented for SOCs that support it (ESP32C2, ESP32C3, ESP32C6, ESP32P4)

freq_hz/0

returns: Clock frequency (in hz)

Return the clock frequency on the chip

get_default_mac/0

returns: The default MAC address of the ESP32 device.

Retrieve the default MAC address of the ESP32 device. This function accesses the EFUSE memory of the ESP32 and reads the factory-programmed MAC address.

The mac address is returned as a 6-byte binary, per the IEEE 802 family of specifications.

get_mac/1

`Interface`: the ESP32 network interface

returns: The network MAC address of the specified interface

Return the network MAC address of the specified interface.

The mac address is returned as a 6-byte binary, per the IEEE 802 family of specifications.

mount/4

`Source`: the device that will be mounted`Target`: the path where the filesystem will be mounted`FS`: the filesystem, only fat is supported now

returns: either a tuple having `ok` and the mounted fs resource, or an error tuple

Mount a filesystem, and return a resource that can be used later for unmounting it

nvs_erase_all/0

This function is deprecated: Please do not use this function.

Equivalent to nvs_erase_all(?ATOMVM_NVS_NS).

nvs_erase_all/1

`Namespace`: NVS namespace

returns: ok

Erase all values in the specified namespace.

nvs_erase_key/1

`Key`: NVS key

returns: ok

This function is deprecated: Please do not use this function.

Equivalent to nvs_erase_key(?ATOMVM_NVS_NS, Key).

nvs_erase_key/2

`Namespace`: NVS namespace`Key`: NVS key

returns: ok

Erase the value associated with a key. If a value does not exist for the specified key, no action is performed.

nvs_fetch_binary/2

`Namespace`: NVS namespace`Key`: NVS key

returns: tagged tuple with binary value associated with this key in NV storage, {error, not_found} if there is no value associated with this key, or in general {error, Reason} for any other error.

Get the binary value associated with a key, or undefined, if there is no value associated with this key.

nvs_get_binary/1

This function is deprecated: Please do not use this function.

Equivalent to nvs_get_binary(?ATOMVM_NVS_NS, Key).

nvs_get_binary/2

`Namespace`: NVS namespace`Key`: NVS key

returns: binary value associated with this key in NV storage, or undefined if there is no value associated with this key.

Get the binary value associated with a key, or undefined, if there is no value associated with this key.

nvs_get_binary/3

`Namespace`: NVS namespace`Key`: NVS key`Default`: default binary value, if Key is not set in Namespace

returns: binary value associated with this key in NV storage, or Default if there is no value associated with this key.

Get the binary value associated with a key, or Default, if there is no value associated with this key.

nvs_put_binary/3

`Namespace`: NVS namespace`Key`: NVS key`Value`: binary value

returns: ok

Set an binary value associated with a key. If a value exists for the specified key, it is over-written.

nvs_reformat/0

returns: ok

Reformat the entire NVS partition. WARNING. This will result in deleting all NVS data and should be used with extreme caution!

nvs_set_binary/2

This function is deprecated: Please use nvs_put_binary instead.

Equivalent to nvs_set_binary(?ATOMVM_NVS_NS, Key, Value).

nvs_set_binary/3

`Namespace`: NVS namespace`Key`: NVS key`Value`: binary value

returns: ok

This function is deprecated: Please use nvs_put_binary instead.

Set an binary value associated with a key. If a value exists for the specified key, it is over-written.

partition_erase_range/2

returns: ok on success, error on failure

Erases a range in the specified partition. The range starts at the given offset and continues to the end of the partition. This is equivalent to calling partition_erase_range/3 with size set to the remaining partition size from the offset.

Note: Erasing sets all bits in the erased range to 1s. This operation is irreversible.

partition_erase_range/3

returns: ok on success, error on failure

Erases a range of the specified size in the partition, starting at the given offset. The size must not exceed the partition's boundaries.

Note: Erasing sets all bits in the erased range to 1s. This operation is irreversible. Make sure the offset and size parameters are valid for the target partition to avoid errors.

partition_list/0

returns: List of partitions

Gets the list of partitions as tuples, such as {name, type, subtype, offset, size, props}. Type and subtype are integers as described in esp-idf documentation.

partition_read/3

returns: {ok, data} on success, error on failure

Read binary data from a specific partition at the given offset.

partition_write/3

returns: ok on success, error on failure

Writes binary data to a specific partition at the given offset.

This function allows writing data to ESP32 flash partitions. Care must be taken when using this function as improper writes can corrupt the partition table or system firmware. Make sure the offset and data size do not exceed the partition's boundaries.

reset_reason/0

returns: the reason for the restart

Returns the reason for the restart

restart/0

Restarts the ESP device

rtc_slow_get_binary/0

returns: the currently stored binary in RTC slow memory.

Get the binary currently stored in RTC slow memory. Must not be called unless the binary was stored with rtc_slow_set_binary/1. A limited checksum is ran and this function may throw badarg if the checksum is not valid.

rtc_slow_set_binary/1

`Bin`: binary to be stored in RTC slow memory

returns: ok

Store a binary to RTC slow memory. This memory is not erased on software reset and deep sleeps.

sleep_disable_ext1_wakeup_io/1

`Mask`: bit mask of GPIO numbers to reset.

returns: `ok | error`

Unconfigure one or more gpio pins for wakeup from deep sleep. Implemented for SOCs that support it (ESP32, ESP32S2, ESP32S3, ESP32C6, ESP32H2). This function resets pins previously configured with `sleep_enable_ext1_wakeup_io`.

Introduced in: ESP-IDF 5.3 and maybe 5.2.2

sleep_enable_ext0_wakeup/2

`Pin`: number of the pin to use as wakeup event`Level`: is the state to trigger a wakeup

returns: `ok | error`

Configure gpio wakeup from deep sleep. Implemented for SOCs that support it (ESP32, ESP32S2, ESP32S3)

sleep_enable_ext1_wakeup/2

`Mask`: bit mask of GPIO numbers which will cause wakeup`Mode`:

used to determine wakeup events

The available modes are:

returns: `ok | error`

This function is deprecated: This function will be removed in ESP-IDF 6.0. Use `sleep_enable_ext1_wakeup_io` instead.

Configure multiple gpio pins for wakeup from deep sleep. Implemented for SOCs that support it (ESP32, ESP32S2, ESP32S3, ESP32C6, ESP32H2).

sleep_enable_ext1_wakeup_io/2

`Mask`: bit mask of GPIO numbers which will cause wakeup`Mode`:

used to determine wakeup events

The available modes are:

returns: `ok | error`

Configure multiple gpio pins for wakeup from deep sleep. Implemented for SOCs that support it (ESP32, ESP32S2, ESP32S3, ESP32C6, ESP32H2). This function does not reset the previously set pins. On some SOCs (ESP32C6, ESP32H2), pins can be configured with different levels.

Introduced in: ESP-IDF 5.3 and maybe 5.2.2

sleep_enable_ulp_wakeup/0

returns: Enable ulp wakeup

sleep_get_wakeup_cause/0

returns: the cause for the wake up

Returns the cause for the wakeup

task_wdt_add_user/1

`Username`: name of the user

returns: the handle to use with `task_wdt_reset_user/1` or an error tuple.

Register a user of the task watchdog timer. Available with ESP-IDF 5.0 or higher.

task_wdt_deinit/0

returns: ok or an error tuple if tasks are subscribed (beyond idle tasks) or if the timer is not initialized

Deinitialize the task watchdog timer Available with ESP-IDF 5.0 or higher.

task_wdt_delete_user/1

`UserHandle`: handle for the user, obtained from `task_wdt_add_user/1`

returns: ok or an error tuple

Unsubscribe a given user from the task watchdog timer. Available with ESP-IDF 5.0 or higher.

task_wdt_init/1

`Config`: configuration for the watchdog timer

returns: ok or an error tuple

Initialize the task watchdog timer with a configuration Available with ESP-IDF 5.0 or higher.

task_wdt_reconfigure/1

`Config`: configuration for the watchdog timer

returns: ok or an error tuple

Update the configuration of the task watchdog timer Available with ESP-IDF 5.0 or higher.

task_wdt_reset_user/1

`UserHandle`: handle for the user, obtained from `task_wdt_add_user/1`

returns: ok or an error tuple

Reset the timer a previously registered user. Available with ESP-IDF 5.0 or higher.

umount/1

returns: either `ok` or an error tuple

Unmounts filesystem located at given path

Module esp_adc

- Description
- Data Types
- Function Index
- Function Details

Analog to digital peripheral support.

Behaviours: `gen_server`.

Description

Use this module to take ADC (analog voltage) readings. Currently this driver only supports the esp32 family of chips, but support for other platforms is planned in the future. On an ESP32 device ADC unit 1 allows taking reading from pins 32-39. ADC unit2 is disabled by default for the ESP32 classic, but when enabled in the build configuration allows pins 0, 2, 4, 12-15, and 25-27 to be used as long as WiFi is not required by the application. Unit 2 is disabled for ESP32C3 due to its inaccurate results. ADC unit 2 is enabled for all other ESP32 series with more than one ADC unit; there is an arbitrator peripheral that allows ADC unit 2 to be used while WiFI is active. The pins available for ADC use vary by device, check your datasheet for specific hardware support.

There are two sets of APIs for interacting with the ADC hardware, only one set of API may be used by an application.

The core functionality is provided by the low level resource based nif functions. To use the resource based nifs `esp_adc:init/0` and `esp_adc:deinit/1` will acquire and release the adc `unit` resource needed for all other functions. A `channel` resource used to take measurements from a pin can be obtained using `esp_adc:acquire/4`, and released using `esp_adc:release_channel/1`. ADC measurements are taken using `sample/3`.

For convenience a gen_server managed set of APIs using pin numbers are also available. A pin may be configured using `esp_adc:start/1,2`, measurements are taken using `esp_adc:read/1,2`, pins can be released individually with `esp_adc:stop/1`, or the driver can be stopped completely using `esp_adc:stop/0`.

Data Types

adc_pin()

ADC capable pins vary by chipset. Consult your datasheet.

adc_rsrc()

attenuation()

The decibel gain determines the maximum safe voltage to be measured. Default is `db_11`. The specific range of voltages supported by each setting varies by device. Typical voltage ranges are depicted in the table below:

Consult the datasheet for your device to determine the exact voltage ranges supported by each gain setting. The option `db_11` has been superseded by `db_12`. The option`db_11` and will be deprecated in a future release, applications should be updated to use `db_12` (except for builds with ESP-IDF versions prior to v5.2). To Continue to support older IDF version builds, the default will remain `db_11`, which is the maximum tolerated voltage on all builds, as `db_12` supported builds will automatically use `db_12` in place of `db_11`, when `db_11` is deprecated in all builds the default will be changed to `db_12`.

bit_width()

The default `bit_max` will select the highest value supported by the chipset. Some models only support a single fixed bit width.

pin_option()

pin_options()

raw_value()

The maximum analog value is determined by `bit_width()`.

read_option()

The value of the `samples` key is the number of samples to be taken and averaged when returning a measurement, default is 64. For optimal stable readings use a 100nF ceramic capacitor input filter, for more info consult Espressif's "ADC Calibration Driver" documentation. The keys `raw` and `voltage` determine if these values are included in the results or returned as `undefined`.

read_options()

reading()

voltage_reading()

The maximum safe millivolt value that can be measured is determined by `attenuation()`, this value should never exceed the chips maximum input tolerance.

Function Index

Function Details

acquire/2

`Pin`: Pin to configure as ADC`UnitHandle`: The unit handle returned from `init/0`

returns: {ok, Channel::adc_rsrc()} | {error, Reason}

Equivalent to `acquire(Pin, UnitHandle, bit_max, db_11)`.

Nif to initialize an ADC pin.

Initializes an ADC pin and returns a channel handle resources.

This is a low level nif that cannot be used in an application that uses the convenience functions.

acquire/4

`Pin`: Pin to configure as ADC`UnitHandle`: The unit handle returned from `init/0BitWidth`: Resolution in bit to measure`Attenuation`: Decibel gain for voltage range

returns: {ok, Channel::adc_rsrc()} | {error, Reason}

Nif to initialize an ADC pin.

The BitWidth value `bit_max` may be used to automatically select the highest sample rate supported by your ESP chip-set, or choose from a bit width supported by the device.

The Attenuation value can be used to adust the gain, and therefore safe measurement range on voltage the exact range of voltages supported by each db gain varies by chip, consult the data sheet for exact range of your model. For more information see the `attenuation()` type specification.

Use the returned `Channel` reference in subsequent ADC operations on the same pin.

This is a low level nif that cannot be used in an application that uses the convenience functions.

deinit/1

`UnitResource`: returned from init/0

returns: ok | {error, Reason}

Nif to release the ADC unit resource returned from init/0.

Stop the ADC driver and free the unit resource. All active ADC channels should be released using `release_channel/1` to free each configured channel before freeing the unit resource.

This is a low level nif that cannot be used in an application that uses the convenience functions.

init/0

returns: {ok, ADCUnit :: adc_rsrc()} | {error, Reason}

Nif to initialize the ADC unit hardware.

The returned ADC unit handle resource must be supplied for all future ADC operations.

This is a low level nif that cannot be used in an application that uses the convenience functions.

read/1

`Pin`: The pin from which to take ADC measurement

returns: {ok, {RawValue, MilliVolts}} | {error, Reason}

Equivalent to `read(Pin, [raw, voltage, {samples, 64}])`.

Take a reading using default values from an ADC pin.

This convenience function is used to take a measurement from a previously started adc pin.

This function cannot be used in an application that uses the low level nif APIs.

read/2

`Pin`: The pin from which to take ADC measurement`ReadOptions`: Extra options

returns: {ok, {RawValue, MilliVolts}} | {error, Reason}

Take a reading from an ADC pin using the supplied options.

This convenience function is used to take a measurement from a previously started adc pin, using the supplied read options parameter.

The Options parameter may be used to specify the behavior of the read operation.

If the ReadOptions contains the atom `raw`, then the raw value will be returned in the first element of the returned tuple. Otherwise, this element will be the atom `undefined`.

If the ReadOptions contains the atom `voltage`, then the voltage value will be returned in millivolts in the second element of the returned tuple. Otherwise, this element will be the atom `undefined`.

You may specify the number of samples to be taken and averaged over using the tuple `{samples, Samples::pos_integer()}`, the default is `64`.

If the error `Reason` is timeout and the adc channel is on unit 2 then WiFi is likely enabled and adc2 readings may be blocked until there is less network traffic. On and ESP32 classic the results for unit 2 will always be `{error, timeout}` if wifi is enabled.

This function cannot be used in an application that uses the low level nif APIs.

release_channel/1

`ChannelResource`: of the pin returned from acquire/4

returns: ok | {error, Reason}

Nif to deinitialize the specified ADC channel.

In the case that an error is returned it is safe to "drop" the `ChannelResource` handle from use. After there are no remaining processes with references to the channel resource handle, the calibration profile and any remaining resources associated with the channel will be released as part of the next garbage collection event.

This is a low level nif that cannot be used in an application that uses the convenience functions.

sample/2

`ChannelResource`: of the pin returned from acquire/4`UnitResource`: of the pin returned from init/0

returns: {ok, {RawValue, MilliVolts}} | {error, Reason}

Equivalent to `sample(ChannelResource,UnitResource,[raw, voltage, {samples, 64}])`.

Nif to take a reading using default values from an ADC channel.

This is a low level nif that cannot be used in an application that uses the convenience functions.

sample/3

`ChannelResource`: of the pin returned from acquire/4`UnitResource`: of the pin returned from init/0`ReadOptions`: extra list of options to override defaults.

returns: {ok, {RawValue, MilliVolts}} | {error, Reason}

Nif to take a reading from an ADC channel.

The Options parameter may be used to specify the behavior of the read operation.

If the ReadOptions contains the atom `raw`, then the raw value will be returned in the first element of the returned tuple. Otherwise, this element will be the atom `undefined`.

If the ReadOptions contains the atom `voltage`, then the voltage value will be returned in millivolts in the second element of the returned tuple. Otherwise, this element will be the atom `undefined`.

You may specify the number of samples to be taken and averaged over using the tuple `{samples, Samples::pos_integer()}`.

If the error `Reason` is timeout and the adc channel is on unit 2 then WiFi is likely enabled and adc2 readings may be blocked until there is less network traffic. On and ESP32 classic the results for unit 2 will always be `{error, timeout}` if wifi is enabled.

This is a low level nif that cannot be used in an application that uses the convenience functions.

start/0

returns: {ok, Pid}

Optionally initialize a gen_server managed ADC driver without a pin.

Use of this function is optional, but may be desired if the drivers pid is needed, or it is desireable to start the driver without configuring an initial ADC channel.

Note: since only one instance of the driver is allowed it is registered with the name `adc_driver`, which also may be used to directly call the gen_server.

This convenience function cannot be used in an application that uses the low level nif APIs.

start/1

`Pin`: Pin to configure as ADC

returns: ok | {error, Reason}

Equivalent to `start(Pin, [{bitwidth, bit_max}, {atten, db_11}])`.

Initialize a gen_server managed ADC pin with default options.

This convenience function configures an ADC pin with the default options for use with the optional `gen_server` APIs. Default options are: `[{bitwidth, bit_max}, {atten, db_11}]`

This function cannot be used in an application that uses the low level nif APIs.

start/2

`Pin`: Pin to configure as ADC`Options`: List of options to override default settings

returns: ok | {error, Reason}

Initialize a gen_server managed ADC pin with the supplied options.

This convenience function configures an ADC pin with the provided options to override the default configuration: `[{bitwidth, bit_max}, {atten, db_11}]`.

For more details about these options see the `attenuation()` and `bit_width()` type specifications.

This function cannot be used in an application that uses the low level nif APIs.

stop/0

returns: ok | {error, Reason}

Stop the ADC driver and release all resources.

This convenience function is used to completely stop the gen_server managed ADC driver and release all resources.

Note: if an error is returned, a full shutdown of the ADC peripheral should still occur, and any remaining resources freed with next VM garbage collection event. Regardless the gen_server will exit normally and the adc peripheral will no longer be usable.

This function cannot be used in an application that uses the low level nif APIs.

stop/1

`Pin`: the pin to be released

returns: ok | {error, Reason}

De-initialize the specified ADC pin.

This convenience function is used to release a pin from the gen_server managed ADC driver. If an error is returned the ADC channel will still be stopped and release internal resources during the next VM garbage collection event, the pin will immediately no longer be useable in any case.

This function cannot be used in an application that uses the low level nif APIs.

Module gpio

GPIO driver module.

Description

This module provides functions for interacting with micro-controller GPIO (General Purpose Input and Output) pins.

Note: `-type pin()` used in this driver refers to a pin number on Espressif chips and normal Raspberry Pi Pico pins, or a tuple {GPIO_BANK, PIN} for STM32 chips and the "extra" GPIOs available on the Pico-W.

Data Types

direction()

The direction is used to set the mode of operation for a GPIO pin, either as an input, an output, or output with open drain. On the STM32 platform pull mode and output_speed must be set at the same time as direction. See @type mode_config()

gpio()

This is the pid returned by `gpio:start/0`.

gpio_bank()

STM32 gpio banks vary by board, some only break out `a` thru `h`. The extra "WL" pins on Pico-W use bank `wl`.

high_level()

level()

Valid pin levels can be atom or binary representation.

low_level()

mode_config()

Extended mode configuration options on STM32. Default pull() is `floating`, default output_speed() is `mhz_2` if options are omitted.

output_speed()

Output clock speed. Only available on STM32, default is `mhz_2`.

pin()

The pin definition for ESP32 and PR2040 is a non-negative integer. A tuple is used on the STM32 platform and for the extra "WL" pins on the Pico-W.

pin_tuple()

A pin parameter on STM32 is a tuple consisting of a GPIO bank and pin number, also used on the Pico-W for the extra "WL" pins `0..2`.

pull()

Internal resistor pull mode. STM32 does not support `up_down`.

trigger()

Event type that will trigger a `gpio_interrupt`. STM32 only supports `rising`, `falling`, or `both`.

Function Index

Function Details

attach_interrupt/2

`Pin`: number of the pin to set the interrupt on`Trigger`: is the state that will trigger an interrupt

returns: ok | error | {error, Reason}

Convenience function for `gpio:set_int/3`

This is a convenience function for `gpio:set_int/3` that allows an interrupt to be set using only the pin number and trigger as arguments.

This function should only be used when only one gpio trigger is used in an application. If multiple pins are being configured with interrupt triggers gpio:set_int/3 should be used otherwise there is a race condition when start() is called internally by this function.

The rp2040 (Pico) port does not support gpio interrupts at this time.

close/1

`GPIO`: pid that was returned from gpio:start/0

returns: ok | error | {error, Reason}

Stop the GPIO interrupt port

This function disables any interrupts that are set, stops the listening port, and frees all of its resources.

Not currently available on rp2040 (Pico) port, use nif functions.

deep_sleep_hold_dis/0

returns: ok

Disable all gpio pad functions during Deep-sleep.

This function is only supported on ESP32.

deep_sleep_hold_en/0

returns: ok

Enable all hold functions to continue in deep sleep.

The gpio pad hold function works in both input and output modes, but must be output-capable gpios.

When the chip is in Deep-sleep mode, all digital gpio will hold the state before sleep, and when the chip is woken up, the status of digital gpio will not be held. Note that the pad hold feature only works when the chip is in Deep-sleep mode, when not in sleep mode, the digital gpio state can be changed even you have called this function.

Power down or call `gpio_hold_dis` will disable this function, otherwise, the digital gpio hold feature works as long as the chip enters Deep-sleep.

This function is only supported on ESP32.

deinit/1

`Pin`: number to deinitialize

returns: ok

Reset a pin back to the NULL function. Currently only implemented for RP2040 (Pico).

detach_interrupt/1

`Pin`: number of the pin to remove the interrupt

returns: ok | error | {error, Reason}

Convenience function for `gpio:remove_int/2`

This is a convenience function for `gpio:remove_int/2` that allows an interrupt to be removed using only the pin number as an argument.

Unlike `gpio:attach_interrupt/2` this function can be safely used regardless of the number of interrupt pins used in the application.

The rp2040 (Pico) port does not support gpio interrupts at this time.

digital_read/1

`Pin`: number of the pin to read

returns: high | low | error | {error, Reason}

Read the digital state of a GPIO pin

Read if an input pin state is high or low. Warning: if the pin was not previously configured as an input using `gpio:set_pin_mode/2` it will always read as low.

The VBUS detect pin on the Pico-W can be read on the extended pin `{wl, 2}`, and does not require or accept `set_pin_mode` or `set_pin_pull` before use.

digital_write/2

`Pin`: number of the pin to write`Level`: the desired output level to set

returns: ok | error | {error, Reason}

Set GPIO digital output level

Set a pin to `high` (1) or `low` (0).

The STM32 is capable of setting the state for any, or all of the output pins on a single bank at the same time, this is done by passing a list of pins numbers in the pin tuple. For example, setting all of the even numbered pins to a `high` state, and all of the odd numbered pins to a `low` state can be accomplished in two lines:

```
gpio:digital_write({c, [0,2,4,6,8,10,12,14]}, high}),
gpio:digital_write({c, [1,3,5,7,9,11,13,15]}, low}).
```

To set the same state for all of the pins that have been previously configured as outputs on a specific bank the atom `all` may be used, this will have no effect on any pins on the same bank that have been configured as inputs, so it is safe to use with mixed direction modes on a bank.

The LED pin on the Pico-W can be controlled on the extended pin `{wl, 0}`, and does not require or accept `set_pin_mode` or `set_pin_pull` before use.

hold_dis/1

`Pin`: number of the pin to be released

returns: ok | error

Release a pin from a hold state.

When the chip is woken up from Deep-sleep, the gpio will be set to the default mode, so, the gpio will output the default level if this function is called. If you don't want the level changes, the gpio should be configured to a known state before this function is called. e.g. If you hold gpio18 high during Deep-sleep, after the chip is woken up and `gpio:hold_dis` is called, gpio18 will output low level(because gpio18 is input mode by default). If you don't want this behavior, you should configure gpio18 as output mode and set it to hight level before calling `gpio:hold_dis`.

This function is only supported on ESP32.

hold_en/1

`Pin`: number of the pin to be held

returns: ok | error

Hold the state of a pin

The gpio pad hold function works in both input and output modes, but must be output-capable gpios.

If pad hold enabled: In output mode: the output level of the pad will be force locked and can not be changed. In input mode: the input value read will not change, regardless the changes of input signal.

The state of digital gpio cannot be held during Deep-sleep, and it will resume the hold function when the chip wakes up from Deep-sleep. If the digital gpio also needs to be held during Deep-sleep `gpio:deep_sleep_hold_en` should also be called.

This function is only supported on ESP32.

init/1

`Pin`: number to initialize

returns: ok

Initialize a pin to be used as GPIO. This is required on RP2040 and for some pins on ESP32.

open/0

returns: Pid | error | {error, Reason}

Start the GPIO driver port

The GPIO port driver will be stared and registered as `gpio`. If the port has already been started through the `gpio:open/0` or `gpio:start/0` the command will fail. The use of `gpio:open/0` or `gpio:start/0` is required before using any functions that require a GPIO pid as a parameter.

Not currently available on rp2040 (Pico) port, use nif functions.

read/2

`GPIO`: pid that was returned from gpio:start/0`Pin`: number of the pin to read

returns: high | low | error | {error, Reason}

Read the digital state of a GPIO pin

Read if an input pin state is `high` or `low`. Warning: if the pin was not previously configured as an input using `gpio:set_direction/3` it will always read as low.

Not supported on rp2040 (Pico), use `gpio:digital_read/1` instead.

remove_int/2

`GPIO`: pid that was returned from `gpio:start/0``Pin`: number of the pin to remove the interrupt

returns: ok | error | {error, Reason}

Remove a GPIO interrupt

Removes an interrupt from the specified pin.

The rp2040 (Pico) port does not support gpio interrupts at this time.

set_direction/3

`GPIO`: pid that was returned from `gpio:start/0Pin`: number of the pin to configure`Direction`: is `input`, `output`, or `output_od`

returns: ok | error | {error, Reason}

Set the operational mode of a pin

Pins can be used for input, output, or output with open drain.

The STM32 platform has extended direction mode configuration options. See @type mode_config() for details. All configuration must be set using `set_direction/3`, including pull() mode, unlike the ESP32 which has a separate function (`set_pin_pull/2`). If you are configuring multiple pins on the same GPIO `bank` with the same options the pins may be configured all at the same time by giving a list of pin numbers in the pin tuple.

Example to configure all of the leds on a Nucleo board:

```
    gpio:set_direction({b, [0,7,14], output)
```

Not supported on rp2040 (Pico), use `gpio:set_pin_mode/2` instead.

set_int/3

`GPIO`: pid that was returned from `gpio:start/0Pin`: number of the pin to set the interrupt on`Trigger`: is the state that will trigger an interrupt

returns: ok | error | {error, Reason}

Set a GPIO interrupt

Available triggers are `none` (which is the same as disabling an interrupt), `rising`, `falling`, `both` (rising or falling), `low`, and `high`. When the interrupt is triggered it will send a tuple: `{gpio_interrupt, Pin}` to the process that set the interrupt. `Pin` will be the number of the pin that triggered the interrupt.

The STM32 port only supports `rising`, `falling`, or `both`.

The rp2040 (Pico) port does not support gpio interrupts at this time.

set_int/4

`GPIO`: pid that was returned from `gpio:start/0Pin`: number of the pin to set the interrupt on`Trigger`: is the state that will trigger an interrupt`Pid`: is the process that will receive the interrupt message

returns: ok | error | {error, Reason}

Set a GPIO interrupt

Available triggers are `none` (which is the same as disabling an interrupt), `rising`, `falling`, `both` (rising or falling), `low`, and `high`. When the interrupt is triggered it will send a tuple: `{gpio_interrupt, Pin}` to the process that set the interrupt. Pin will be the number of the pin that triggered the interrupt.

The STM32 port only supports `rising`, `falling`, or `both`.

The rp2040 (Pico) port does not support gpio interrupts at this time.

set_level/3

`GPIO`: pid that was returned from `gpio:start/0Pin`: number of the pin to write`Level`: the desired output level to set

returns: ok | error | {error, Reason}

Set GPIO digital output level

Set a pin to `high` (1) or `low` (0).

The STM32 is capable of setting the state for any, or all of the output pins on a single bank at the same time, this is done by passing a list of pins numbers in the pin tuple.

For example, setting all of the even numbered pins to a `high` state, and all of the odd numbered pins to a `low` state can be accomplished in two lines:

```
    gpio:digital_write({c, [0,2,4,6,8,10,12,14]}, high}),
    gpio:digital_write({c, [1,3,5,7,9,11,13,15]}, low}).
```

To set the same state for all of the pins that have been previously configured as outputs on a specific bank the atom `all` may be used, this will have no effect on any pins on the same bank that have been configured as inputs, so it is safe to use with mixed direction modes on a bank.

Not supported on rp2040 (Pico), use `gpio:digital_write/2` instead.

set_pin_mode/2

`Pin`: number to set operational mode`Direction`: is `input`, `output`, or `output_od`

returns: ok | error | {error, Reason}

Set the operational mode of a pin

Pins can be used for input, output, or output with open drain.

The STM32 platform has extended direction mode configuration options. See @type mode_config() for details. All configuration must be set using `set_direction/3`, including pull() mode, unlike the ESP32 which has a separate function (`set_pin_pull/2`). If you are configuring multiple pins on the same GPIO `bank` with the same options the pins may be configured all at the same time by giving a list of pin numbers in the pin tuple. Example to configure all of the leds on a Nucleo board:

```
    gpio:set_direction({b, [0,7,14], output)
```

set_pin_pull/2

`Pin`: number to set internal resistor direction`Pull`: is the internal resistor state

returns: ok | error

Set the internal resistor of a pin

Pins can be internally pulled `up`, `down`, `up_down` (pulled in both directions), or left `floating`.

This function is not supported on STM32, the internal resistor must be configured when setting the direction mode, see `set_direction/3` or `set_pin_mode/2`.

start/0

returns: Pid | error | {error, Reason}

Start the GPIO driver port

Returns the pid of the active GPIO port driver, otherwise the GPIO port driver will be stared and registered as `gpio`. The use of `gpio:open/0` or `gpio:start/0` is required before using any functions that require a GPIO pid as a parameter.

Not currently available on rp2040 (Pico) port, use nif functions.

stop/0

returns: ok | error | {error, Reason}

Stop the GPIO interrupt port

This function disables any interrupts that are set, stops the listening port, and frees all of its resources.

Not currently available on rp2040 (Pico) port, use nif functions.

Module http_server

- Function Index
- Function Details

Function Index

Function Details

parse_query_string/1

`parse_query_string(L) -> any()`

reply/3

`reply(StatusCode, ReplyBody, Conn) -> any()`

reply/4

`reply(StatusCode, ReplyBody, ReplyHeaders, Conn) -> any()`

start_server/2

`start_server(Port, Router) -> any()`

Module i2c

- Description
- Data Types
- Function Index
- Function Details

AtomVM I2c interface.

Description

This module provides and interface into the AtomVM I2C driver.

Use this module to communicate with devices connected to your ESP32 device via the 2-wire I2C interface.

Using this interface, you can read or write data to an I2C device at a given I2C address. In addition, you may read from or write to specific registers on the I2C device.

Data Types

address()

freq_hz()

i2c()

param()

params()

peripheral()

pin()

register()

Function Index

Function Details

begin_transmission/2

`I2C`: I2C instance created via `open/1Address`: I2C Address of the device (typically fixed for the device type)

returns: `ok` or `{error, Reason}`

Begin a transmission of I2C commands

This command is typically followed by one or more calls to `write_byte/2` and then a call to `end_transmission/1`

close/1

`I2C`: I2C instance created via `open/1` returns: `ok` atom

Closes the connection to the I2C driver

This function will close the connection to the I2C driver and free any resources in use by it.

end_transmission/1

`I2C`: I2C instance created via `open/1`

returns: `ok` or `{error, Reason}`

End a transmission of I2C commands

This command is typically preceded by a call to `begin_transmission/2` and one or more calls to `write_byte/2`.

open/1

`Params`: Initialization parameters

returns: process id of the driver.

Open a connection to the I2C driver

This function will open a connection to the I2C driver.

read_bytes/3

`I2C`: I2C instance created via `open/1Address`: I2C Address of the device (typically fixed for the device type)`Count`: The number of bytes to read

returns: `{ok, Data}` which includes the read binary data or `{error, Reason}`

Read a block of bytes from the I2C device.

This command is not wrapped in a `begin_transmission/2` and `end_transmission/1` call.

read_bytes/4

`I2C`: I2C instance created via `open/1Address`: I2C Address of the device (typically fixed for the device type)`Register`: The register address in the device from which to read data`Count`: The number of bytes to read

returns: `{ok, Data}` which includes the read binary data or `{error, Reason}`

Read a block of bytes from the I2C device starting at a specified register address

This command is not wrapped in a `begin_transmission/2` and `end_transmission/1` call.

write_byte/2

`I2C`: I2C instance created via `open/1Byte`: value to write

returns: `ok` or `{error, Reason}`

Write a byte to the device.

This command must be wrapped in a `begin_transmission/2` and `end_transmission/1` call.

write_bytes/2

`I2C`: I2C instance created via `open/1Bytes`: value to write

returns: `ok` or `{error, Reason}`

Write a sequence of bytes to the device.

This command must be wrapped in a `begin_transmission/2` and `end_transmission/1` call.

write_bytes/3

`I2C`: I2C instance created via `open/1Address`: I2C Address of the device (typically fixed for the device type)`BinOrInt`: The binary or byte value to write

returns: `ok` or `{error, Reason}`

Write a block of bytes to the I2C device.

This command is not wrapped in a `begin_transmission/2` and `end_transmission/1` call.

write_bytes/4

`I2C`: I2C instance created via `open/1Address`: I2C Address of the device (typically fixed for the device type)`Register`: The register address in the device to which to write data`BinOrInt`: The binary or byte value to write

returns: `ok` or `{error, Reason}`

Write a block of bytes to the I2C device starting at a specified register address.

This command is not wrapped in a `begin_transmission/2` and `end_transmission/1` call.

Module json_encoder

- Description
- Function Index
- Function Details

JSON specific APIs.

Description

This module contains functions for working with json data.

Function Index

Function Details

encode/1

`Data`: data to encode to json

returns: JSON encoded data

Convert data to json encoded binary

Module ledc

- Description
- Data Types
- Function Index
- Function Details

LED Controller low-level APIs.

Description

The functions in this module broadly reflect the ESP IDF-SDK LED Controller API.

See the IDF-SDK LEDC documentation for more information about these APIs.

Data Types

channel()

channel_cfg()

channel_config()

duty()

duty_cfg()

duty_resolution()

duty_resolution_cfg()

fade_mode()

freq_hz()

freq_hz_cfg()

gpio_num()

gpio_num_cfg()

hpoint()

hpoint_cfg()

ledc_error_code()

speed_mode()

Speed modes: use `0` for high speed, `1` for low speed.

speed_mode_cfg()

timer_config()

timer_num()

timer_num_cfg()

timer_sel()

timer_sel_cfg()

Function Index

Function Details

channel_config/1

`Config`: channel configuration

returns: ok | {error, ledc_error_code()}

LEDC channel configuration.

Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution.

fade_func_install/1

`Flags`: Flags used to allocate the interrupt. One (or multiple, using an ORred mask) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.

returns: ok | {error, ledc_error_code()}

Install LEDC fade function.

This function will occupy interrupt of LEDC module.

fade_func_uninstall/0

returns: ok

Uninstall LEDC fade function.

fade_start/3

`SpeedMode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.`Channel`: LEDC channel index (0-7).`FadeMode`: Whether to block until fading done.

returns: ok | {error, ledc_error_code()}

Start LEDC fading.

Note. Call ledc:fade_func_install() once before calling this function. Call ledc:fade_start() after this to start fading.

get_duty/2

`SpeedMode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.`Channel`: LEDC channel index (0-7).

returns: ok | {error, ledc_error_code()}

LEDC get duty.

get_freq/2

`SpeedMode`: Select the LEDC channel group with specified speed mode.`TimerNum`: LEDC timer index (0-3).

returns: ok | {error, ledc_error_code()}

LEDC get channel frequency (Hz)

set_duty/3

`SpeedMode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.`Channel`: LEDC channel index (0-7).`Duty`: Set the LEDC duty, the range of setting is [0, (2^duty_resolution)-1].

returns: ok | {error, ledc_error_code()}

LEDC set duty.

set_fade_with_step/5

`SpeedMode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.`Channel`: LEDC channel index (0-7).`TargetDuty`: Target duty of fading. (0..(2^duty_resolution)-1)`Scale`: Controls the increase or decrease step scale.`CycleNum`: increase or decrease the duty every cycle_num cycles

returns: ok | {error, ledc_error_code()}

Set LEDC fade function

Note. Call ledc:fade_func_install() once before calling this function. Call ledc:fade_start() after this to start fading.

set_fade_with_time/4

`SpeedMode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.`Channel`: LEDC channel index (0-7).`TargetDuty`: Target duty of fading. (0..(2^duty_resolution)-1)`MaxFadeTimeMs`: The maximum time of the fading (ms).

returns: ok | {error, ledc_error_code()}

Set LEDC fade function, with a limited time.

Note. Call ledc:fade_func_install() once before calling this function. Call ledc:fade_start() after this to start fading.

set_freq/3

`SpeedMode`: Select the LEDC channel group with specified speed mode.`TimerNum`: LEDC timer index (0-3).`FreqHz`: Set the LEDC frequency.

returns: ok | {error, ledc_error_code()}

LEDC set channel frequency (Hz)

stop/3

`SpeedMode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.`Channel`: LEDC channel index (0-7).`IdleLevel`: Set output idle level after LEDC stops.

returns: ok | {error, ledc_error_code()}

LEDC stop. Disable LEDC output, and set idle level.

timer_config/1

`Config`: timer configuration

returns: ok | {error, Reason}

LEDC timer configuration.

Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution.

update_duty/2

`SpeedMode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.`Channel`: LEDC channel index (0-7).

returns: ok | {error, ledc_error_code()}

LEDC update channel parameters.

Module mdns

- Data Types
- Function Index
- Function Details

Data Types

config()

Function Index

Function Details

parse_dns_message/1

```
parse_dns_message(Message) -> any()
```

parse_dns_name/2

```
parse_dns_name(Message, Data) -> any()
```

serialize_dns_message/1

```
serialize_dns_message(Dns_message) -> any()
```

serialize_dns_name/1

```
serialize_dns_name(Name) -> any()
```

start_link/1

Start mdns server and resolve `Hostname`.local

stop/1

Stop mdns responder

Module network

- Data Types
- Function Index
- Function Details

Data Types

ap_channel_cfg()

ap_config()

ap_config_property()

ap_max_connections_config()

ap_ssid_hidden_config()

ap_sta_connected_config()

ap_sta_disconnected_config()

ap_sta_ip_assigned_config()

ap_started_config()

db()

dhcp_hostname_config()

ghz24_channel()

This is the global 2.4 Ghz WiFI channel range, check your local jurisdiction for allowed channels in your geographic region.

ghz5_160mhz_channel()

This is the global 5 Ghz WiFI channel range when using 160Mhz bandwidth channels, check your local jurisdiction for allowed channels in your geographic region.

ghz5_20mhz_channel()

This is the global 5 Ghz WiFI channel range when using 20Mhz bandwidth channels, check your local jurisdiction for allowed channels in your geographic region.

ghz5_40mhz_channel()

This is the global 5 Ghz WiFI channel range when using 40Mhz bandwidth channels, check your local jurisdiction for allowed channels in your geographic region.

ghz5_80mhz_channel()

This is the global 5 Ghz WiFI channel range when using 80Mhz bandwidth channels, check your local jurisdiction for allowed channels in your geographic region.

ip_info()

ipv4_info()

mac()

mdns_config()

mdns_config_property()

mdns_hostname_config()

mdns_ttl_config()

network_config()

psk_config()

sntp_config()

sntp_config_property()

sntp_host_config()

sntp_synchronized_config()

ssid_config()

sta_beacon_timeout_config()

sta_config()

sta_config_property()

sta_connected_config()

sta_disconnected_config()

sta_got_ip_config()

wifi_channel()

Function Index

Function Details

handle_continue/2

```
handle_continue(X1, State) -> any()
```

sta_rssi/0

returns: {ok, Rssi} in decibels, or {error, Reason}.

Get the rssi information of AP to which the device is associated with.

start/1

`Config`: The network configuration

returns: ok, if the network interface was started, or {error, Reason} if a failure occurred (e.g., due to malformed network configuration).

Start a network interface.

This function will start a network interface, which will attempt to connect to an AP endpoint in the background. Specify callback functions to receive definitive information that the connection succeeded. See the AtomVM Network FSM Programming Manual for more information.

start_link/1

stop/0

returns: ok, if the network interface was stopped, or {error, Reason} if a failure occurred.

Stop a network interface.

wait_for_ap/0

Equivalent to wait_for_ap(15000).

wait_for_ap/1

`TimeoutOrApConfig`: The AP network configuration or timeout in ms.

Equivalent to wait_for_ap([], Timeout) or wait_for_ap(StaConfig, 15000).

wait_for_ap/2

`ApConfig`: The AP network configuration`Timeout`: amount of time in milliseconds to wait for a connection

returns: ok, when the network has started the AP, or {error, Reason} if a failure occurred (e.g., due to malformed network configuration).

Start a network interface in access point mode and wait the AP to be up and running

This function will start a network interface in AP mode, and will wait until the network is up and ready to be connected. This is a convenience function, for applications that do not need to be notified of connectivity changes in the network.

wait_for_sta/0

Equivalent to wait_for_sta(15000).

wait_for_sta/1

`TimeoutOrStaConfig`: The STA network configuration or timeout in ms.

Equivalent to wait_for_sta([], Timeout) or wait_for_sta(StaConfig, 15000).

wait_for_sta/2

`StaConfig`: The STA network configuration`Timeout`: amount of time in milliseconds to wait for a connection

returns: {ok, IpInfo}, if the network interface was started, or {error, Reason} if a failure occurred (e.g., due to malformed network configuration).

Start a network interface in station mode and wait for a connection to be established

This function will start a network interface in station mode, and will wait for a connection to be established. This is a convenience function, for applications that do not need to be notified of connectivity changes in the network.

Module network_fsm

- Description
- Function Index
- Function Details

network_fsm.

Description

This module is depreciated. Use the network module instead.
Function Index

Function Details

start/1

`start(Config) -> any()`

stop/0

`stop() -> any()`

wait_for_ap/0

`wait_for_ap() -> any()`

wait_for_ap/1

`wait_for_ap(ApConfig) -> any()`

wait_for_ap/2

`wait_for_ap(ApConfig, Timeout) -> any()`

wait_for_sta/0

`wait_for_sta() -> any()`

wait_for_sta/1

`wait_for_sta(StaConfig) -> any()`

wait_for_sta/2

`wait_for_sta(StaConfig, Timeout) -> any()`

Module pico

- Description
- Function Index
- Function Details

PICO-specific APIs.

Description

This module contains functions that are specific to the PICO platform.
Function Index

Function Details

cyw43_arch_gpio_get/1

`GPIO`: pin to read

returns: the level of the GPIO pin

Read a GPIO of the CYW43. This function is only available on Pico-W.

cyw43_arch_gpio_put/2

`GPIO`: pin to write`Level`: value to write

Write a GPIO of the CYW43. This function is only available on Pico-W. It is typically used to drive the on-board LED.

rtc_set_datetime/1

`Datetime`: `calendar:datetime()` to set rtc clock.

Set the datetime on the RTC. The datetime can be obtained through bif erlang:localtime()

Module port

- Description
- Function Index
- Function Details

AtomVM port driver APIs.

Description

This module contains functions that are intended to be used by drivers that rely on a `port` interface rather than `nifs`.

The port driver should be initialized with: `open_port({spawn, "Name"}, Param)` Where Name is an atom(), and is the name of the driver. The return from `open_port/2` will be the Port that will be required for future `port:call/2` or `port:call/3` use.

Examples:

```
    open_port({spawn, "i2c"}, Param)
```

or

```
    open_port({spawn, "spi"}, Params)
```

Function Index

Function Details

call/2

`Port`: Port to which to send messages`Message`: the message to send

returns: term() | {error, Reason}.

Send a message to a given port driver.

This function is used to send a message to an open port drivers port and will return a term or `{error, Reason}`.

call/3

`Port`: Port to which to send messages`Message`: the message to send`Timeout`: the timeout value in milliseconds

returns: term() | {error, Reason}.

Send a message to a given port driver with a timeout.

This function is used to send a message to an open port drivers port and will return a term or `{error, Reason}`, or `{error, timeout}` if the TimeoutMs is reached first.

Module spi

- Description
- Data Types
- Function Index
- Function Details

This module provides an interface into the Serial Peripheral Interface (SPI) supported on many devices.

Description

This module currently support the SPI "leader" (historically known as the "master") interface, allowing the leader to connect to one or more "follower" (historically known as "slave") devices.

Users interact with this interface by creating an instance of the driver via the `open/1` function, with returns an opaque reference to the driver instance. The `open/1` function takes a complex map structure, which configures the driver to connect to follower devices. See the `open/1` documentation for details about the structure of this configuration map.

Subsequent read and write operations use the SPI instance returned from the `open/1` function. Users may read from a specific follower device at a specific address, write to the device at an address, or simultaneously read from and write to the device in a single transaction.

Data Types

address()

bus_config()

device_config()

device_name()

params()

peripheral()

spi()

transaction()

Function Index

Function Details

close/1

`SPI`: SPI instance created via `open/1`

Close the SPI driver.

Close the SPI driver and free any resources in use by the driver.

The SPI instance will no longer be valid and usable after this function has been called.

open/1

`Params`: Initialization parameters

returns: process id of the driver.

throws `badarg`

Open a connection to the SPI driver

This function will open a connection to the SPI driver.

Supply a set of parameters to initialize the driver.

The parameters list must contain an SPI Bus configuration, together with a properties list containing one or more device configurations. This list must contain atom keys as names, which are used to identify the device in the subsequent read and write operations. You may use any atom value of your choosing.

The SPI Bus configuration is a properties list containing the following entries:

Each device configuration is a properties list containing the following entries:

Example:

```
Params = [
    {bus_config, [
        {miso, 16},
        {mosi, 17},
        {sclk, 5}
    },
    {device_config, [
        {device1, [
            {cs, 18}
        ]},
        {device2, [
            {cs, 19}
        ]}
    ]}
]
```

Note that `device1` and `device2` are atom names used to identify the device for read and write operations.

This function raises an Erlang exception with a `badarg` reason, if initialization of the SPI Bus or any device fails.

The `write/3` and `write_read/3` functions in this module are designed to provide the maximum mount of flexibility when interfacing with the SPI device. The both make use of a map structure to encapsulate an SPI transaction.

An SPI transaction may contain a command, and address, and/or a blob of data, each of which is optional and each of which depends on how users interact with the device. Consult the data sheet for your SPI device to understand which fields should be used with your device.

The fields of a transaction map are as follows:

read_at/4

`SPI`: SPI instance created via `open/1DeviceName`: device name from configuration`Address`: SPI Address from which to read`Len`: in bytes to read

returns: `{ok, Value}` or `error`

Read a value from and address on the device.

write/3

`SPI`: SPI instance created via `open/1DeviceName`: SPI device name (use key in `device_config`)`Transaction`: transaction map.

returns: `ok` or `{error, Reason}`, if an error occurred.

Write data to the SPI device, using the instructions encoded in the supplied transaction.

The supplied `Transaction` encodes information about how data is to be written to the selected SPI device. See the description above for the fields that may be specified in this map.

When a binary is supplied in the `write_data` field, the data is written to the SPI device in the natural order of the binary. For example, if the input binary is `<<16#57, 16#BA>>`, then the first byte is `0x57` and the second byte is `0xBA`.

The value of the `write_bits` field, if specified, must be less than or equal to `8 * byte_size(write_data)`. If `write_bits` is less than `8 * byte_size(write_data)`, only the first `write_bits` bits from `write_data` will be written.

This function will return a tuple containing the `error` atom if an error occurred writing to the SPI device at the specified address. The returned reason term is implementation-defined.

write_at/5

`SPI`: SPI instance created via `open/1DeviceName`: device name from configuration`Address`: SPI Address to which to write`Len`: in bytes to read`Data`: byte(s) to write

returns: `{ok, Value}` or `error`

Write a value to and address on the device.

The value returned from this function is dependent on the device and address. Consult the documentation for the device to understand expected return values from this function.

write_read/3

`SPI`: SPI instance created via `open/1DeviceName`: SPI device name (use key in `device_config)Transaction`: transaction.

returns: `{ok, binary()}` or `{error, Reason}`, if an error occurred.

Write data to the SPI device, using the instructions encoded in the supplied transaction. device, and simultaneously read data back from the device, returning the read data in a binary.

The supplied `Transaction` encodes information about how data is to be written to the selected SPI device. See the description above for the fields that may be specified in this map.

When a binary is supplied in the `write_data` field, the data is written to the SPI device in the natural order of the binary. For example, if the input binary is `<<16#57, 16#BA>>`, then the first byte is `0x57` and the second byte is `0xBA`.

The value of the `write_bits` field, if specified, must be less than or equal to `8 * byte_size(write_data)`. If `write_bits` is less than `8 * byte_size(write_data)`, only the first `write_bits` bits from `write_data` will be written.

The return value contains a sequence of bytes that have been read from the SPI device. The number of bytes returned will be `ceil(read_bits / 8)`. Only the first `read_bits` will be populated.

This function will return a tuple containing the `error` atom if an error occurred writing to the SPI device at the specified address. The returned reason term is implementation-defined.

Module timestamp_util

- Description
- Data Types
- Function Index
- Function Details

Utility functions for comparing timestamps.

Description

This module contains functions that are useful for comparing timestamps without running the risk of integer overflow.

Note that the functions in this module may be obsoleted in future versions of AtomVM, as support for arbitrary sized integers is added; however, the functions may still be useful in their own right.

Data Types

megasecs()

microsecs()

secs()

timestamp()

Function Index

Function Details

delta/2

`TS2`: a timestamp`TS1`: a timestamp

returns: TS2 - TS1, as a timestamp

Computes the difference between TS2 and TS1, as a timestamp.

delta_ms/2

`TS2`: a timestamp`TS1`: a timestamp

returns: TS2 - TS1, in milliseconds

Computes the difference between TS2 and TS1, in milliseconds.

Module uart

- Data Types
- Function Index
- Function Details

Data Types

peripheral()

The peripheral `Name` may be one of: `"UART0"` | `"UART1"` | `"UART2"` | `<<"UART0">>` | `<<"UART1">>` | `<<"UART2">>`.

uart_opts()

Function Index

Function Details

close/1

`Port`: of the uart port to be closed

returns: ok.

Close a port connection to the UART driver

This function will close the given port connection to the UART driver.

open/1

`Opts`: uart configuration options

returns: Port of the driver.

Open a connection to the UART driver default port

This function will open a connection to the UART driver.

open/2

`Name`: the uart peripheral to be opened`Opts`: uart configuration options

returns: Port of the driver.

Open a connection to the UART driver

This function will open a connection to the UART driver.

read/1

`Port`: of the uart port to be read

returns: {ok, Data} or {error, Reason}

Read data from a UART port

This function will return any data that is available, or return a `{error, timeout}` tuple. The driver will sent the next available data from the UART driver to the process that made the last read. Example:

```
Data = case uart:read(Uart) of
    {ok, Binary} -> Binary;
    {error, timeout} ->
        receive
            {ok, RecvBinary} -> RecvBinary;
            Error -> error(Error)
        end;
    Error -> error(Error)
end,
```

Any attempt by another (or the same process) to read from uart before the next uart payload is sent by the driver will result in `{error, ealready}`.

read/2

`Port`: of the uart port to be read`Timeout`: millisecond to wait for data to become available

returns: `{ok, Data}`, or `{error, Reason}`

Read data from a UART port

This function will return any data that is available within the timeout period to the process. After the timeout has expired a new read command may be used regardless of whether the last read was sent a payload. Example:

```
            Data = case uart:read(Uart, 3000) of
                {ok, Bin} -> Bin;
                {error, timeout} -> <<"">>;
                Error -> error_handler_fun(Uart, Error)
            end,
```

Any data sent to the esp32 over uart between reads with a timeout will be lost, so be sure this is what you want. Most applications will want a single process to read from UART and continue to listen until a payload is received, and likely pass the payload off for processing and immediately begin another read.

write/2

`Port`: of the uart port to be written to`Data`: to be written to the given uart port

returns: ok or {error, Reason}

Write data to a UART port

This function will write the given data to the UART port.

Module websocket

- Description
- Data Types
- Function Index
- Function Details

websocket API.

Description

The functions in this module are currently only implemented for the emscripten platform.

Data Types

ready_state()

websocket()

abstract datatype: `websocket()`

Function Index

Function Details

buffered_amount/1

`Websocket`: websocket to get the buffered amount of

returns: a number of bytes

Returns the number of bytes of application data that have been queued using `send/2` but not yet been transmitted to the network.

close/1

`Websocket`: websocket to close

returns: `ok`

Equivalent to `close(Websocket, CLOSE_STATUS_CODE_NORMAL, [])`.

Close websocket

close/2

`Websocket`: websocket to close`StatusCode`: status code to send to server

returns: `ok`

Equivalent to `close(Websocket, StatusCode, [])`.

Close websocket

close/3

`Websocket`: websocket to close`StatusCode`: status code to send to server`Reason`: reason to send to server

returns: `ok`

Close websocket

controlling_process/2

`Websocket`: websocket to set the controlling process for`ControllingProcess`: Process that gets messages from the websocket

returns: `ok` or an error tuple

Set the controlling process of the websocket. Can only be called from the current owner of the websocket.

extensions/1

`Websocket`: websocket to get the extensions of

returns: the websocket extensions

Returns the extensions selected by the server, if any.

is_supported/0

returns: `true` if websockets are supported

Determine if websockets are supported in the current environment (browser).

new/1

Equivalent to `new(URL, <<"binary,base64">>, self())`.

Open a new websocket with the given server.

new/2

Equivalent to `new(URL, Protocols, self())`.

Open a new websocket with the given server.

new/3

`URL`: URL to connect to`Protocols`: List of protocol strings for negociation with the server.`ControllingProcess`: Process that gets messages from the websocket

returns: a websocket resource

Open a new websocket with the given server.

protocol/1

`Websocket`: websocket to get the protocol of returns: the websocket protocol

Returns the subprotocol selected by the server, if any.

ready_state/1

`Websocket`: websocket to get the ready state of

returns: the ready state of the socket

Returns the state of the websocket connection.

send_binary/2

`Websocket`: websocket to send data to`Data`: data to send

returns: `ok` or an error tuple.

Send data to the websocket.

send_utf8/2

`String`: string to send

returns: `ok` or an error tuple.

Send data to the websocket.

url/1

`Websocket`: websocket to get the URL of

returns: the websocket URL

Returns the URL that was used to open the websocket

12.1.3 alisp

The alisp library

Modules

Module alisp

- Function Index
- Function Details

Function Index

Function Details

booleanize/1

`booleanize(V) -> any()`

eval/1

`eval(S) -> any()`

run/1

`run(S) -> any()`

Module alisp_stdlib

- Function Index
- Function Details

Function Index

Function Details

''/1*

`*(L) -> any()`

'+'/1

`+(L) -> any()`

'-'/1

`-(T) -> any()`

'='/1

`=(T) -> any()`

'remove-if'/1

`remove-if(X1) -> any()`

'remove-if-not'/1

`remove-if-not(X1) -> any()`

append/1

`append(X1) -> any()`

binaryp/1

`binaryp(X1) -> any()`

car/1

`car(X1) -> any()`

cdr/1

`cdr(X1) -> any()`

cons/1

`cons(X1) -> any()`

floatp/1

`floatp(X1) -> any()`

identity/1

`identity(X1) -> any()`

integerp/1

`integerp(X1) -> any()`

last/1

`last(X1) -> any()`

list/1

`list(List) -> any()`

listp/1

`listp(X1) -> any()`

mapcar/1

`mapcar(X1) -> any()`

numberp/1

`numberp(X1) -> any()`

pidp/1

`pidp(X1) -> any()`

print/1

`print(X1) -> any()`

refp/1

`refp(X1) -> any()`

tuple/1

`tuple(List) -> any()`

tuplep/1

`tuplep(X1) -> any()`

Module arepl

- Function Index
- Function Details

Function Index

Function Details

start/0

`start() -> any()`

Module sexp_lexer

- Function Index
- Function Details

Function Index

Function Details

string/1

`string(Bin) -> any()`

Module sexp_parser

- Function Index
- Function Details

Function Index

Function Details

parse/1

`parse(T) -> any()`

Module sexp_serializer

- Function Index
- Function Details

Function Index

Function Details

serialize/1

`serialize(Value) -> any()`

12.1.4 etest

The etest library

Modules

Module etest

- Description
- Function Index
- Function Details

This modules provides a basic testing framework for AtomVM Erlang libraries.

Function Index

Function Details

assert_equals/2

`X`: a term`Y`: a term

returns: ok if X and Y are equal; fail otherwise.

assert_exception/1

`F`: a function to evaluate

returns: ok if evaluating F results in Error being raised; fail, otherwise

assert_exception/2

`F`: a function to evaluate`Class`: expected exception class

returns: ok if evaluating F results in an exception of class Class being raised; fail, otherwise

assert_exception/3

`F`: a function to evaluate`Class`: expected exception class`E`: expected exception value

returns: ok if evaluating F results in an exception of class Class and of value E being raised; fail, otherwise

assert_match/2

`X`: a term`Y`: a term

returns: ok if X and Y unify; fail otherwise.

assert_true/1

`X`: a term

returns: ok if X is true; fail otherwise.

flush_msg_queue/0

returns: ok after flushing all messages in the process message queue

Use optionally to flush messages in test cases that run in a single test module

test/1

`Tests`: a list of test modules

returns: ok if all of the tests pass, or the atom fail, if any of the tests failed.

Test a sequence of test modules.

This function will execute the test/0 function for each module provided in the input list of test modules. If all of the tests return the atom ok, then this function returns ok. If any of the test modules return a value other than ok, then this function returns the atom fail.

Module eunit

- Description
- Data Types
- Function Index
- Function Details

This modules provides an eunit-compatible API for AtomVM generating an output that can be parsed by dut.expect_unity_test_output.

Data Types

option()

test()

Function Index

Function Details

start/0

```
start() -> any()
```

test/1

Equivalent to `test(Tests, [])`.

test/2

`Tests`: tests or module to run tests from`Options`: options. If `exact_execution` is true and Tests is a module name, tests from the module with _tests suffix are not collected.

returns: ok if all of the tests pass, or the atom fail, if any of the tests failed.

Run tests print result using io:format in a format compatible with `dut.expect_unity_test_output`.

12.2 AtomVM 'C' APIs

Chapter 13

Contributing

AtomVM is open to any contribution.

Pull requests, bug reports and feature requests are welcome.

However before contributing, please read carefully our Code of Conduct and the following contribution guidelines.

Please, also make sure to understand the Apache 2.0 license and the Developer Certificate of Origin.

Last but not least, **do not use GitHub issues for vulnerability reports**, read instead the security policy for instructions.

13.1 Git Recommended Practices

- Commit messages should have a
- summary and a description
- Remove any trailing white spaces
- Always `git pull --rebase`
- Clean up your branch history with `git rebase -i`
- Squash commits before PR, unless there is a good reason not to
- All your intermediate commits should build

13.2 Coding Style

For all source code modules:

- Remove all trailing whitespace
- Newlines (`\n`) at end of file
- Use line ending conventions appropriate for the platform (e.g., `\n` on UNIX-like systems)

13.2.1 Copyright Headers

All source code modules should include copyright headers that are formatted for the relevant module language. Copyright headers should take the following form:

```
/*
* This file is part of AtomVM.
*
```

```
* Copyright 2020 Your name <your@email.address>
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*    http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*
* SPDX-License-Identifier: Apache-2.0 OR LGPL-2.1-or-later
*/
```

13.2.2 C Code

C source code style is enforced with `clang-format-16`. To automatically fix a file, run:

```
clang-format-16 --style=file -i file.c
```

Indentation

- K&R indentation and braces style
- Mandatory braces
- 4 space indentation (no tabs)

Good:

```
void f(bool reverse)
{
    if (reverse) {
        puts("!dlroW olleH");
    } else {
        puts("Hello world");
    }
}
```

Bad:

```
void f(bool reverse) {
    if (reverse)
        puts ("!dlroW olleH");
    else
        puts ("Hello world");
}
```

Naming Conventions

- Struct names are PascalCase (e.g. `Context`)
- Scalar types are lower case (e.g. `term`)
- All other names (e.g. functions and variables) are snake_case (e.g. `term_is_integer`)
- Always prefix exported function names with the module in which they are defined (e.g. `term_is_nil`, `term_is_integer`, `context_new`, `context_destroy`)

Other Coding Conventions

- Pointers (`*`) should be with the variable name rather than with the type (e.g. `char *name`, not `char* name`)
- Avoid long lines, use intermediate variables with meaningful names.
- Function definitions should be separated by 1 empty line

Function declarations should be structured as follows:

```
func(main_input, additional_inputs, main_output, additional_outputs, opts,
 [context])
```

where `context` is a context structure (such as `Context` or `GlobalContext`).

Any functions that are not exported should be qualified with the `static` keyword.

Functions that return booleans should be named `is_something` (or possibly `module_is_something`, if the function is exported).

C header modules (`.h`) should be organized as follows:

```
+-------------------
| Copyright Header
|
| #ifdef MODULE__H__
| #define MODULE__H__
|
| #ifdef __cplusplus
| extern "C" {
| #endif
|
| #includes (alphabetical)
|
| #defines
|
| type definitions
|
| function declarations
|
| #ifdef __cplusplus
| }
| #endif
|
| #endif
+-------------------
 module.h
```

C source modules (`.c`) should be organized as follows:

```
+-------------------
| Copyright Header
|
| #includes (alphabetical)
|
| #defines
|
| type definitions
|
| forward declarations (only if necessary)
|
| function definitions
|   dependent static functions first
|   exported functions and entrypoints last
```

```
+-------------------
 module.c
```

Documentation

Doxygen Javadoc style code comments will be picked up and added to the documentation. Changes will automatically be added to the libAtomVM Source Files and the libAtomVM Index. But to have `Data Structures`, `Types`, `MACROS`, and `Functions` appear in the correct C Library APIs section the corresponding entries must be added to the similarly named `*.rst` files in the `AtomVM/doc/src/apidocs/libatomvm/` directory. The exact names of the files that need to be altered are: `data_structures.rst`, `functions.rst`, `macros.rst`, and `types.rst`. The other files in the directory handle auto`generated content and do not need to be altered.

In the rare case that a function declaration and definition are both in different header files (rather than the definition in a `*.c` file) this can cause rendering errors for `Doxygen`. The work around for these cases can be demonstrated with this example for the function `sys_listener_destroy` it is documented and declared in `sys.h` and defined as follows in `listeners.h`:

```
#ifndef DOXYGEN_SKIP_SECTION /* documented in sys.h */
void sys_listener_destroy(struct ListHead *item)
{
    EventListener *listener = GET_LIST_ENTRY(item, EventListener,
 listeners_list_head);
    free(listener);
}
#endif /* DOXYGEN_SKIP_SECTION */
```

> Note: You should include a short `/* comment */` trailing the `#ifndef` entry mentioning the file where the function is actually documented.

13.2.3 Erlang Code

Erlang source code style is enforced using erlfmt.

13.2.4 Elixir Code

Just use Elixir formatter enforced style.

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

14.1 Unreleased

14.1.1 Added

- Added a limited implementation of the OTP `ets` interface
- Added `code:all_loaded/0` and `code:all_available/0`
- Added `erlang:split_binary/2`
- Added `inet:getaddr/2`
- Added support for external pids and encoded pids in external terms
- Added support for external refs and encoded refs in external terms
- Introduce ports to represent native processes and added support for external ports and encoded ports in external terms
- Added `atomvm:get_creation/0`, equivalent to `erts_internal:get_creation/0`
- Added menuconfig option for enabling USE_USB_SERIAL, eg. serial over USB for certain ESP32-S2 boards etc.
- Partial support for `erlang:fun_info/2` and `erlang:fun_info/1`
- Added support for `registered_name` in `erlang:process_info/2` and `Process.info/2`
- Added `net:gethostname/0` on platforms with gethostname(3).
- Added `socket:getopt/2`
- Added `supervisor:terminate_child/2`, `supervisor:restart_child/2` and `supervisor:delete_child/2`
- Added support for 'erlang:--/2'.
- Added `esp:partition_read/3`, and documentation for `esp:partition_erase_range/2/3` and `esp:partition_write/3`
- Added support for list insertion in 'ets:insert/2'.
- Support to OTP-28
- Added `atomvm:subprocess/4` to perform pipe/fork/execve on POSIX platforms
- Added `externalterm_to_term_with_roots` to efficiently preserve roots when allocating memory for external terms.

- Added `erl_epmd` client implementation to epmd using `socket` module
- Added support for socket asynchronous API for `recv`, `recvfrom` and `accept`.
- Added support for UDP multicast with socket API.
- Added support for `ets:update_counter/3` and `ets:update_counter/4`.
- Added `erlang:+/1`
- Added `lists:append/1` and `lists:append/2`
- Added `erlang:spawn_monitor/1`, `erlang:spawn_monitor/3`
- Added `lists:dropwhile/2`.
- Support for `float/1` BIF.
- Added `erlang:get/0` and `erlang:erase/0`.
- Added `erlang:unique_integer/0` and `erlang:unique_integer/1`
- Added support for 'ets:delete/1'.
- Added `lists:flatmap/2`
- Added `io:fwrite/1,2,3` and `io:format/3` as well as few io functions required by remote shell
- Added `code:is_loaded/1` and `code:which/1`
- Added several `io_lib` functions including `io_lib:fwrite/2` and `io_lib:write_atom/1`
- Added `init:get_argument/1`, `init:get_plain_arguments/0` and `init:notify_when_started/1`
- Added `application:get_env/2`
- Added CodeQL analysis to esp32, stm32, pico, and wasm workflows
- Added Function.ex and Protocol.ex improving Elixir 1.18 support
- Added WiFi support for ESP32P4 via esp-wifi-external for build with ESP-IDF v5.4 and later
- Added Process.link/1 and unlink/1 to Elixir Process.ex

14.1.2 Changed

- Removed `externalterm_to_term_copy` added in [0.6.5] and introduced flags to `externalterm_to_term` to perform copy.
- Release images for ESP32 chips are built with ESP-IDF v5.4
- ESP32: SPI peripheral defaults to `"spi2"` instead of deprecated `hspi`

14.1.3 Fixed

- ESP32: improved sntp sync speed from a cold boot.
- Fixed `gen_server` internal messages to match OTP so it works across erlang distribution
- Utilize reserved `phy_init` partition on ESP32 to store wifi calibration for faster connections.
- Support for zero count in `lists:duplicate/2`.
- packbeam: fix memory leak preventing building with address sanitizer

14.2 [0.6.6] - Unreleased

14.2.1 Added

- Added the ability to run beams from the CLI for Generic Unix platform (it was already possible with nodejs and emscripten)
- Added preliminary support for ESP32P4 (no networking support yet).
- Added memory info in `out_of_memory` crash logs to help developers fix memory issues.
- Added documentation and function specs for uart driver
- Added `uart:read/2` with a timeout parameter.
- Missing `erlang:is_function/2` BIF
- Added `erlang:is_record/2`
- Added ability to set per-interface `dhcp_hostname` on Pico W if present in config.

14.2.2 Fixed

- Fixed specifications of nifs from `esp_adc` module
- ESP32: fix `gpio:init/1` on GPIO >= 32
- Adding missing check, passing a non numeric argument to a function expecting a floating point might lead to a crash in certain situations.
- Fixed several bugs in `http_server` (#1366)
- Fixed generic_unix `socket_driver` to return `{gen_tcp, closed}` when socket is closed on Linux instead of `{gen_tcp, {recv, 104}}`
- Fixed a memory leak where modules were not properly destroyed when the global context is destroyd
- alisp: fix support to variables that are not binaries or integers.
- Fixed destruction of ssl-related resources
- Fixed corruption when dealing with specific situations that involve more than 16 x registers when certain VM instructions are used.
- Fixed ESP32 GPIO interrupt trigger `none`
- Fixed an issue where a timeout would occur immediately in a race condition
- Fixed SPI close command
- Added missing lock on socket structure
- Fixed a race condition affecting multi-core MCUs where a timeout would not be properly cleared
- Fixed a double free when esp32 uart driver was closed, yielding an assert abort
- Fixed compilation with latest debian gcc-arm-none-eabi
- Fixed `network:stop/0` on ESP32 so the network can be started again
- Fixed a memory corruption caused by `binary:split/2,3`
- Fixed deadlock in socket code
- Fixed bug in opcode implementation (`select_val`): when selecting a value among many others a shallow comparison was performed, so it was working just for plain values such as atoms and small integers
- Fixed support for setting esp32 boot_path in NVS.

- Fixed race conditions in network:start/stop.
- Fixed crash calling network:sta_rssi(), when network not up.
- Fixed error handling when calling `min` and `max` with code compiled before OTP-26: there was a bug when handling errors from BIFs used as NIFs (when called with `CALL_EXT` and similar opcodes)
- Fixed matching of binaries on unaligned boundaries for code compiled with older versions of OTP
- Added missing out of memory handling in binary_to_atom
- Fixed call to funs such as fun erlang:'not'/1, that make use of BIFs
- Fixed potential crashes or memory leaks caused by a mistake in calculation of reference counts and a race condition in otp_socket code
- Fixed an out of memory issue by forcing GC to copy data from message fragments
- Fixed a bug where calling repeatedly `process_info` on a stopped process could cause an out of memory error
- Fixed possible concurrency problems in ESP32 UART driver
- Fixed concurrency and memory leak related to links and monitors
- Fixed issues with parsing of line references for stack traces
- Fixed memory corruption issue with `erlang:make_tuple/2`
- Fix potential use after free with code generated from OTP <= 24
- Fix `is_function/2` guard
- Fixed segfault when calling `lists:reverse/1` (#1600)
- Fixed nif_atomvm_posix_read GC bug
- Fixed `erlang:is_number/1` function, now returns true also for floats
- Fixed unlink protocol and add support for `link/1` on ports
- Do not abort when an out of memory happens while loading a literal value
- Fixed potential memory corruption when handling integer immediates that are stored as boxed integer (this never happens with integers < 28 bits)
- Correctly set Pico-W unique dhcp hostname when using the default, previously all rp2040 devices used the same "PicoW" dhcp hostname, causing collisions when multiple rp2040 are on the same network. (See issue #1094)

14.2.3 Changed

- ESP32 UART driver no longer aborts because of badargs in configuration, instead raising an error
- ESP32: `v0.6.6` uses esp-idf v5.4.1 for pre-built images and `v5.4.x` is the suggested release also for custom builds

14.3 [0.6.5] - 2024-10-15

14.3.1 Added

- ESP32: add a new Elixir release "flavor" with a bigger boot.avm partition that has room for Elixir standard library modules
- ESP32: `--boot` option to mkimage.sh tool

- Add `erlang:atom_to_binary/1` that is equivalent to `erlang:atom_to_binary(Atom, utf8)`
- Support for Elixir `String.Chars` protocol, now functions such as `Enum.join` are able to take also non string parameters (e.g. `Enum.join([1, 2], ",")`
- Support for Elixir `Enum.at/3`
- Add support for `is_bitstring/1` construct which is used in Elixir protocols runtime.
- Add support to Elixir `Enumerable` protocol also for `Enum.all?`, `Enum.any?`, `Enum.each`, `Enum.filter`, `Enum.flat_map`, `Enum.reject`, `Enum.chunk_by` and `Enum.chunk_while`
- Support for `maps:merge_with/3`
- Support for `lists:last/1` and `lists:mapfoldl/3`
- Add support to Elixir for `Process.send/2` `Process.send_after/3/4` and `Process.cancel_timer/1`
- Add support for `handle_continue` callback in `gen_server`
- Support for Elixir `List.Chars` protocol
- Support for `gen_server:start_monitor/3,4`
- Support for `code:ensure_loaded/1`
- Support for `io_lib:latin1_char_list/1`
- Add support to Elixir for `Keyword.split/2`
- Support for `binary:split/3` and `string:find/2,3`
- Support for large tuples (more than 255 elements) in external terms.
- Support for `io:put_chars/2`
- Support for `lists:nthtail/2`
- Support for Elixir `IO.chardata_to_string/1`
- Support for Elixir `List.duplicate/2`
- Support for `binary:copy/1,2`
- Support for directory listing using POSIX APIs: (`atomvm:posix_opendir/1`, `atomvm:posix_readdir/1`, `atomvm:posix_closedir/1`).
- ESP32: add support for `esp_adc` ADC driver, with Erlang and Elixir examples
- Add handler for ESP32 network driver STA mode `beacon_timeout` (event: 21), see issue #1100
- Support for mounting/unmounting storage on ESP32 (such as SD or internal flash) using `esp:mount/4` and `esp:umount/1`
- Support for `binary_to_integer/2`
- Support for `binary:decode_hex/1` and `binary:encode_hex/1,2`
- Support for Elixir `Base.decode16/2` and `Base.encode16/2`
- Make external term serialize functions available without using `externalterm_to_binary` so terms can be written directly to a buffer.
- Support for `erlang:list_to_integer/2`
- Add `externalterm_to_term_copy` that can be safely used from NIFs taking temporary buffers

14.3.2 Changed

- ESP32: Elixir library is not shipped anymore with `esp32boot.avm`. Use `elixir_esp32boot.avm` instead

- `Enum.find_index` and `Enum.find_value` support Enumerable and not just lists
- Install AtomVM libraries source code and binaries for better dialyzer integration
- Made the `device_config` properties list in `spi:open/1` optional (defaults to `[]`), so you can use the function with only a `bus_config`

14.3.3 Fixed

- ESP32: content of `boot.avm` partition is not truncated anymore
- ESP32: `Fixed gpio:set_int` to accept any pin, not only pin 2
- Fix memory corruption in `unicode:characters_to_binary`
- Fix handling of large literal indexes and large extended literal indexes
- `unicode:characters_to_list`: fixed bogus out_of_memory error on some platforms such as ESP32
- Fix crash in Elixir library when doing `inspect(:atom)`
- General inspect() compliance with Elixir behavior (but there are still some minor differences)
- Fix several uses of free on prevously released memory on ESP32, under certain error condition using `network:start/1`, that would lead to a hard crash of the VM.
- Fix a bug in ESP32 network driver where the low level driver was not being stopped and resoureces were not freed when `network:stop/0` was used, see issue #643
- `uart:open/1,2` now works with uppercase peripheral names

14.4 [0.6.4] - 2024-08-18

14.4.1 Added

- Implement `gpio:init/1` on esp32 to initialize pins for GPIO usage, which some pins require depending on default function and bootloader code
- Implement missing opcode 161 (raw_raise), that looks more likely to be generated with Elixir code
- Support for Elixir `Map.replace/3` and `Map.replace!/3`
- Support for Elixir `Kernel.struct` and `Kernel.struct!`
- Support for Elixir `IO.iodata_to_binary/1`
- Support for Elixir exceptions: `Exception` module and the other error related modules such as `ArgumentError`, `UndefinedFunctionError`, etc...
- Support for Elixir `Enumerable` and `Collectable` protocol
- Support for Elixir `Enum` functions: `split_with`, `join`, `map_join`, `into`, `reverse`, `slice` and `to_list`
- Support for Elixir `MapSet` module
- Support for Elixir `Range` module
- Support for Elixir `Kernel.min` and `Kernel.max`
- Support (as stub) for `erlang:error/3` (that is required from Elixir code)

14.5 [0.6.3] - 2024-07-20

14.5.1 Added

- Simple http client, that can be used for different use case such as downloading OTA updates
- Elixir support for `Keyword.merge` `Keyword.take` `Keyword.pop(!)` `Keyword.keyword?` `Keyword.has_key?` functions.
- Support for ESP32-H2
- lists:keytake/3 implemented.
- Support for setting channel used by network driver wifi access point.
- Support for `maps:iterator/2` and `~kp` with `io_lib:format/2` that were introduced with OTP26.
- Support for `erlang:apply/2`
- Support for `lists:keystore/4`
- Support for `erlang:size/1` bif
- Support for USB serial output on ESP32 (needs to be manually enabled)
- Support for `lists:filtermap/2`
- Support for standard library `queue` module
- Support for `maps:from_keys/2` NIF
- Support for standard library `sets` module

14.5.2 Changed

- ESP32 network driver messages for event 40 (home channel change events) are now suppressed, but the details for the channel changes can be observed in the console log if "debug" level logging is enabled in ESP-IDF Kconfig options.
- Default size of ESP32 RTC slow memory from 4086 to 4096, except on ESP32-H2 where it's 3072
- Update `byte_size/1` and `bit_size/1` to implement OTP27 match context reuse optimization OTP-18987.

14.5.3 Fixed

- Fix bug (with code compiled with OTP-21) with binary pattern matching: the fix introduced with `02411048` was not completely right, and it was converting match context to bogus binaries.
- Fix creation of multiple links for the same process and not removing link at trapped exits. See issue #1193.
- Fix error that is raised when a function is undefined
- Fix a bug that could yield crashes when functions are sent in messages
- Fix bug where failing guards would corrupt x0 and x1
- Fix a memory leak when raising out of memory error while executing PUT_MAP_ASSOC instruction

14.6 [0.6.2] - 25-05-2024

14.6.1 Added

- Support for DragonFly BSD (generic_unix platform).
- Added guards `is_even` and `is_odd` to the `Integer` module
- Add a number of functions to proplists module, such as `delete/2`, `from/to_map/1`, etc…
- Add `esp:deep_sleep_enable_gpio_wakeup/2` to allow wakeup from deep sleep for ESP32C3 and ESP32C6.
- Obtain RSSI of the current connection with `network:sta_rssi/0` on ESP32.
- Pico-W support for `network:sta_rssi/0`.
- Add support to ESP32C2

14.6.2 Fixed

- Fix invalid read after free in ssl code, see also issue #1115.
- Fix semantic of `ssl:recv(Socket, 0)` to return all available bytes, matching what OTP does.
- Fix `binary` option handling in `ssl:connect/3` so `binary` can be used instead of `{binary, true}`.
- Fix scheduling of trapped process that were wrongly immediately rescheduled before being signaled.
- Fix `gen_tcp` and `ssl` types.
- Fix documentation and specification of `esp:sleep_enable_ext0_wakeup/2` and `esp:sleep_enable_ext1_wakeup/2`.

14.6.3 Changed

- Stacktraces are included by default on Pico devices.
- Changed ssl default from `{active, false}` to `{active, true}` in order to have same behavior as OTP. Since active mode is not supported right now, `active` must be explicitly set to false: `ssl:connect(..., ..., [{active, false}, ...])`, otherwise it will crash.

14.7 [0.6.1] - 2024-04-17

14.7.1 Added

- Added experimental optimized GC mode that makes use of C realloc instead of copying data around, it can be enabled with `-DENABLE_REALLOC_GC=On`.

14.7.2 Fixed

- Fix bug in `erlang:ref_to_list/1` and `erlang:display/1`: the unique integer was truncated on some 32-bit architectures
- Stop hardcoding `erl_eval` as module name in both display and fun_to_list

- Correctly display and convert to list funs such as `fun m:f/a`
- Fixed bug in STM32 cmake that could cause builds with multiple jobs to fail due to incorrect artifact dependency
- Fix crash on macOS due to missing call to `psa_crypto_init` for TLS 1.3
- Fix crypto test on rp2040

14.8 [0.6.0] - 2024-03-05

14.8.1 Fixed

- Fix a bug that broke sockets on ESP32-C3 and other single core ESP32 devices, that may also cause other issues. The bug has been introduced with messages from tasks change between beta.1 and rc.0
- Fixed several issues related to Esp32 socket_driver that made it unreliable, especially with single core MCUs

14.9 [0.6.0-rc.0] - 2024-03-03

14.9.1 Added

- `BOOTLOADER_OFFSET` for all current Esp32 models.
- Added API to send messages from FreeRTOS tasks or pthreads, typically to easily support integration with Esp32 callbacks

14.9.2 Fixed

- `BOOTLOADER_OFFSET` was incorrect for Esp32-C6 and Esp32-S2.
- Fixed a bug that would fail to set DHCP hostname in STA+AP mode on all ESP32 platforms.
- ESP32-S3: crash in network driver caused by a smaller stack size for scheduler threads, when calling `esp_wifi_init()`. See also issue #1059.
- Fixed Esp32 network driver on non-SMP builds
- ESP32: fixed bug in `gpio:stop/0` and `gpio:close/1` that would cause the VM to crash.

14.10 [0.6.0-beta.1] - 2024-02-28

14.10.1 Added

- Support for utf8 encoding to `*_to_atom` and `atom_to_*` functions
- `binary_to_atom/1` and `atom_to_binary/1` that default to utf8 (they were introduced with OTP23)
- Added Pico cmake option `AVM_WAIT_BOOTSEL_ON_EXIT` (default `ON`) to allow tools to use automated `BOOTSEL` mode after main application exits
- Use UTF-8 encoding for atoms when using `erlang:term_to_binary/1`, in conformance with OTP-26

- Pico: Wait for USB serial connection `cmake` configuration option `AVM_USB_WAIT_SECONDS` added with 20 second default.
- Support for code that makes use of more than 16 live registers, such as functions with > 16 parameters and complex pattern matchings.

14.10.2 Fixed

- ESP32: fix i2c_driver_acquire and i2c_driver_release functions, that were working only once.
- Sending messages to registered processes using the `!` operator now works.
- Fixed bug in `OP_SEND` that would accept sending a message to any integer or term without raising an error.
- `binary_to_term` checks atom encoding validity, and fix latin1 support (when non-ASCII chars are used)
- ESP32: fixed bug in `gpio:set_pin_mode/2` and `gpio:set_direction/3` that would accept any atom for the mode parameter without an error.
- ESP32: GPIO driver fix bug that would accept invalid `pull` direction, and silently set `pull` direction to `floating` without issuing an error.
- ESP32: fixed bug in gpio driver that would accept invalid pin numbers (either negative, or too large)
- RP2040: fixed bug in `gpio:set_pin_pull/2` that would accept any parameter as a valid `pull` mode.
- Support to function with 10 or more parameters
- Very unlikely but possible corruption caused by generated code that uses 16 live registers

14.10.3 Changed

- `binary_to_atom/2` validates utf8 strings
- `*_to_atom` and `atom_to_*` properly convert latin1 (not just ASCII) to utf8 and viceversa
- ESP32: use esp-idf v5.1.3 for building release binaries

14.11 [0.6.0-beta.0] - 2024-02-08

14.11.1 Added

- Added `esp:get_default_mac/0` for retrieving the default MAC address on ESP32.
- Added support for `pico` and `poci` as an alternative to `mosi` and `miso` for SPI
- ESP32: Added support to SPI peripherals other than hspi and vspi
- Added `gpio:set_int/4`, with the 4th parameter being the pid() or registered name of the process to receive interrupt messages
- Added support for `lists:split/2`
- Added ESP32 API for allowing coexistence of native and Erlang I2C drivers

14.11.2 Changed

- Shorten SPI config options, such as `sclk_io_num` -> `sclk`
- Shorten I2C config options, such as `scl_io_num` -> `scl`

- Shorten UART config options, such as `tx_pin` -> `tx`
- Introduced support to non-integer peripheral names, `"i2c0"`, `"uart1"` (instead of just `0` and
- `1`, which now they are deprecated)
- New atom table, which uses less memory, has improved performances and better code.
- SPI: when gpio number is not provided for `miso` or `mosi` default to disabled
- Change port call tuple format to the same format as gen_server, so casts can be supported too

14.11.3 Fixed

- Fix several missing memory allocation checks in libAtomVM.
- Fixed a possible memory leak in libAtomVM/module.c `module_destroy`.
- Fix possibile bug in random number generator on ESP32 and RPi2040
- Fixed interpretation of live for opcodes, thus altering GC semantics for nifs. See also UPDATING.

14.12 [0.6.0-alpha.2] - 2023-12-10

14.12.1 Fixed

- Fixed a bug where guards would raise exceptions instead of just being false
- Fixed support for big endian CPUs (such as some MIPS CPUs).
- Fixed STM32 not aborting when `AVM_ABORT()` is used
- Fixed a bug that would leave the STM32 trapped in a loop on hard faults, rather than aborting
- Fixed a bug that would make the VM to loop and failing to process selected fds on Linux
- Fixed classes of exceptions in estdlib.
- Fixed STM32 code that was hard coded to the default target device, now configured based on the `cmake -DDEVICE=` parameter
- Fixed hard fault on STM32 durung malloc on boards with more than one bank of sram
- Fixed invalid src_clk error on ESP-IDF >= 5.0
- Fixed changed default to `AVM_USE_32BIT_FLOAT=on` for STM32 platform to enable use of single precision hardware FPU on F4/F7 devices.
- Fixed a bug where emscripten `register_*_callback/1` functions would use x[1] as second argument
- Fixed precision of integers used with timers which could yield to halts and wait times smaller than expected
- Add support for ESP32-C6

14.12.2 Changed

- Crypto functions on generic_unix platform now rely on MbedTLS instead of OpenSSL
- Platform function providing time used by timers was changed from `sys_monotonic_millis` to `sys_monotonic_time_u64`, `sys_monotonic_time_u64_to_ms` and `sys_monotonic_time_ms_to_u64`.
- Implement `atomvm:random/0` and `atomvm:rand_bytes/1` on top of `crypto:strong_rand_bytes/1` on generic_unix, ESP32 and RP2040 platforms.

- Performance improvements

14.12.3 Added

- Added support for the OTP `socket` interface.
- Enhancd performance of STM32 by enabling flash cache and i-cache with branch prediction.
- Added cmake configuration option `AVM_CONFIG_REBOOT_ON_NOT_OK` for STM32
- New gpio driver for STM32 with nif and port support for read and write functions.
- Added support for interrupts to STM32 GPIO port driver.
- Added suppoprt for PicoW extra gpio pins (led) to the gpio driver.
- Added support for `net:getaddrinfo/1,2`
- Added minimal support for the OTP `ssl` interface.
- Added support for `crypto:one_time/4,5` on Unix and Pico as well as for `crypto:hash/2` on Pico
- Added ability to configure STM32 Nucleo boards onboard UART->USB-COM using the `-DBOARD=nucleo` cmake option
- Added STM32 cmake option `-DAVM_CFG_CONSOLE=` to select a different uart peripheral for the system console
- Added `crypto:strong_rand_bytes/1` using Mbed-TLS (only on generic_unix, ESP32 and RP2040 platforms)
- Added support for setting the default receive buffer size for sockets via `socket:setopt/3`
- Added support for pattern matching binaries containing 32 and 64 bit floating point values, but only when aligned to byte boundaries (e.g. `<<0:4, F:32/float>> = Bin` is not supported).
- Added experimental backend to `get_tcp` and `get_udp` based on the new `socket` interface
- Added API for managing ESP32 watchdog (only on `esp-idf` >= v5.x)

14.12.4 Removed

- OpenSSL support, Mbed-TLS is required instead.

14.13 [0.6.0-alpha.1] - 2023-10-09

14.13.1 Added

- Added erlang:spawn_link/1,3
- Added erlang:exit/2
- Added links to process_info/2
- Added lists:usort/1,2
- Added missing documentation and specifications for available nifs
- Added configurable logging macros to stm32 platform
- Added support for ULP wakeup on ESP32
- Added heap growth strategies as a fine-tuning option to `spawn_opt/2,4`
- Added `crypto:crypto_one_time/4,5` on ESP32
- Improved nif and port support on STM32

- Added support for `atomvm:posix_clock_settime/2`
- Added support for creations of binaries with unaligned strings
- Added `-h` and `-v` flags to generic_unix AtomVM command
- Removed support to ESP32 NVS from network module in order to make it generic. See also UPDATING.
- Added initial support for Pico-W: on-board LED, Wifi (STA and AP modes).

14.13.2 Changed

- Changed offset of atomvmlib and of program on Pico. See also UPDATING.

14.13.3 Fixed

- Fixed incorrect exit reason for exceptions of class exit
- Fixed several incorrect type specifications
- Fixed `esp:nvs_set_binary` functions.
- Fixed `monotonic_time/1` and `system_time/1` functions for Raspberry Pi Pico
- Fixed race conditions in atoms table.
- Fixed a bug in the STM32 port that caused the final result to never be returned.
- Fix bug when building a binary using a 64-bit integer on a 32-bit CPU.
- Fix (using 'auto' option) SPI on ESP32 models other than ESP32, such as ESP32S2, ESP32C3, …

14.14 [0.6.0-alpha.0] - 2023-08-13

14.14.1 Added

- Added the ability to specify the HSPI or VSPI ESP32 hardware interfaces when initializing the SPI Bus.
- Added support for the `spi:close/1` function.
- Added `AVM_VERBOSE_ABORT` CMake define, which when set to on, will print the C module and line number when a VM abort occurs. This define is off by default.
- Added `spi:write/3` and `spi:write_read/3` functions to support generalized SPI transactions and arbitrary-length reads and writes from SPI devices.
- Added support for building ESP32 port with all currently supported versions of Espressif ESP-IDF, version 4.1.x through 4.4.x.
- Added support for `controlling_process/2` in `gen_udp` and `gen_tcp` modules.
- Added ability to get the atomvm version via `erlang:system_info`.
- Added `erlang:is_boolean/1` Bif.
- Added support for `esp:partition_erase_range/2`
- Added support for `i2c:close/1`
- Added support for `erlang:unregister/1`
- Added Elixir ESP32 LEDC driver and example
- Added support for `uart:close/1`
- Added Bitwise support for Elixir

- Added support for esp32-s2, esp32-s3, and esp32-c3 chips.
- Added Elixir I2C driver and example
- Added the ability to specify the I2C port
- Added support for the OTP `math` module
- Added support for `erlang:integer_to_list/2` and `erlang:integer_to_binary/2`
- Added functions `esp:sleep_enable_ext0_wakeup/2` and `esp:sleep_enable_ext1_wakeup/2`.
- Added support for FP opcodes 94-102 thus removing the need for `AVM_DISABLE_FP=On` with OTP-22+
- Added support for stacktraces
- Added support for `utf-8`, `utf-16`, and `utf-32` bit syntax modifiers (put and match)
- Added support for Erlang `gpio:close/1` and Elixir `GPIO.close/1` for ESP32
- Added support for the Erlang `gen_event` module
- Added `start_link` support for the `network` module
- Added support for `erlang:monotonic_time/1`
- Added `start_link` support for the `gen_statem` module
- Added support for serializing floats in erlang external term encoding
- Added support for the `SMALL_BIG_EXT` erlang external term encoding
- Added support for `erlang:memory(binary)`
- Added support for callbacks on SNTP updates
- Multithreading support (SMP)
- Added support for code:load_abs/1, code:load_binary/3
- Added support for loading / closing AVMPacks at runtime
- Added support for ESP-IDF v5.x
- Added support for `calendar:system_time_to_universal_time/2`
- Added support for `calendar:datetime_to_gregorian_seconds/1`
- Added support for Raspberry Pi Pico
- Added support for nodejs with Wasm
- Added support for a subset of the OTP logger interface
- Added `esp:partition_list/0` function
- Added `esp:nvs_fetch_binary/2` and `nvs_put_binary/3` functions (`esp:nvs_set_binary` and functions that default to `?ATOMVM_NVS_NS` are deprecated now).
- Added most format possibilities to `io:format/2` and `io_lib:format/2`
- Added `unicode` module with `characters_to_list/1,2` and `characters_to_binary/1,2,3` functions
- Added support for `crypto:hash/2` (ESP32 and generic_unix with openssl)

14.14.2 Fixed

- Fixed issue with formatting integers with io:format() on STM32 platform
- Fixed a bug in the order of child initialization in the `supervisor` module
- Fixed a bug in the evaluation of `receive ... after infinity -> ...` expressions
- Fixed a bug in when putting integers in bit syntax with integer field sizes

- Fixed numerous bugs in memory allocations that could crash the VM
- Fixed SNTP support that had been broken in IDF 4.x builds
- Fixed `erlang:send/2` not sending to registered name

14.14.3 Breaking Changes

IMPORTANT: These changes are incompatible with previous releases of AtomVM.

- Changed the configuration model of the SPI driver, in order to allow for multiple "follower" devices to be attached to the same SPI Bus.
- Changed the return value from `erlang:system_info(esp32_chip_info)` from a tuple to a map, with additional information.
- Changed the return type of the `network:start` function to return the tuple `{ok, Pid}` on a successful call, instead of the bare atom `ok`. Applications that use `network:start` and check the return value will need to be modified.
- The return type of `i2c:read_bytes` has changed from returning just a binary to returning the tuple `{ok, Binary}` when successful.
- The return type of many `i2c` operations under error conditions has changed from `error` to `{error, Reason}`, for improved diagnostics.
- The eavmlib logger interface has been removed

14.14.4 Removed

- ESP-IDF v3.x support.

14.15 [0.5.1] - Unreleased

14.15.1 Added

- New function for atom comparison, useful when writing 3rd party components.
- New function for translating an atom term to an int value, according to a given translation table. This function can be used for translating an atom term to an enum const before doing a switch.
- New no-op `ATOM_STR(...)` macro for avoiding issues with clang-format.
- [ESP32] `REGISTER_PORT_DRIVER` for registering additional port drivers without editing any source file. This allows adding new components by just copying them to the components directory.
- [ESP32] `REGISTER_NIF_COLLECTION` for registering additional NIFs sets without editing any source file. This allows adding new NIFs by just copying them to the components directory.
- New function for getting a map or proplist value using an atom string without poluting the atom table.

14.15.2 Fixed

- Fix `gen_statem`: Cancel outstanding timers during state transitions in order to prevent spurious timeout messages from being sent to `gen_statem` process.
- Fix missing Elixir libraries: examvlib was not packed into atomvmlib.avm
- Fix `bs_context_to_binary`: match offset wasn't used, leading in certain situations to infinite loops while matching binaries.

- Fix how `start` option was handled from `bs_restore2` instruction: last saved match offset was used instead of match starting offset, causing some bytes being skipped.
- Fix another potential bug when doing pattern matching using code compiled with OTP 21.
- [ESP32] [UART]: Allow using different pins for rx, tx, cts and rts.
- [ESP32] [UART]: Replace custom UART handling with esp-idf UART event queues, hence other UARTs than UART0 are supported, with better performances and stability.
- Fix binaries concat (`bs_append` instruction) that was adding some extra zeroes at the end of built binaries.
- Fixed a bug in `gen_tcp` that prevents an accepting socket from inheriting settings on the listening socket.
- Fixed a bug in packing and unpacking integers into and from binaries when the bit length is not a multiple of 8.
- Fixed `esp:deep_sleep/1` that did not accept values above 31 minutes.
- Fixed a bug that could cause processes to hang indefinitely when calling ports that have terminated.
- Fixed potential VM crash when parsing external terms.
- Fixed the enforcement of `min_free_space` process option.

14.16 [0.5.0] - 2022-03-22

Chapter 15

Contributor Covenant Code of Conduct

15.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

15.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

15.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

15.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

15.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at davide AT uninstall.it. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

15.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

15.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

15.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

15.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

15.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

15.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

16 Security Policy

16.1 Supported Versions

Version	Supported
0.6.x	:white_check_mark:
< 0.5	:x:

16.2 Reporting a Vulnerability

To submit a vulnerability report, please contact us at davide at uninstall.it. Please, **do not use GitHub issues for vulnerability reports**. Your submission will be promptly reviewed and validated by a member of our team.

Chapter 17

AtomVM Update Instructions

17.1 v0.6.6 -> Unreleased

- Ports now represent the native processes. Port drivers should return a port (instead of a pid), by using `term_port_from_local_process_id` instead of `term_from_local_process_id`. Sockets, from port socket driver, are also represented by a port and some matching code may need to be updated from `is_pid/1` to `is_port/1`.
- Ports and pids can be registered. Function `globalcontext_get_registered_process` result now is a term that can be a `port()` or a `pid()`.

17.2 v0.6.4 -> v0.6.5

- ESP32: `esp32boot.avm` doesn't contain anymore Elixir standard library, use instead `elixir_esp32boot.avm`, or the Elixir release flavor when using an image.
- ESP32: partitioning schema for Elixir flavor is different, so app offset has been changed for Elixir images. Make sure to use `0x250000` as offset in your mix.exs or when performing manual flashing.
- ESP32 a bug was discovered in `i2c:write_bytes/2` that has not been fixed yet. Writing bytes sequentally using `i2c:write_byte/2` still works as a temporary workaround.
- STM32 devices with 512k of flash are not supported in this release, due to lack of flash space. Support may return in a future release.

17.3 v0.6.0-beta.1 -> v0.6.0-rc.0

- Drivers that send messages from Esp32 callbacks should use new functions `port_send_message_from_task`, `globalcontext_send_message_from_task` or `memory_destroy_heap_from_task` instead of `port_send_message`, `globalcontext_send_message` and `memory_destroy_heap`.

17.4 v0.6.0-alpha.2 -> v0.6.0-beta.0

- Registers are no longer preserved by GC by default when invoking nifs, as part of the fix of interpretation of the emulator of the live parameter of many opcodes. NIFs may need to call `memory_ensure_free_with_roots` and pass their arguments are roots, instead of `memory_ensure_free` or `memory_ensure_free_opt`.

- Port call message tuple format has been changed, hence previous version of the standard library cannot be used. **Libraries (or boot .avm file) from latest version must be used**.

17.5 v0.6.0-alpha.0 -> v0.6.0-alpha.1

- **Libraries (or boot .avm file) from latest version must be used**. Standard library from `v0.6.0-alpha.0` cannot work on top of latest version.
- Address (offset) of programs for Pico was changed from 0x100A0000 to 0x10100000. UF2 binaries need to be rebuilt with the proper offset using `uf2tool`.
- On ESP32, SSID and PSK stored in NVS are no longer read by network module. Applications must fetch the values and pass them to `network:start/1` or `network:start_link/1`.
- The `lib.avm` partition is no longer supported on ESP32. If you have been using a spacialized partitioning of your ESP32 flash (uncommon), AtomVM will no longer try to load code off this partition name.

owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   1. You must give any other recipients of the Work or Derivative Works a copy of this License; and

   2. You must cause any modified files to carry prominent notices stating that You changed the files; and

   3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

- genindex
- modindex

- search