AtomVM

Release 0.5.0

github.com/atomvm

Mar 22, 2022

Table of Contents

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 Alpha status. Software may contain bugs and should not be used for mission-critical applications. Application Programming Interfaces may change without warning.

[PDF][EPUB]

Chapter 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;
- 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 mind 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 than 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 tot he 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.

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

Getting Started Guide

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 Generic UNIX platform

2.1 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.

These instructions cover how to get the AtomVM virtual machine flashed to your ESP32 device, as well as how to flash your Erlang and Elixir programs that will be executed by the virtual machine running on the 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.

2.1.1 Requirements

Deployment of AtomVM applications requires the following components:

- A computer running MacOS or Linux (Windows support is not currently supported);
- 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;
- An Erlang/OTP release (21, 22, or 23);
- A serial console program, such as `minicom` or `screen`, so that you can view console output from your AtomVM application.
- (optional) For Erlang programs, `rebar3`;
- (optional) For Elixir programs, `mix`, which ships with the Elixir runtime;

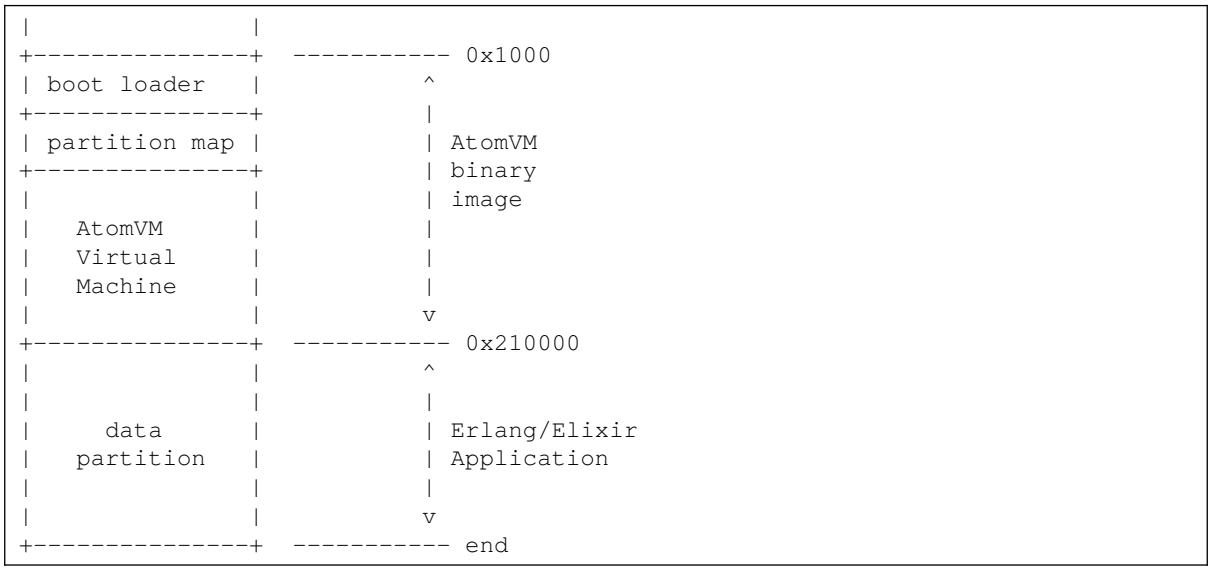
2.1.2 Deployment Overview

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.



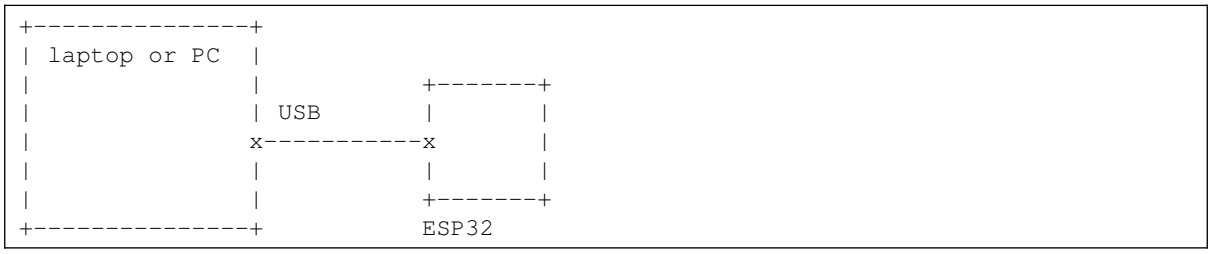
Deploying an AtomVM application to an ESP32 device typically involved two steps:

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.

2.1.3 Connecting the ESP32 device

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



> Note. 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.

2.1.4 Deploying the 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 images

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.,

```
shell$ esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 115200 erase_flash
...
```

> Note. Specify the device port and baud settings and AtomVM image name to suit your particular environment.

Next, download the latest stable or nightly ESP32 release image from the AtomVM web site.

> Note. Nightly images may be unstable and may result in unpredictable behavior.

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

```
shell$ esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 115200 \
    --before default_reset --after hard_reset \
    write_flash -u --flash_mode dio --flash_freq 40m --flash_size detect \
    0x1000 atomvm-esp32-v0.1.0.bin
...
```

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

Building 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

2.1.5 Deploying an AtomVM application

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`.

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.

This section describes both Erlang and Elixir tooling for deploying AtomVM applications to ESP32 devices.

Erlang Tooling

Deployment of AtomVM applications written in the Erlang programming language is supported via the `atomvm_rebar3_plugin` plugin, a community-supported plugin to the `rebar3` Erlang build

tool.

You can generate a simple application from scratch using the `atomvm_rebar3_plugin` template, as follows:

Edit or create the `$HOME/.config/rebar3/rebar.config` file to include the `atomvm_rebar3_plugin` plugin:

```
%% $HOME/.config/rebar3/rebar.config
{plugins, [
    {atomvm_rebar3_plugin, "0.3.0"},
    ...
]}.
```

In any directory in which you have write permission, issue

```
shell$ rebar3 new atomvm_app <app-name>
```

where `<app-name>` is the name of the application you would like to create (e.g., `myapp`). This command will generate a rebar project under the directory `<app-name>`.

The generated application will contain the proper `rebar.config` configuration and will contain the `<app-name>.erl` module, which exports the `start/0` function with a stubbed implementation.

Specifically, note the following stanza in the generated `rebar.config` file:

```
%% rebar.config
{plugins, [
    {atomvm_rebar3_plugin, "0.3.0"},
    ...
]}.
```

And note the `myapp` application exports a `start/0` function, e.g.,

```
%% erlang
-module(myapp).
-export([start/0]).

start() ->
    ok.
```

With this plugin installed, you have access to the `esp32_flash` target, which will build an AtomVM packbeam

```
shell$ rebar3 esp32_flash --port /dev/ttyUSB0
===> Fetching atomvm_rebar3_plugin v0.3.0
===> Fetching rebar3_hex v6.11.3
===> Fetching hex_core v0.7.1
===> Fetching verl v1.0.2
===> Analyzing applications...
===> Compiling verl
===> Compiling hex_core
===> Compiling rebar3_hex
===> Fetching atomvm_packbeam v0.3.0
===> Analyzing applications...
===> Compiling atomvm_rebar3_plugin
===> Compiling packbeam
===> Verifying dependencies...
===> Analyzing applications...
===> Compiling myapp
===> AVM file written to : myapp.avm
===> esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 115200 --before
default_reset --after hard_reset write_flash -u --flash_mode dio --flash_freq 40m
--flash_size detect 0x210000 /home/frege/myapp/_build/default/lib/myapp.avm
```

Note. Consult the the `atomvm_rebar3_plugin` plugin documentation, for more detailed information about how to use this tool.

Once the application has been flashed, you may connect to the ESP32 over the serial port using `minicom`, `screen`, or equivalent.

Elixir Tooling

TODO mix + https://github.com/atomvm/ExAtomVM + hex

2.2 Getting Started on the STM32 platform

TODO will se distribute a binary image for STM32?

2.3 Getting Started on the Generic UNIX platform

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

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.

```
shell$ AtomVM /path/to/myapp.avm
```

Currently, the `AtomVM` executable must be built from source.

See the AtomVM Build Instructions for instructions about how to build AtomVM on the Generic UNIX platform.

2.4 Where to go from here

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

- Example Programs
- Programmers Guide

Chapter 3

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.

3.1 AtomVM Features

Currently, AtomVM implements a strict subset of the BEAM instruction set, as of Erlang/OTP R21. Previous versions of Erlang/OTP are not supported.

A high level overview of the supported language features include:

- All the major Erlang types, including
    - integers (with size limits)
    - limited support for floats (not supported on all platforms)
    - 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

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

ing:

- 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)
    - deep sleep

3.1.1 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):

- Bingnums. Integer values are restricted to 64-bit values.
- SMP support. The AtomVM VM is currently a single-threaded process.
- 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.

3.2 AtomVM Development

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

3.2.1 Development Environment

In general, for most development purposes, you should be able to get away with an Erlang/OTP development environment (OTP21 or later), and for Elixir developers, and Elixir version TODO development environment. 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 devel-

opment 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

TODO

3.2.2 Development Workflow

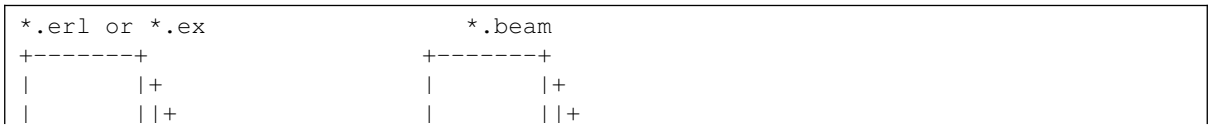
For the majority of users, AtomVM applications are written in the Erlang or Elixir programming language. These applications are compiled to BEAM (`.beam`) files using standard Erlang or Elixir compiler tool chains (`erlc`, `rebar`, `mix`, etc). The generated BEAM files contain byte-code that can be executed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

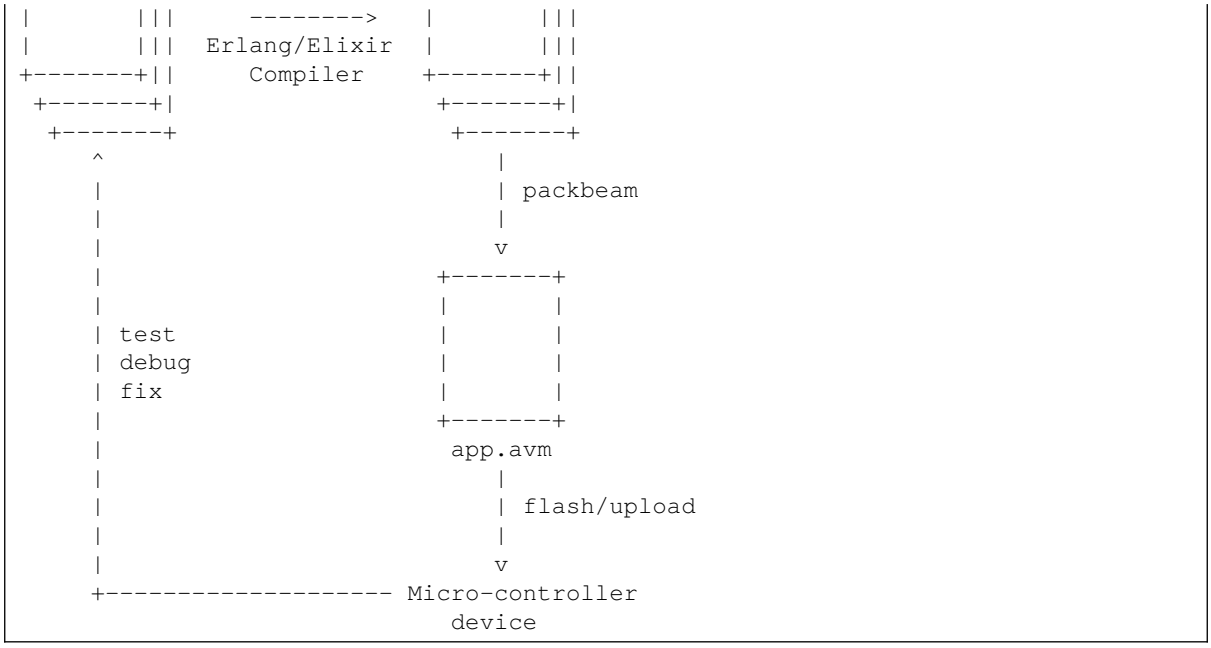
> Note. In a small number of cases, it may be useful to write parts of an application in the C programming language, as AtomVM nifs or ports. However, writing AtomVM nifs and ports is outside of the scope of this document.

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:





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

3.3 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.

> Note. The return value from the `start/0` function is ignored.

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

```
%% erlang
-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.,

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

3.3.1 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.

> Note. 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.

3.3.2 `PackBEAM` tool

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

```
shell$ PackBEAM -h
Usage: PackBEAM [-h] [-l] <avm-file> [<options>]
    -h                                              Print this help menu.
    -l <input-avm-file>                             List the contents of an AVM
file.
    [-a] <output-avm-file> <input-beam-or-avm-file>+  Create an AVM file (archive
if -a specified).
```

To create a packbeam file, specify the name of the AVM file to created (by convention, ending in `.avm`), followed by a list of BEAM files:

```
shell$ PackBEAM foo.avm path/to/foo.beam path/to/bar.beam
```

You can also specify another AVM file to include. Thus, for example, to add to BEAM file to an existing AVM file, you might enter:

```
shell$ PackBEAM foo.avm foo.avm path/to/gnu.beam
```

To list the contents of an AVM file, use the `-l` flag:

```
shell% PackBEAM -l foo.avm
foo.beam *
bar.beam
gnu.beam
```

Any BEAM files that export a `start/0` function will contain an asterisk (`*`) in the AVM file contents.

3.3.3 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.,

```
shell$ AtomVM foo.avm
```

> Note. 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.

3.4 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.

3.4.1 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.

3.4.2 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
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:

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

Note that the `io_lib` module can be used to format string data, as well.

> Note. Formatting parameters are currently limited to `~p`, `~s`, and `~n`.

3.4.3 Process Management

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

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

And for each process, you can get detailed process information via the `erlang:process_info/1` function:

```
%% erlang
[io:format("Process info for Pid ~p: ~p~n", [Pid, erlang:process_info(Pid)]) ||
Pid <- Pids].
```

The return value is a property list containing values for `heap_size`, `stack_size`, `message_queue_len`,and `memory` consumed by the process.

3.4.4 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).

For example,

```
%% erlang
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.

```
%% erlang
```

```
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.

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

3.4.5 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):

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

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

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

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

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

> Note. Setting the system time is done in a platform-specific manner. For information about how to set system time on the ESP32, see the Network Programming Guide.

3.4.6 Miscellaneous

Use the `erlang:md5/1` function to compute the MD5 hash of an input binary. The output is a fixed-length binary ()

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

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

```
%% erlang
RandomInetger = atomvm:random().
```

Use `atomvm:random_bytes/1` to return a randomly populated binary of a specified size:

```
%% erlang
RandomBinary = erlang:random_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.

```
%% erlang
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.

3.5 ESP32-specific APIs

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

3.5.1 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:

- `esp_free_heap_size` Returns the available free space in the ESP32 heap
- `esp_chip_info` Returns 4-tuple of the form `{esp32, Features, Cores, Revision}`, where `Features` is a bit mask of features enabled in the chip, `Cores` is the number of CPU cores on the chip, and `Revision` is the chip version.
- `esp_idf_version` Return the IDF SDK version, as a string.

For example,

```
%% erlang
FreeHeapSize = erlang:system_info(esp_free_heap_size).
```

3.5.2 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.

> Note. 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:set_binary/3` function, and specify a namespace, key, and value:

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

To retrieve a value in non-volatile storage, use the `esp: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:

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

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

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

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

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

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

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

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

> Note. 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.

3.5.3 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.

```
%% erlang
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.

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

Use the `esp:sleep_get_wakeup_cause/0` function can be used to inspect the reason for a wakeup. Currently, the only supported return value is the atom `undefined` or `sleep_wakeup_timer`.

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

3.5.4 Miscellaneous

The `freq_hz` function can be used to retrieve the clock frequency of the chip.

- `esp:freq_hz/0`

3.6 Peripherals

The AtomVM virtual machine and libraries support APIs for interfacing with peripheral devices connected to the ESP32. This section provides information about these APIs.

3.6.1 GPIO

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 direction of the pin using the `gpio:set_direction/2` function, using `input` or `output` as the direction parameter.

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

```
%% erlang
Pin = 2,
gpio:set_direction(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.
```

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

```
%% erlang
Pin = 2,
gpio:set_direction(Pin, output),
gpio:digital_write(Pin, low).
```

Interrupt Handling

You can get notified of changes in the state of a GPIO pin by using the `gpio:set_int/2` function. This function takes a reference to a GPIO 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 containing the atom `gpio_interrupt` and the pin.

```
%% erlang
Pin = 2,
gpio:set_direction(Pin, input),
```

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

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

3.6.2 I2C

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

> Note. 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_io_num`	`integer()`	yes	I2C clock pin (SCL)
`sda_io_num`	`integer()`	yes	I2C data pin (SDA)
`i2c_clock_hz`	`integer()`	yes	I2C clock frequency (in hertz)

For example,

```
%% erlang
I2C = i2c:open([
    {scl_io_num, 21}, {sda_io_num, 22}, {i2c_clock_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:qwrite_bytes/2,3` write operations between calls to `i2c:begin_transmission/1` 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` 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.,

```
%% erlang
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:

```
%% erlang
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:

```
%% erlang
BinaryData = i2c:read_bytes(I2C, DeviceAddress, Register, Len)
```

3.6.3 SPI

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

> Note. 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 two elements:

- `bus_config` – a properties list containing entries for the SPI bus
- `device_config` – a properties list containing entries for the device

The `bus_config` properties list contains the following entries:

Key	Value Type	Required	Description
`miso_io_num`	`integer()`	yes	SPI leader-in, follower-out pin (MOSI)
`mosi_io_num`	`integer()`	yes	SPI leader-out, follower-in pin (MISO)
`sclk_io_num`	`integer()`	yes	SPI clock pin (SCLK)

The `device_config` – a properties list containing entries for the device properties list contains the following entries:

Key	Value Type	Required	Description
`spi_clock_hz`	`integer()`	yes	SPI clock frequency (in hertz)
`spi_mode`	`integer()`	yes	SPI mode
`spi_cs_io_num`	`integer()`	yes	SPI chip select pin (CS)
`address_len_bits`	`integer()`	yes	number of bits in a read/write operation (for example, 8, to read and write single bytes at a time)

For example,

```
%% erlang
SPIConfig = [
    {bus_config, [
        {miso_io_num, 12},
        {mosi_io_num, 13},
        {sclk_io_num, 14}
    ]},
    {device_config, [
        {spi_clock_hz, 1000000},
        {spi_mode, 0},
        {spi_cs_io_num, 18},
        {address_len_bits, 8}
    ]}
],
```

```
SPI = spi:open(SPIConfig),
...
```

Once the port is opened, you can use the returned `SPI` instance 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/3` function:

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

To write a byte at a given address on the device, use the `spi_write_at/4` function:

```
%% erlang
write_at(SPI, Address, 8, Byte)
```

> Note. The `spi:write_at/4` takes integer values as inputs and the `spi:read_at/3` 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.

3.6.4 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.

> Note. 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. Legal values are:

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

The selection of the hardware interface dictates the default RX and TX pins on the ESP32:

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

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

Key	Value Type	Required	Default Value	Description
`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

For example,

```
%% erlang
UART = uart:open("UART0", [{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:

```
%% erlang
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:

```
%% erlang
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.

3.6.5 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.

```
%% erlang
-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).
```

3.7 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.

3.7.1 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.

> Note. 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

> Note. 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:

```
%% erlang
```

```
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 reconnections 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.

3.7.2 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.

> Note. Not all of the Erlang/OTP `gen_udp` functionality is implemented in AtomVM. For details, consults 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}]) 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 binary.

```
%% erlang
receive
```

```
    {udp, _Socket, Address, Port, Packet} ->
        io:format("Received UDP packet ~p from address ~p port ~p~n", [Packet,
Address, 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:

```
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
```

Note. IPv6 networking is not currently supported in AtomVM.

3.7.3 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.

Note. 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.

Note. 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 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 continuosuly 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.

Chapter 4

Example Programs

AtomVM includes a collection of useful examples for getting started. This section describes what these examples do, and how to run them, for example, on an ESP32 device.

4.1 Erlang Examples

Erlang examples may be run in the UNIX shell or on supported microcontroller devices.

4.1.1 `hello_world`

This example program prints the string "Hello World" and quits.

Command line

The `hello_world.avm` file will get created as part of a build. This file may be supplied as an argument to the `AtomVM` command:

```
shell$ ./src/AtomVM ./examples/erlang/hello_world.avm
Hello World
Return value: ok
```

4.1.2 `udp_server`

This example program listens on UDP port 44444 and will print information about the received message, including the source IP, (ephemeral) source port, and packet received, to the console.

Command line

The `udp_server.avm` file will get created as part of a build. This file may be supplied as an argument to the `AtomVM` command:

```
shell$ ./src/AtomVM ./examples/erlang/udp_server.avm
Opened UDP socket on "0.0.0.0:44404".
Waiting to receive data...
```

You can send UDP packets to the AtomVM instance using `netcat` (or `nc` on some platforms), in a separate terminal window:

```
shell$ nc -u localhost 44404
```

This command will wait for you to enter a line of text, e.g.,

```
testing 1 2 3
```

In the AtomVM termianl window, you see:

```
Received UDP packet <<116,101,115,116,105,110,103,32,49,32,50,32,51,10>> from
"127.0.0.1:55261"
Waiting to receive data...
```

Note. Netcat appends a newline character at the end of the input, so the packet binary does not display as printable text.

4.1.3 udp_client

This example program send the packet of data (":?????") over UDP to port 44444 on the loopback address every 5 seconds, in a loop. The program will print a period (`.`) to the console, every time it sends a message.

This command may be used in tandem with the `udp_server` program to illustrate sending messages between AtomVM processes over UDP.

Command line

The `udp_client.avm` file will get created as part of a build. This file may be supplied as an argument to the `AtomVM` command:

```
shell$ ./src/AtomVM ./examples/erlang/udp_client.avm
Opened UDP socket on "0.0.0.0:63665".
Sent <<58,-94,-56,-32,54,45>>
Sent <<58,-94,-56,-32,54,45>>
Sent <<58,-94,-56,-32,54,45>>
...
```

If you are running the `udp_server` program, you should see messages like the following printed to the console:

```
Received UDP packet <<58,-94,-56,-32,54,45>> from "127.0.0.1:63665"
Waiting to receive data...
Received UDP packet <<58,-94,-56,-32,54,45>> from "127.0.0.1:63665"
Waiting to receive data...
Received UDP packet <<58,-94,-56,-32,54,45>> from "127.0.0.1:63665"
Waiting to receive data...
...
```

Note. AtomVM does not currently treat characters outside of the printable ASCII character set as printable characters.

4.1.4 tcp_server

This example program listens on TCP port 44404 and accept connections on that port. Once accepted, it will wait for packets to be sent from the client. Once received, the server will print the packet received to the console, and then echo the packet back to the calling client.

Command line

The `tcp_server.avm` file will get created as part of a build. This file may be supplied as an argument to the `AtomVM` command:

```
shell$ ./src/AtomVM ./examples/erlang/tcp_server.avm
Listening on "0.0.0.0:44404".
Waiting to accept connection...
```

You can send TCP packets to the AtomVM instance using `netcat` (or `nc` on some platforms), in a separate terminal window:

```
shell$ nc localhost 44404
```

This will open a TCP connection to the `tcp_server`, and you should see the following on the

console:

```
Accepted connection.  local: "127.0.0.1:44404" peer: "127.0.0.1:56628"
Waiting to receive data...
Waiting to accept connection...
```

The netcat command will wait for you to enter a line of text, e.g.,

```
testing 1 2 3
```

In the AtomVM terminal window, you see:

```
Received packet [116,101,115,116,105,110,103,32,49,32,50,32,51,10] from
"127.0.0.1:56628".  Echoing back...
Waiting to receive data...
```

> Note. Netcat appends a newline character at the end of the input, so the packet binary does not display as printable text.

You may enter as much data as you like, though by default, the packet size will be limited to 128 bytes.

If you stop the `netcat` command (via ^C), you should

```
Connection closed.
```

printed to the AtomVM console.

Note that you can have multiple, concurrent TCP/IP connections to your AtomVM server.

4.1.5 `tcp_client`

This example program send the packet of data (":?????") over TCP to port 44404 on the loopback address every 1 second, in a loop. The program will wait for a response from the server, before proceeding.

This command may be used in tandem with the `tcp_server` program to illustrate sending messages between AtomVM processes over TCP.

> Note. You will need to change the `Address` variable in the `tcp_client.erl` program in order to test against an AtomVM server running on a different host or device.

Command line

The `tcp_client.avm` file will get created as part of a build. This file may be supplied as an argument to the `AtomVM` command:

```
shell$ ./src/AtomVM ./examples/erlang/tcp_client.avm
Connected to "127.0.0.1:44404" from "127.0.0.1:56741"
Sent <<58,-94,-56,-32,54,45>> to "127.0.0.1:44404"
Received [58,162,200,224,54,45] from "127.0.0.1:44404"
Sent <<58,-94,-56,-32,54,45>> to "127.0.0.1:44404"
Received [58,162,200,224,54,45] from "127.0.0.1:44404"
Sent <<58,-94,-56,-32,54,45>> to "127.0.0.1:44404"
Received [58,162,200,224,54,45] from "127.0.0.1:44404"
...
```

If you are running the `tcp _server` program, you should see messages like the following printed to the console:

```
Accepted connection.  local: "127.0.0.1:44404" peer: "127.0.0.1:56741"
Waiting to receive data...
Waiting to accept connection...
Received packet [58,162,200,224,54,45] from "127.0.0.1:56741".  Echoing back...
Waiting to receive data...
```

```
Received packet [58,162,200,224,54,45] from "127.0.0.1:56741".  Echoing back...
Waiting to receive data...
Received packet [58,162,200,224,54,45] from "127.0.0.1:56741".  Echoing back...
Waiting to receive data...
...
```

> Note. AtomVM does not currently treat characters outside of the printable ASCII character set as printable characters.

You may run multiple concurrent instances of the `tcp_client` against a single `tcp_server` instance.

4.2 ESP32 Examples

AtomVM includes examples that are specifically designed for the ESP32 and other microcontrollers.

4.3 Flashing AtomVM Examples for ESP32

In order to run the ESP32 examples, you will need to flash the example AVM files that are created as part of the build to your device.

In the remainder of this document, we assume the `flash.sh` script, located in the `tools/dev` directory of the AtomVM source tree.

> Note. You must set the `ESP_IDF` environment variable to the root directory of the ESP IDF SDK installation on your development machine.

You can control the serial port and baud rate via the `FLASH_SERIAL_PORT` and `FLASH_BAUD_RATE` environment variables, e.g.,

```
shell$ export FLASH_SERIAL_PORT="/dev/tty.SLAB_USBtoUART"
shell$ export FLASH_BAUD_RATE=921600
```

The default values for these variables, if not set, are `/dev/ttyUSB0` and `115200`, respectively.

> Note. Experiment with baud rates (e.g., 921600). You may find you can shorten the flash-debug-flash cycle with higher rates.

You can montor the console output of these examples by issuing the `monitor` target to `make`, in the `src/platforms/esp32` directory of the AtomVM source tree:

```
shell$ make monitor
MONITOR
--- WARNING: Serial ports accessed as /dev/tty.* will hang gdb if launched.
--- Using /dev/cu.SLAB_USBtoUART instead...
--- idf_monitor on /dev/cu.SLAB_USBtoUART 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets Jun  8 2016 00:22:57
...
```

4.3.1 `blink`

The `blink` example will turn the blue LED on an ESP32 SoC (pin 2) on and off, once every second.

Flash the example program to your device as follows:

```
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/blink.avm
...
Hard resetting via RTS pin...
```

You should see the blue LED turn on and off on your ESP32 device.

4.3.2 esp_random

This demo program illustrates use of the ESP32 `random`, `restart`, and `reset_reason` functions.

The program will generate a random binary of a random size (at most 127 bytes) every 5 seconds. If a 0-length byte sequence is generated (1:128 probability), the ESP will restart.

Flash the example program to your device as follows:

```
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/esp_random.avm
...
Hard resetting via RTS pin...
```

You should see something like the following output when monitoring the ESP32 output (truncated for brevity):

```
shell$ make monitor
...
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: esp_random.beam...
---
Reset reason: esp_rst_poweron
Random bytes: <<71,13,221,24,8,15,...,197,120,152,205>>
Random bytes: <<37,155,124,177,44,141,40,106,...,43,48,62,109,2,78,39,107>>
Random bytes:
<<217,210,239,183,...,78,68,253,146,212,71,17,208,219,126,240,218,34,0,152,80,20,166,194,106>>
Random bytes: <<112,77,123,249,162,...,238,237,128,227,58,29,64,74>>
...
<<"">>
ets Jun  8 2016 00:22:57
...
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: esp_random.beam...
---
esp_rst_sw
Random bytes:
<<155,174,204,143,232,202,136,...,118,177,77,230,10,21,72,91,92,160,198,115,249,217,206,52,102,3
...
```

4.3.3 esp_nvs

This demo program illustrates the use of ESP32 non-volatile storage (NVS).

The program will store the number of times the device has been rebooted, along with the start time, in NVS.

Flash the example program to your device as follows:

```
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/esp_nvs.avm
...
Hard resetting via RTS pin...
```

You should see the following output when monitoring the ESP32 output (truncated for brevity):

```
shell$ make monitor
...
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: esp_nvs.beam...
---
Saving count 0 to NVS...
```

```
Reset device to increment.
AtomVM finished with return value = ok
going to sleep forever..
```

Hit the reset button on your device, and the ESP device will reboot, and display something like the following:

```
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: esp_nvs.beam...
---
Saving count 1 to NVS...
Reset device to increment.
AtomVM finished with return value = ok
going to sleep forever..
```

4.3.4 `reformat_nvs`

This demo program will reformat the non-volatile storage (NVS) partition.

Flash the example program to your device as follows:

```
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/reformat_nvs.avm
...
Hard resetting via RTS pin...
```

You should see the following output when monitoring the ESP32 output (truncated for brevity):

```
shell$ make monitor
...
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: esp_nvs.beam...
---
Warning: Reformatted NVS partition!
AtomVM finished with return value = ok
going to sleep forever..
```

The NVS partition on your ESP device should be reformatted.

> Note. This program will irrevocably delete all existing key-values stored on the NVS partition. Use with caution.

4.3.5 `set_network_config`

This demo program can be used to set the WIFI credentials in NVS. Setting WIFI credentials in NVS can greatly simplify the task of running ESP programs that require connectivity to WIFI networks.

> Note. Credentials are stored unencrypted and in plaintext and should not be considered secure. Future versions may use encrypted NVS storage.

Edit the `sta_network_config.erl` program and set the `Ssid` binary with your WIFI AP SSID, and `Psk` binary with the password used to access your WIFI network. Save the file, rebuild, and flash to your device:

```
shell$ make
...
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/sta_network_config.avm
...
Hard resetting via RTS pin...
```

You should see the following output when monitoring the ESP32 output (truncated for brevity):

```
shell$ make monitor
```

```
...
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: set_network_config.beam...
---
{atomvm,sta_ssid,<<"myssid">>}
{atomvm,sta_psk,<<"xxxxxx">>}
AtomVM finished with return value = ok
going to sleep forever..
```

You may now run programs that use your WIFI network (see below) without needing to enter WIFI credentials.

4.3.6 sta_network

The `sta_network` example will connect to your local WiFi network and obtain and IP address. Once a connection is established, a `connected` message will be displayed. Once an IP address is obtained, the device IP address, netmask, and gateway will be displayed on the console.

> Note. AtomVM currently only supports station mode (STA).
>
> Note. AtomVM currently only supports IPv4 addresses.
>
> Note. If you have not set WIFI credentials in NVS (see above), you will need to edit the `examples/erlang/esp32/sta_network.erl` source file and set the `ssid` and `psk` parameters to match your local WiFi network, and then rebuild the example.

Flash the example program to your device as follows:

```
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/sta_network.avm
...
Hard resetting via RTS pin...
```

You should see the following output when monitoring the ESP32 output (truncated for brevity):

```
shell$ make monitor
...
Starting: sta_network.beam...
---
I (220) wifi: wifi driver task: 3ffc3b54, prio:23, stack:3584, core=0
I (220) wifi: wifi firmware version: d5da5a5
I (220) wifi: config NVS flash: enabled
I (220) wifi: config nano formatting: disabled
I (230) system_api: Base MAC address is not set, read default base MAC address
from BLK0 of EFUSE
I (240) system_api: Base MAC address is not set, read default base MAC address
from BLK0 of EFUSE
I (270) wifi: Init dynamic tx buffer num: 32
I (270) wifi: Init data frame dynamic rx buffer num: 32
I (270) wifi: Init management frame dynamic rx buffer num: 32
I (270) wifi: Init static rx buffer size: 1600
I (280) wifi: Init static rx buffer num: 10
I (280) wifi: Init dynamic rx buffer num: 32
I (290) NETWORK: starting wifi: SSID: [myssid], password: [XXXXXXXX].
I (360) phy: phy_version: 4000, b6198fa, Sep  3 2018, 15:11:06, 0, 0
I (370) wifi: mode : sta (3c:71:bf:84:d9:08)
I (370) NETWORK: SYSTEM_EVENT_STA_START received.
I (490) wifi: n:1 0, o:1 0, ap:255 255, sta:1 0, prof:1
I (1470) wifi: state: init -> auth (b0)
I (1480) wifi: state: auth -> assoc (0)
I (1490) wifi: state: assoc -> run (10)
I (1500) wifi: connected with myssid, channel 1
I (1500) wifi: pm start, type: 1
I (1500) NETWORK: SYSTEM_EVENT_STA_CONNECTED received.
```

```
I (3690) event: sta ip: 192.168.1.236, mask: 255.255.255.0, gw: 192.168.1.1
I (3690) NETWORK: SYSTEM_EVENT_STA_GOT_IP: 192.168.1.236
connected
{{192,168,1,236},{255,255,255,0},{192,168,1,1}}
```

4.3.7 udp_server_blink

The `udp_server_blink` example will connect to your local WiFi network and obtain and IP address. It will then start a UDP server on port 44444. When a UDP message is received, the blue LED on the ESP32 SoC (pin 2) will toggle on and off.

> Note. AtomVM currently only supports station mode (STA).
>
> Note. AtomVM currently only supports IPv4 addresses.
>
> Note. You will need to edit the `examples/erlang/esp32/udp_server_blink.erl` source file and set the `ssid` and `psk` parameters to match your local WiFi network, and then rebuild the example.

Flash the example program to your device as follows:

```
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/udp_server_blink.avm
...
Hard resetting via RTS pin...
```

You should see the following output when monitoring the ESP32 output (truncated for brevity):

```
shell$ make monitor
...
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: udp_server_blink.beam...
---
I (222) wifi: wifi driver task: 3ffc3de8, prio:23, stack:3584, core=0
I (222) wifi: wifi firmware version: d5da5a5
I (222) wifi: config NVS flash: enabled
I (232) wifi: config nano formatting: disabled
I (232) system_api: Base MAC address is not set, read default base MAC address
from BLK0 of EFUSE
I (242) system_api: Base MAC address is not set, read default base MAC address
from BLK0 of EFUSE
I (312) wifi: Init dynamic tx buffer num: 32
I (312) wifi: Init data frame dynamic rx buffer num: 32
I (312) wifi: Init management frame dynamic rx buffer num: 32
I (312) wifi: Init static rx buffer size: 1600
I (322) wifi: Init static rx buffer num: 10
I (322) wifi: Init dynamic rx buffer num: 32
I (332) NETWORK: starting wifi: SSID: [myssid], password: [XXXXXXXX].
I (442) phy: phy_version: 4000, b6198fa, Sep  3 2018, 15:11:06, 0, 0
I (442) wifi: mode : sta (3c:71:bf:84:d9:08)
I (442) NETWORK: SYSTEM_EVENT_STA_START received.
I (572) wifi: n:1 0, o:1 0, ap:255 255, sta:1 0, prof:1
I (1552) wifi: state: init -> auth (b0)
I (1552) wifi: state: auth -> assoc (0)
I (1562) wifi: state: assoc -> run (10)
I (1582) wifi: connected with myssid, channel 1
I (1582) wifi: pm start, type: 1
I (1582) NETWORK: SYSTEM_EVENT_STA_CONNECTED received.
I (2212) event: sta ip: 192.168.1.236, mask: 255.255.255.0, gw: 192.168.1.1
I (2212) NETWORK: SYSTEM_EVENT_STA_GOT_IP: 192.168.1.236
Acquired IP address: "192.168.1.236" Netmask: "255.255.255.0" Gateway:
"192.168.1.1"
Opened UDP socket on "0.0.0.0:44404".
Waiting to receive data...
```

You can send UDP packets to the AtomVM instance using `netcat` (or `nc` on some platforms), in a separate terminal window:

```
shell$ nc -u 192.168.1.236 44404
```

Every time you enter a line of text, the blue LED on the ESP32 SoC (pin 2) should toggle on and off, and you should see output on the console, such as

```
Received UDP packet <<100,115,102,115,100,10>> from "192.168.1.237:53291"
Waiting to receive data...
```

4.3.8 `tcp_server_blink`

The `tcp_server_blink` example will connect to your local WiFi network and obtain and IP address. It will then start a TCP server on port 44404. When a TCP message is received, the blue LED on the ESP32 SoC (pin 2) will toggle on and off.

> Note. AtomVM currently only supports station mode (STA).
>
> Note. AtomVM currently only supports IPv4 addresses.
>
> Note. You will need to edit the `examples/erlang/esp32/tcp_server_blink.erl` source file and set the `ssid` and `psk` parameters to match your local WiFi network, and then rebuild the example.

Flash the example program to your device as follows:

```
shell$ ./tools/dev/flash.sh build/examples/erlang/esp32/tcp_server_blink.avm
...
Hard resetting via RTS pin...
```

You should see the following output when monitoring the ESP32 output (truncated for brevity):

```
shell$ make monitor
Found AVM partition: size: 1048576, address: 0x110000
Booting file mapped at: 0x3f420000, size: 1048576
Starting: tcp_server_blink.beam...
---
start
I (296) wifi: wifi driver task: 3ffc650c, prio:23, stack:3584, core=0
I (296) wifi: wifi firmware version: 9415913
I (296) wifi: config NVS flash: enabled
I (296) wifi: config nano formatting: disabled
I (296) system_api: Base MAC address is not set, read default base MAC address
from BLK0 of EFUSE
I (306) system_api: Base MAC address is not set, read default base MAC address
from BLK0 of EFUSE
I (346) wifi: Init dynamic tx buffer num: 32
I (346) wifi: Init data frame dynamic rx buffer num: 32
I (346) wifi: Init management frame dynamic rx buffer num: 32
I (346) wifi: Init static rx buffer size: 1600
I (356) wifi: Init static rx buffer num: 10
I (356) wifi: Init dynamic rx buffer num: 32
I (366) NETWORK: starting wifi: SSID: [myssid], password: [XXXXXXXX].
I (446) phy: phy_version: 4008, c9ae59f, Jan 25 2019, 16:54:06, 0, 0
I (446) wifi: mode : sta (3c:71:bf:84:d9:08)
I (446) NETWORK: SYSTEM_EVENT_STA_START received.
I (1176) wifi: n:6 0, o:1 0, ap:255 255, sta:6 0, prof:1
I (2156) wifi: state: init -> auth (b0)
I (2166) wifi: state: auth -> assoc (0)
I (2166) wifi: state: assoc -> run (10)
I (2196) wifi: connected with myssid, channel 6
I (2196) wifi: pm start, type: 1
I (2196) NETWORK: SYSTEM_EVENT_STA_CONNECTED received.
```

```
I (2746) event: sta ip: 192.168.1.236, mask: 255.255.255.0, gw: 192.168.1.1
I (2746) NETWORK: SYSTEM_EVENT_STA_GOT_IP: 192.168.1.236
Acquired IP address: "192.168.1.236" Netmask: "255.255.255.0" Gateway:
"192.168.1.1"
Listening on "0.0.0.0:44404".
Waiting to accept connection...
```

You can send TCP packets to the AtomVM instance using `netcat` (or `nc` on some platforms), in a separate terminal window, e.g.,

```
shell$ nc 192.168.1.236 44404
```

On the ESP32 console, you should see:

```
Accepted connection.  local: "192.168.1.236:44404" peer: "192.168.1.237:55275"
Waiting to receive data...
Waiting to accept connection...
```

Every time you enter a line of text, the blue LED on the ESP32 SoC (pin 2) should toggle on and off, and the data you entered should get echoed back to the `netcat` console.

On the ESP32 console, you should see:

```
Received packet [115,100,102,115,100,102,10] from "192.168.1.237:55275".  Echoing
back...
Waiting to receive data...
```

every time a packet is sent to the server.

Network Programming Guide

One of the exciting features of the ESP32 is its support for WiFi networking, allowing ESP32 micro-controllers to communicate with the outside world over common IP networking protocols, such as TCP or IDP. The ESP32 and ISF SDK supports configuring an ESP32 in station mode (STA), whereby the device connects to an existing access point, as well as "softAP" mode (AP), whereby it functions 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 devices, providing support for configuring your ESP32 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.

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, 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.

5.1 Station (STA) mode

In STA mode, the ESP32 connects 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.

  Note that the station mode SSID and password *may* be stored in non-volatile storage, in which case these parameters may be skipped. See the "NVS Credentials" section below, for more information about using non-volatile storage to store credentials that persist across device

reboots.

The `network:start/1` will immediately return `ok`, 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.

  Note. 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:

- `{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).

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.

```
%% erlang
Config = [
    {sta, [
        {ssid, <<"myssid">>},
        {psk,  <<"mypsk">>},
        {connected, fun connected/0,
        {got_ip, fun got_ip/1},
        {disconnected, fun disconnected/0}
        {dhcp_hostname, <<"myesp32">>}
    ]}
],
ok = network:start(Config),
...
```

The following callback functions will be called when the corresponding events occur during the lifetime of the network connection.

```
%% erlang
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.

5.1.1 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:

```
%% erlang
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
```

5.2 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.

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.

> Note that the station mode SSID and password *may* be stored in non-volatile storage, in which case these parameters may be skipped. See the "NVS Credentials" section below, for more information about using non-volatile storage to store credentials that persist across device reboots.

The `network:start/1` will immediately return `ok`, 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.

  Note. 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 = network:start(Config),
...
```

The following callback functions will be called when the corresponding events occur during the lifetime of the network connection.

```
%% erlang
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.

5.2.1 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:

```
%% 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
```

5.3 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.

5.4 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 `endpoint` property tag, e.g.,

```
{sntp, [{endpoint, <<"pool.ntp.org">>}]}
```

The endpoint value can be a string or binary.

5.5 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

under the `atomvm` namespace. The following entries may be specified in non-volatile storage:

namespace	mode	key	type	value
`atomvm`	STA	`sta_ssid`	`binary()`	Station ID
`atomvm`	STA	`sta_psk`	`binary()`	Station password (if applicable)
`atomvm`	AP	`ap_ssid`	`binary()`	Access Point ID
`atomvm`	AP	`ap_psk`	`binary()`	Access Point password (if applicable)

If set in NVS storage, you may remove the corresponding `ssid` and `psk` parameters from the configuration used to initialize the network, and the SSID and PSK configured in NVS will be used, instead. An SSID or PSK defined explicitly in configuration will override any values in NVS.

You can set these credentials once, as follows:

```
esp:nvs_set_binary(atomvm, sta_ssid, <<"myssid">>).
esp:nvs_set_binary(atomvm, sta_psk, <<"mypsk">>).
```

or

```
esp:nvs_set_binary(atomvm, ap_ssid, <<"myssid">>).
esp:nvs_set_binary(atomvm, ap_psk, <<"mypsk">>).
```

With these settings, you can run ESP programs that initialize the network without configuring your SSID and PSK explicitly in source code.

> Note. Credentials are stored un-encrypted and in plaintext and should not be considered secure. Future versions may use encrypted NVS storage.

5.6 Stopping the Network

To stop the network and free any resources in use, issue the `stop/0` function:

```
network:stop().
```

> Note. This function is currently not well tested.

Chapter 6

Build Instructions

This guide is intended for anyone interested in the implementation of AtomVM, including developers who would like to provide enhancements or bug fixes to the core AtomVM virtual machine, as well as providing third-party extensions via the implementation of AtomVM Nifs or Ports. Understanding the virtual machine and how it is implemented is also interesting in its own right, and having an understanding of the underlying mechanics can be helpful for writing better Erlang and Elixir programs.

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. Currently, Erlang/OTP versions 21 and 22 are supported.

This guide provides information about how to build AtomVM for the various supported platforms (Generic UNIX, ESP32, and STM32).

> Note. 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.

6.1 Downloading AtomVM

The AtomVM source code is available by cloning the AtomVM github repository:

```
shell$ git clone https://github.com/atomvm/AtomVM
```

> Note. Downloading the AtomVM github repository requires the installation of the `git` program. Consult your local OS documentation for installation of the `git` package.

6.2 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).

For information about porting to new platforms, see Porting to new platforms, below.

6.3 Building for Generic UNIX

The following instructions apply to unix-like environments, including Linux, FreeBSD, and MacOS.

> Note. 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.

6.3.1 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`
- Erlang/OTP 21

Consult your local OS documentation for instructions about how to install these components.

6.3.2 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 clcal `build` directory, and then point `cmake` to the parent directory for the root of the source tree:

```
shell$ mkdir build
shell$ cd build
shell$ 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:

```
shell$ make -j 8
...
```

> Note. 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.

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:

```
shell$ mkdir xcode
shell$ cmake -G Xcode ..
...
shell$ open AtomVM.xcodeproj
```

The above commands will build and open an AtomVM project in the Xcode IDE.

6.3.3 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:

```
shell$ ./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:

```
shell$ ./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`

6.4 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).

6.4.1 Build Requirements

The following software is required in order to build AtomVM for the ESP32 platform:

- Espressif Xtensa tool chains
- Espressif IDF SDK version 3.x (4.x support is currently *experimental*)
- `cmake`
- GNU `make`

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.

6.4.2 Build Instructions

Change directories to the `src/platforms/esp32` directory under the AtomVM source tree root:

```
shell$ cd <atomvm-source-tree-root>
shell$ cd src/platforms/esp32
```

Start by updating the configuration of local `sdkconfig` file via the `menuconfig` Make target:

```
shell$ make menuconfig
```

This command will bring up a curses dialog box. You can exit the program by typing `E`. Save the changes, and the program will exit.

You can now build AtomvVM using the `make` command:

```
shell$ make -j 8
...
```

> Note. You may specify `-j <n>`, where `<n>` is the number of CPUs you would like to assign to run the build in parallel.

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:

```
To flash all build output, run 'make flash' or:
python /path/to/esp-idf-sdk/components/esptool_py/esptool/esptool.py --chip esp32
--port /dev/ttyUSB0 --baud 115200 --before default_reset --after hard_reset
write_flash
-z --flash_mode dio --flash_freq 40m --flash_size detect
0x1000
/path-to-atomvm-source-tree/Atomvm/src/platforms/esp32/build/bootloader/bootloader.bin
0x10000
/path-to-atomvm-source-tree/Atomvm/src/platforms/esp32/build/atomvvm-esp32.bin
0x8000 /path-to-atomvm-source-tree/Atomvm/src/platforms/esp32/build/partitions.bin
```

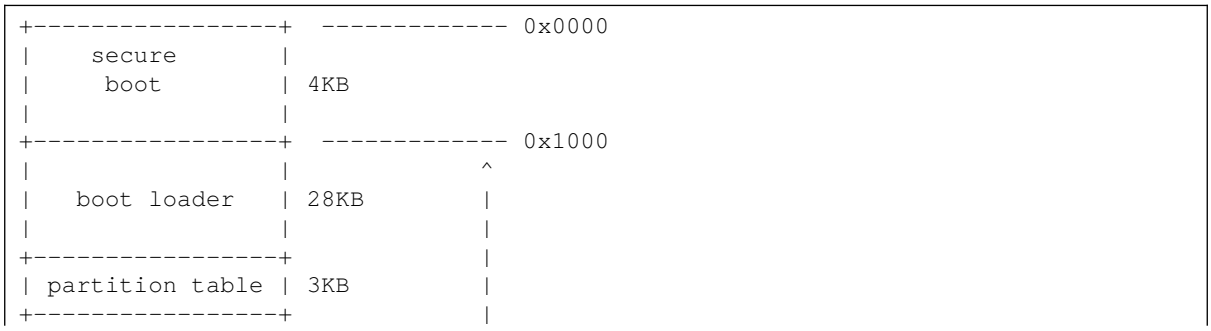
At this point, you can run `make flash` to upload the 3 binaries up to your ESP32 device, and in some development scenarios, this is a preferable shortcut.

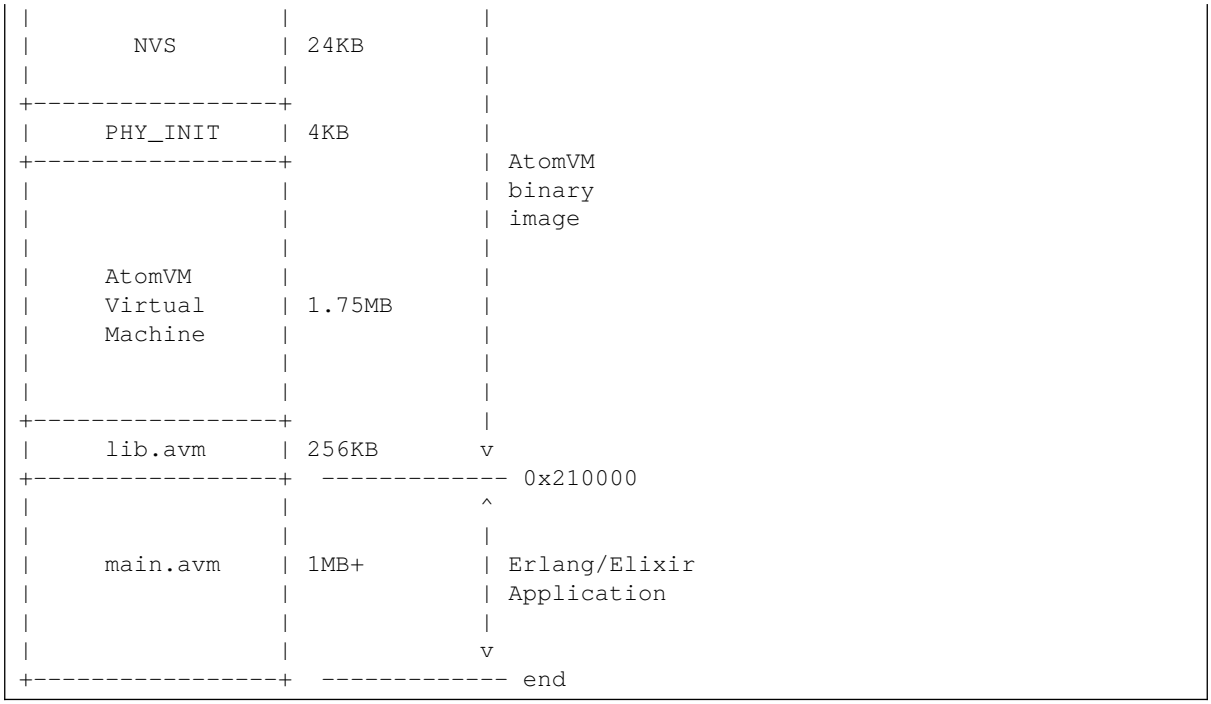
However, first, we will build a single binary image file containing all of the above 3 binaries, as well as the AtomVM core libraries. See Building a Release Image, below. But first, it is helpful to understand a bit about how the AtomVM partitioning scheme works, on the ESP32.

6.4.3 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/Elixire code.

The flash layout is roughly as follows (not to scale):





The following table summarizes the partitions created on the ESP32 when deploying AtomVM:

Partition	Offset	Length	Description
Secure Boot	0x00	4kB	Initialization vectors and other data needed for ESP32 secure boot.
Bootloader	0x1000	28kB	The ESP32 bootloader, as built from the IDF-SDK. AtomVM does not define its own bootloader.
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).
lib.avm	0x1D0000	256k	The AtomVM BEAM library, compiled from Erlang and Elixir files in the AtomVM source tree.
main.avm	0x210000	1mB	The user application. This is where users flash their compiled Erlang/Elixir code

6.4.4 The `lib.avm` and `main.avm` partitions

The `lib.avm` and `main.avm` partitions are intended to store Erlang/Elixir libraries (compiled down to BEAM files, and assembled as AVM files).

The `lib.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 `lib.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`, 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 `lib.avm`. Users should not have a need to override any functionality in the `lib.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 `lib.avm` partition.

Note. The location of the `main.avm` partition may change over time, depending on the relative sizes of the AtomVM binary and `lib.avm` partitions.

6.4.5 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.

Note. 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`.

The `mkimage.sh` script is run as follows:

```
shell$ ./tools/release/esp32/mkimage.sh
Writing output to /home/frege/AtomVM/src/platforms/esp32/build/atomvm-602e6bc.img
============================================
Wrote bootloader at offset 0x1000 (4096)
Wrote partition-table at offset 0x8000 (32768)
Wrote AtomvVM Virtual Machine at offset 0x10000 (65536)
Wrote AtomvVM Core BEAM Library at offset 0x110000 (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`,

But first, it is a good idea to erase the flash, e.g.,

```
shell$ 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...
```

You can then use the `./tools/dev/flash.sh` tool (or `esptool.py` directly, if you prefer), to flash the entire image to your device:

```
shell$ FLASH_OFFSET=0x1000 ./tools/dev/flash.sh
./src/platforms/esp32/build/atomvm-602e6bc.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...
```

Note. 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.

Finally, you can then flash your own application, e.g.,

```
shell$ ./tools/dev/flash.sh 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...
```

Note. 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.

6.4.6 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.

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 `comonents` directory of the AtomVM source tree;
2. Adding the component to the corresponding `main/component_nifs.txt` or `main/component_ports.txt` files;
3. Building the AtomVM binary.

Note. 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 your `<moniker>` to the `main/component_nifs.txt` file in the `src/platforms/esp32` directory.

Note. The `main/component_nifs.txt` file will not exist until after the first clean build.

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 your `<moniker>` to the `main/component_ports.txt` file in the `src/platforms/esp32` directory.

Note. The `main/component_ports.txt` file will not exist until after the first clean build.

6.5 Building for STM32

TODO

Chapter 7

AtomVM Internals

7.1 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 that 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.

7.2 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.

7.2.1 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.

> Note. 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). This set is the complement of the set of ready processes.
- `ready_processes` the subset of processes that are ready to run. This set is the complement of the set of waiting processes.

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` or the `ready_processes` field.

> Note. The C programming language treats structures in memory as contiguous sequences of fields of given types. Structures have no hidden pramble 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:

Error opening image file: [Errno 2] No such file or directory: '/Users/fadushin/work/src/github/atomvm/AtomVM/doc/src/_static/globalcontext-processes.svg'

> Note. The `Context` data structure is described in more detail below.

Module Management

An Aside: What's in a HashTable?

7.2.2 Modules

7.2.3 Contexts

7.3 Runtime Execution Loop

7.4 Module Loading

7.5 Function Calls and Return Values

7.6 Exception Handling

7.7 The Scheduler

Chapter 8

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.

8.1 The Context structure

8.1.1 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.

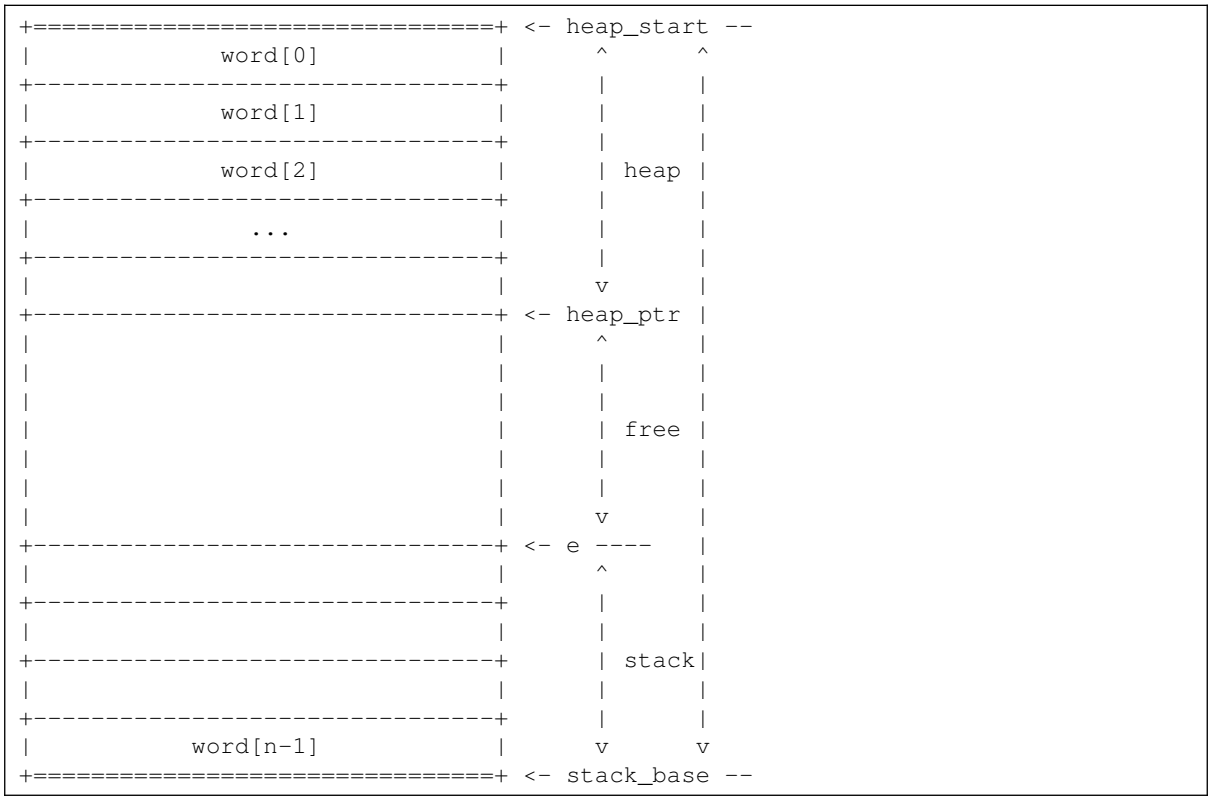
The heap grows incrementally, as memory is allocated, and terms are allocated sequentially, in increasing memory addresses. There is, therefore, no memory fragmentation, properly speaking, at least insofar as a portion of memory might be in use and then freed. However, it is possible that previously 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.

> Note. 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.

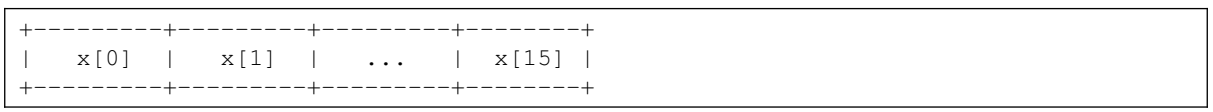
The following diagram illustrates an allocated block of memory that stores terms (or term pointers) in the heap and stack:



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 (typically by 2x) 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.

8.1.2 Registers

Registered are allocated in an array of 16 terms (words) and are referenced by the `x` field in the Context data structure:



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.

8.1.3 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)

8.1.4 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.

8.1.5 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, with the exception that messages in the process mailbox are not garbage collected, in the way that the process heap is. Instead, messages in the process mailbox are copied to the process heap when the message is read off the mailbox (e.g., via `receive ... end`). Messages (and their term contents) are then destroyed once they are no longer needed, and after being copied into the heap.

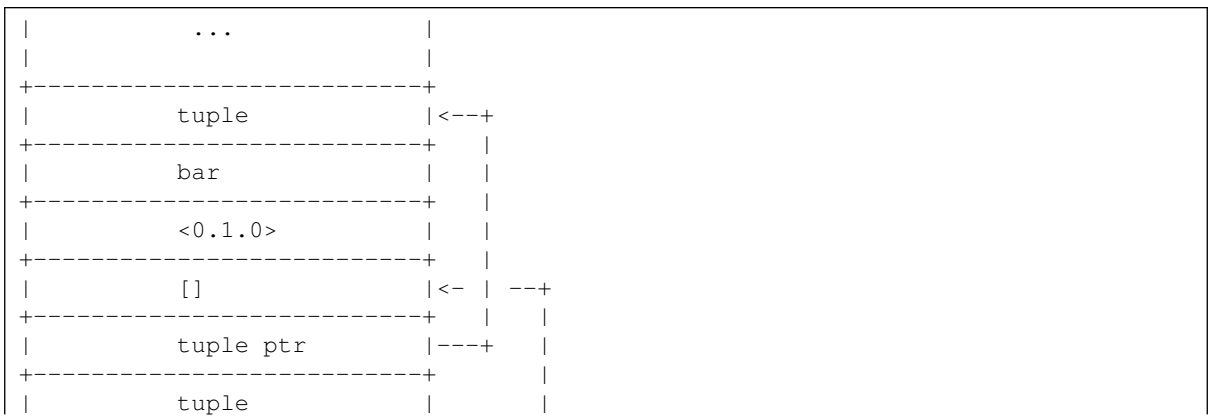
8.1.6 Memory Graph

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 beenn 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:



```
+---------------------------+       |
|          foo              |       |
+---------------------------+       |
|          list ptr         |-------+
+---------------------------+
|                           |
|           ...             |
012345678901234567890123456789012345678901234567890123456789
```

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 termill 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.

8.2 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.

8.2.1 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 ---------->|
```

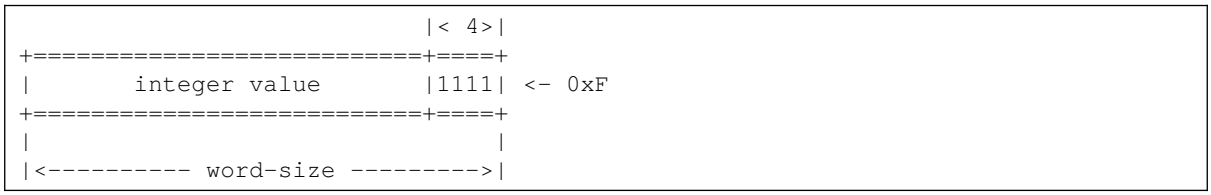
There may therefore only be `2^{word-size-6}` atoms in an AtomVM program (e.g., on a 32-bit platform, `67,108,864`). Plenty to work with!

> 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.

8.2.2 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:

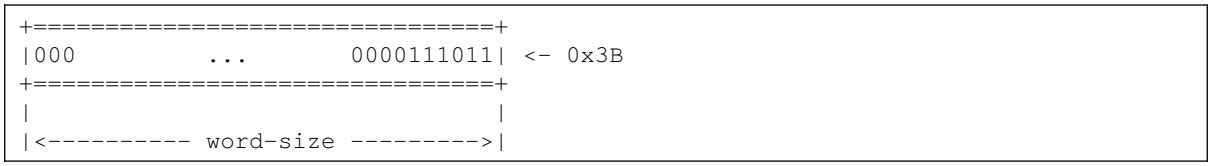


The magnitude of an integer is therefore limited to `2^{word-size - 4}` in an AtomVM program (e.g., on a 32-bit platform, `+- 134,217,728`).

Note. Arbitrarily large integers (bignums) are not currently supported in AtomVM.

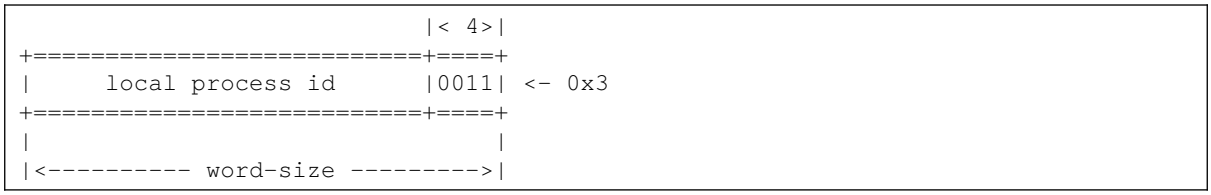
8.2.3 nil

The special value `nil` (typically the tail of the tail … of the tail of a list, or `[]`) is the special value 0x3B:



8.2.4 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:



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.

8.3 Boxed terms

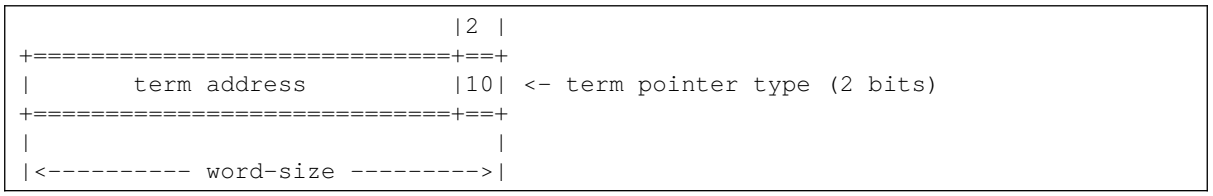
Some term types cannot fit in a single word, and must therefore used a sequence of contiguous words to represent the term contents. These terms are called "Boxed" terms. Boxed terms use the low-order 6 bits of the first byte (`boxed[0]`) to represent the term type, and the high order `word-size - 6` bits to represent the remaining size (in words) of the boxed term, not including the first word.

8.3.1 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.

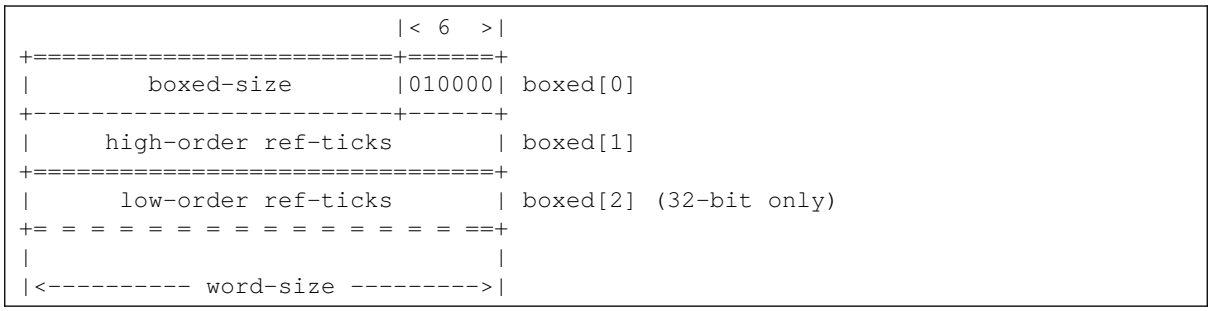
A boxed term pointer is a single-word term that contains the address of the referenced term in the high-order `word-size - 2` bits, and `0x2` (`10b`) in the low-order 2 bits.



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.

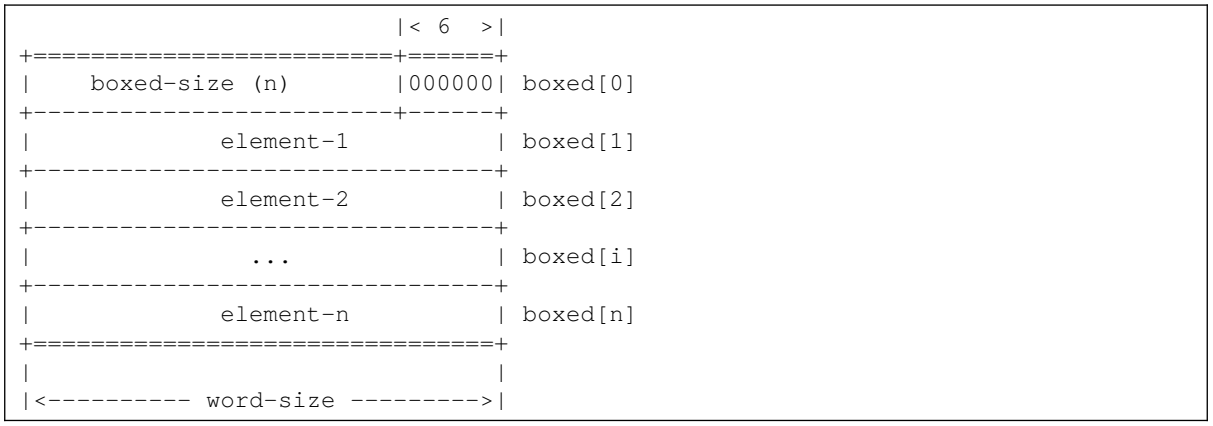
8.3.2 References

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]`:



8.3.3 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:



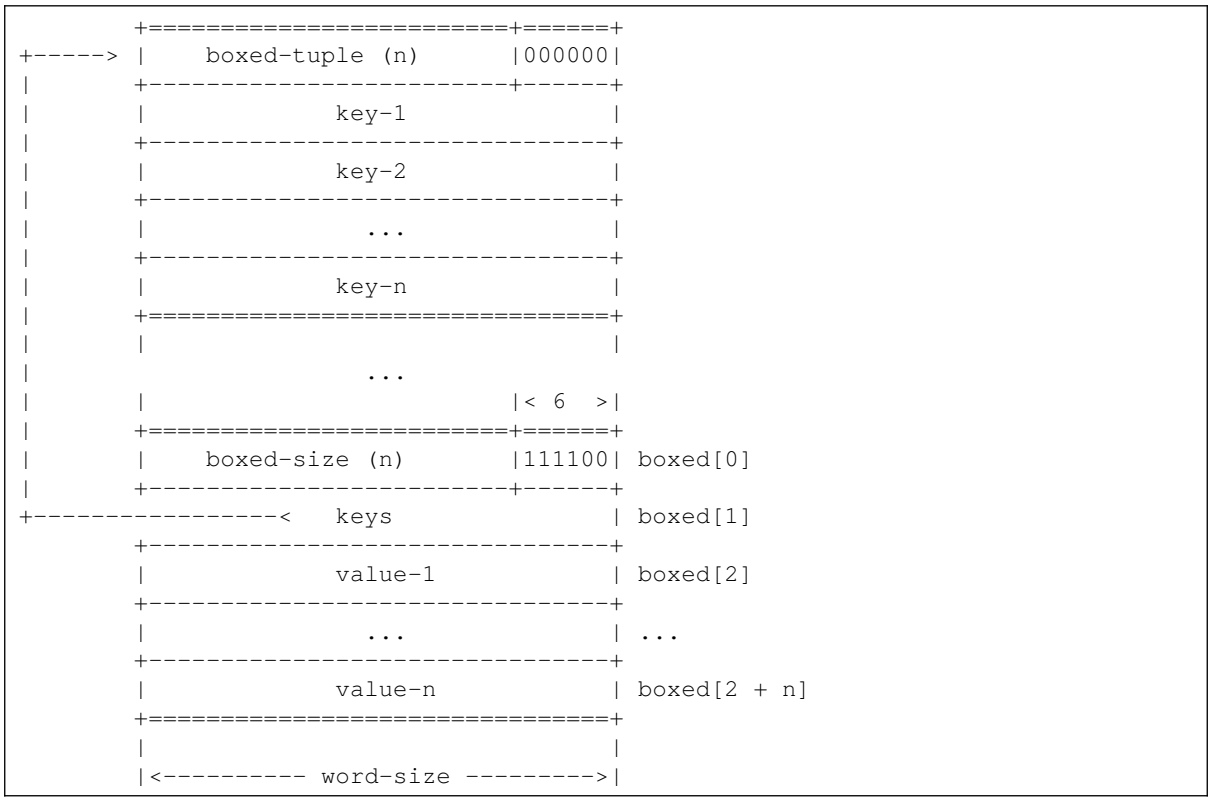
8.3.4 Maps

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.



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.

8.3.5 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.

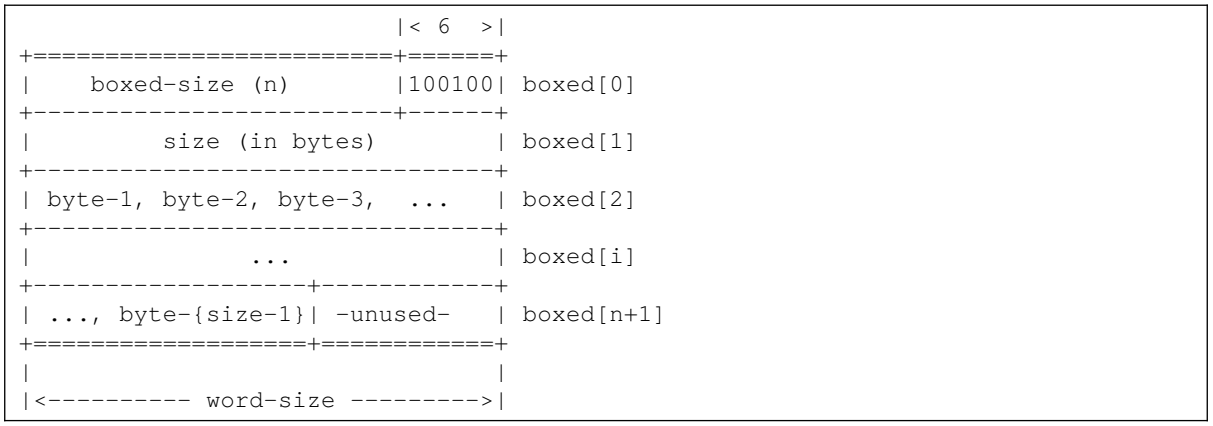
Finally, a special kind of binary is used in the heap to maintain the state of a match context, when, for example, matching binary terms using Erlang bit syntax. Like non-const binaries, creation and

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`):



> 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.

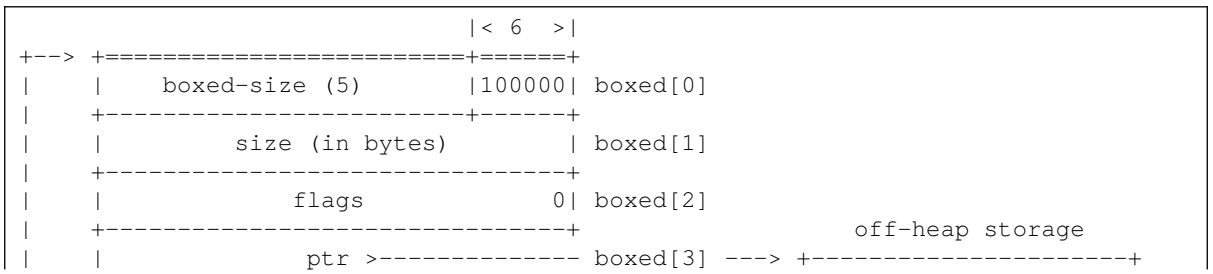
Currently, only the low-order bit of the flags field is used. A `0` value of indicates that the referenced binary is non-literal.

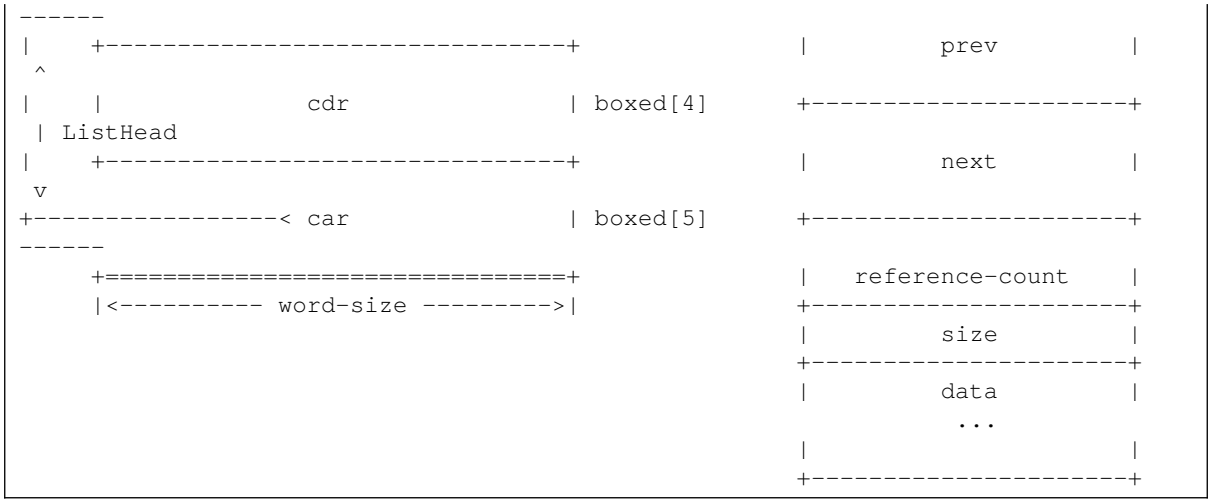
The off-heap data is a block of allocated data, containing:

- 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.





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`).

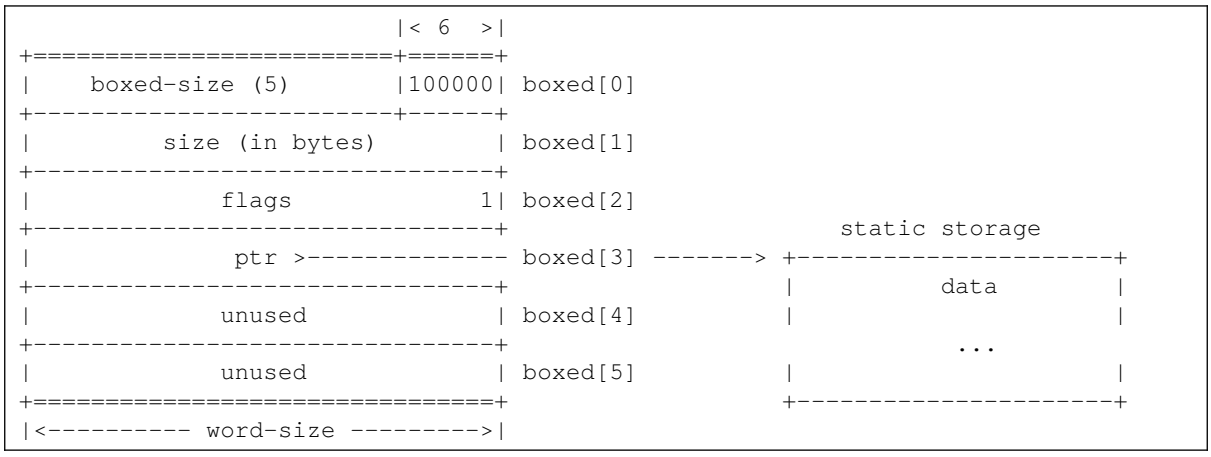
In addition, a reference-counted boxed term contains a cons-cell appended to the end of the boxed term, which is used by the garbage collector for tracking references. The `car` of this cell points to the boxed term, itself, and the `cdr` points to the "previous" cons cell associated with a reference 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 Garabage 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 uncesessary. These values are initialized to `nil`.

This heap structure has the following representation:



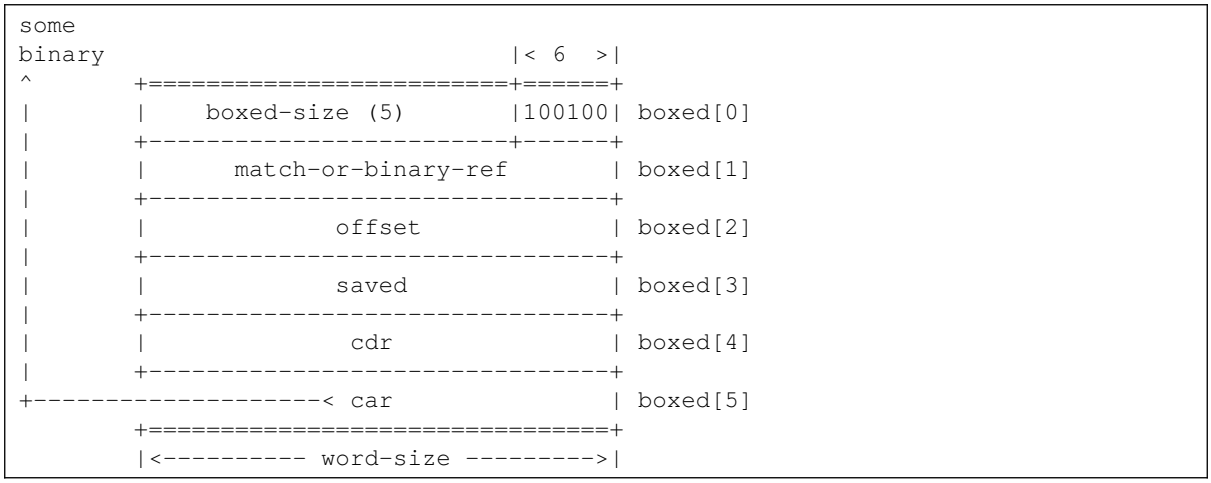
Match Binaries

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`.

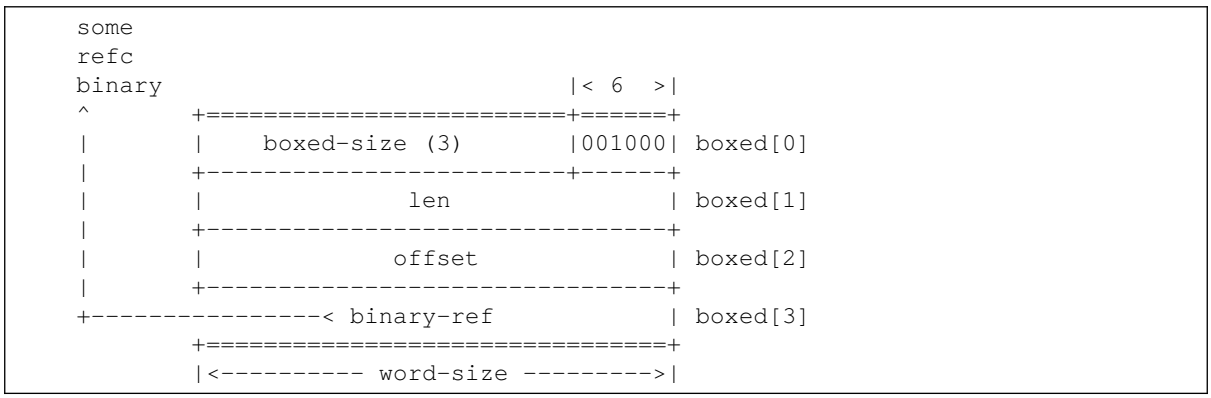


A reference to a reference-counted binary counts as a reference, in which case the creation or copying of a match binary results in the increment of the reference-counted binary's reference count, and the garbage collection of a match binary results in a decrement (and possible `free`ing) of a reference-counted 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.



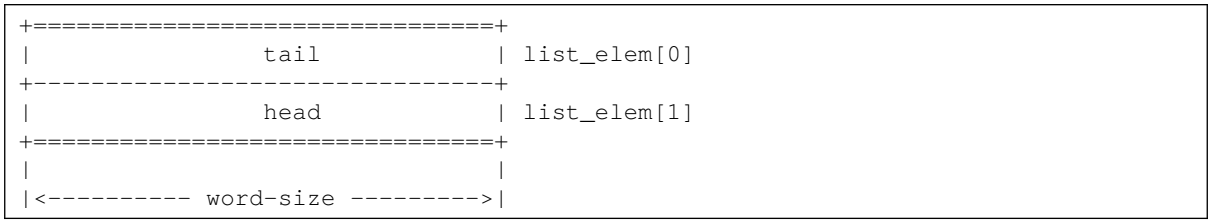
Note than when a sub-binary is copied between processes (e.g., via `erlang:send`, or `!`), the sub-binary boxed term, as well as the boxed-term that manages the reference-counted binary is copied, as well. Thus, sending a sub-binary to another process will result in an increment of the reference count 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 the `/binary` bit syntax specifier.

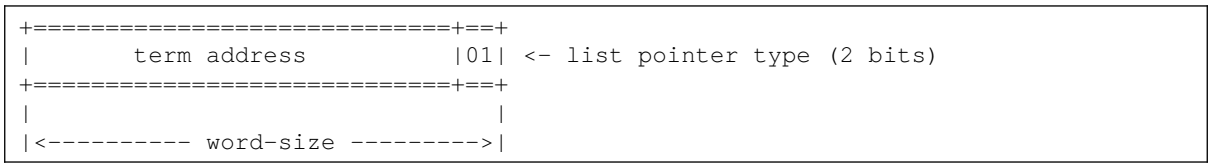
8.4 Lists

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.



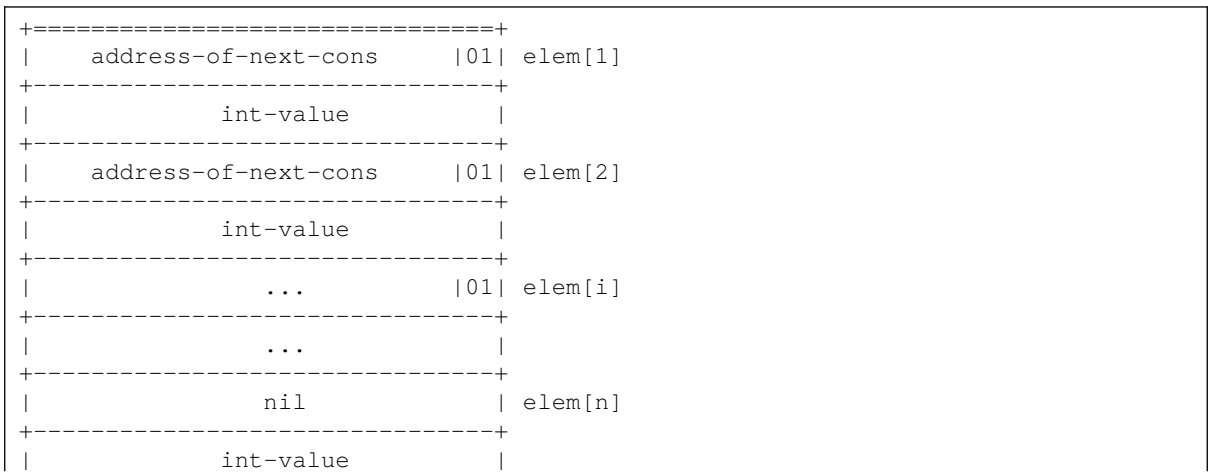
> Note. Lists are typically terminated with the empty list (`[]`), represented by the nil term, described above. However, nothing in Erlang requires that a sequence of cons cells is `nil`-terminated.

Unlike boxed terms, the low-order two bits of list pointers are `0x1` (`01b`):



8.4.1 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.

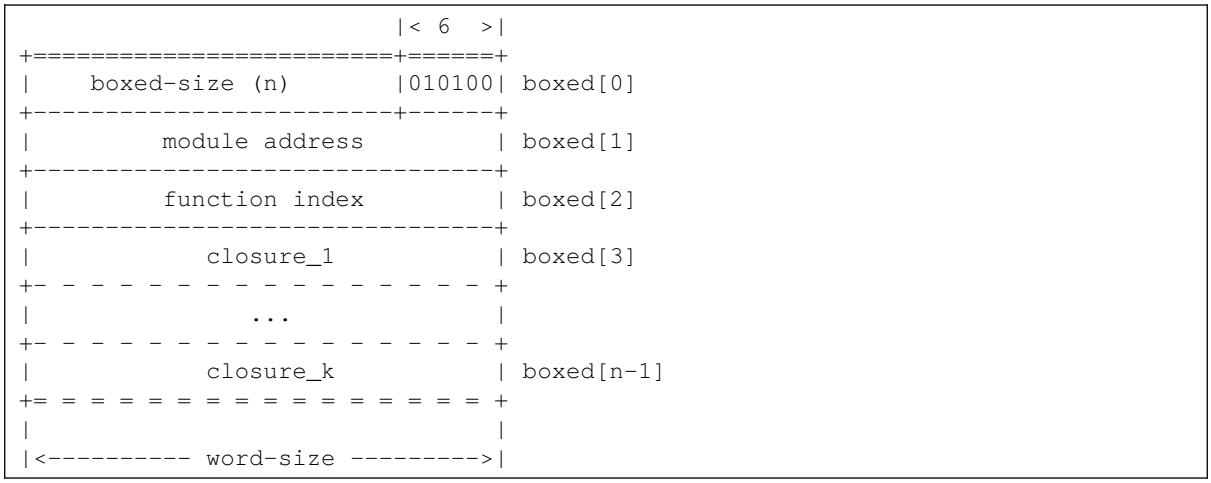


Note. String elements may not remain contiguous after a garbage collection event.

8.4.2 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



8.5 Special Stack Types

Some terms are only used in the stack.

8.5.1 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`):

8.5.2 Catch Labels

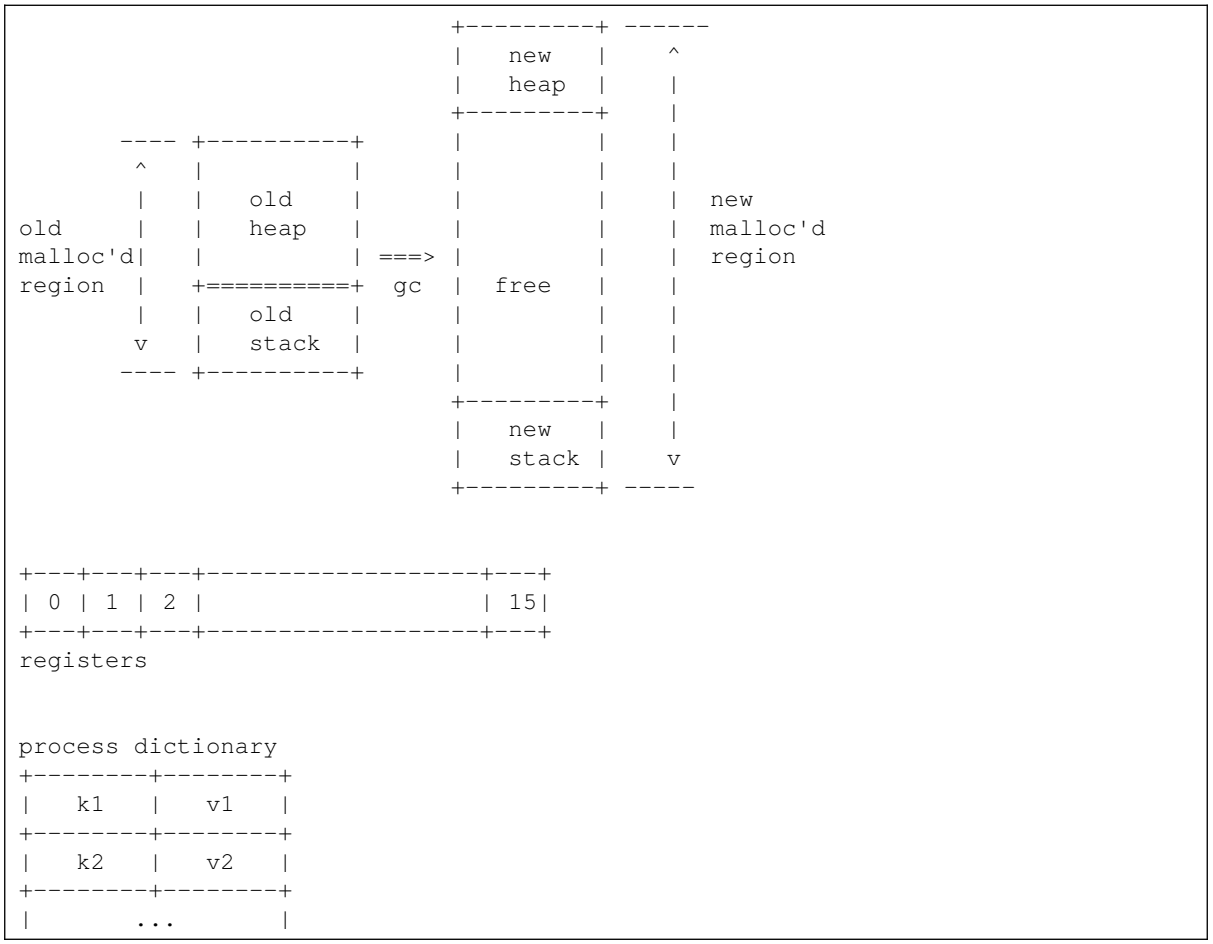
A catch label is used to indicate a position in code to which to jump in a try-catch expression. The term occupies a single term, with the low order 6 bits having the value `0x1B`, the high order 8 bits holding the module index (`m_i`), and the middle 18 bits holding the catch label index (`l_i`):

Module and catch label indices are stored outside of the process heap and are outside of the scope of this document.

8.6 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 (typically, enough to allocate a requested object, plus a little extra), 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.



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.

8.6.1 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).

> Note. Currently, AtomVM does not support symmetric multi-processing, or execution of multiple processes in parallel on separate machine cores.

8.6.2 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.)

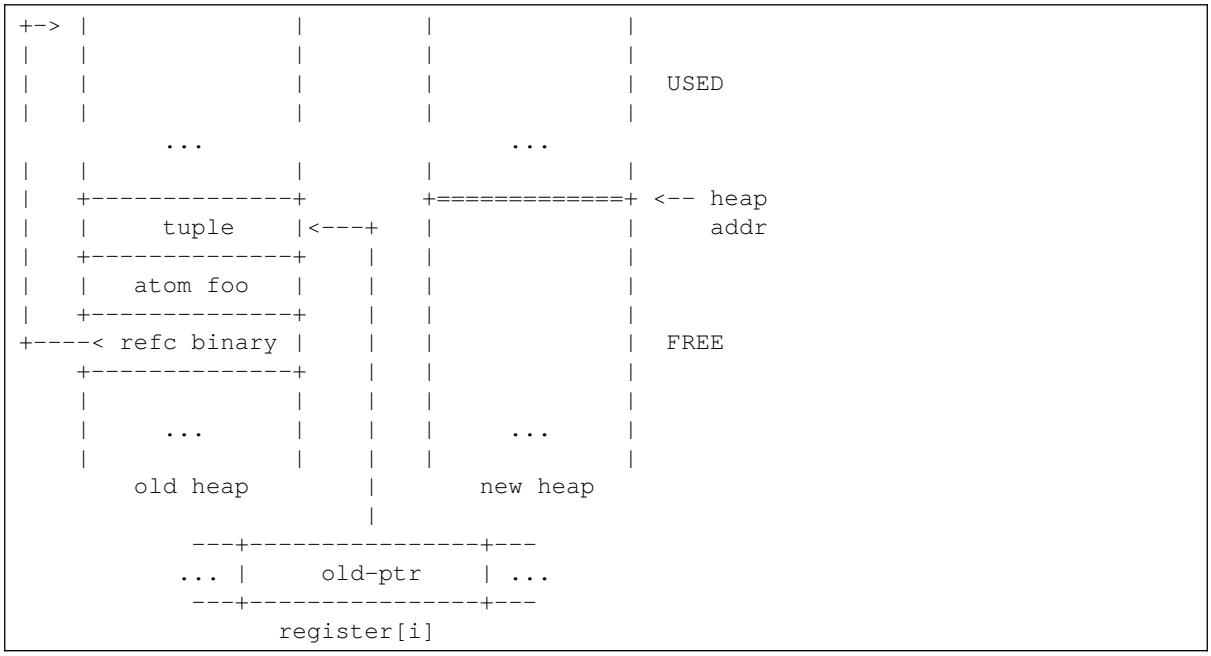
On the other hand, if the term in the root set points to a boxed term in the old heap, then three things happen:

- 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.

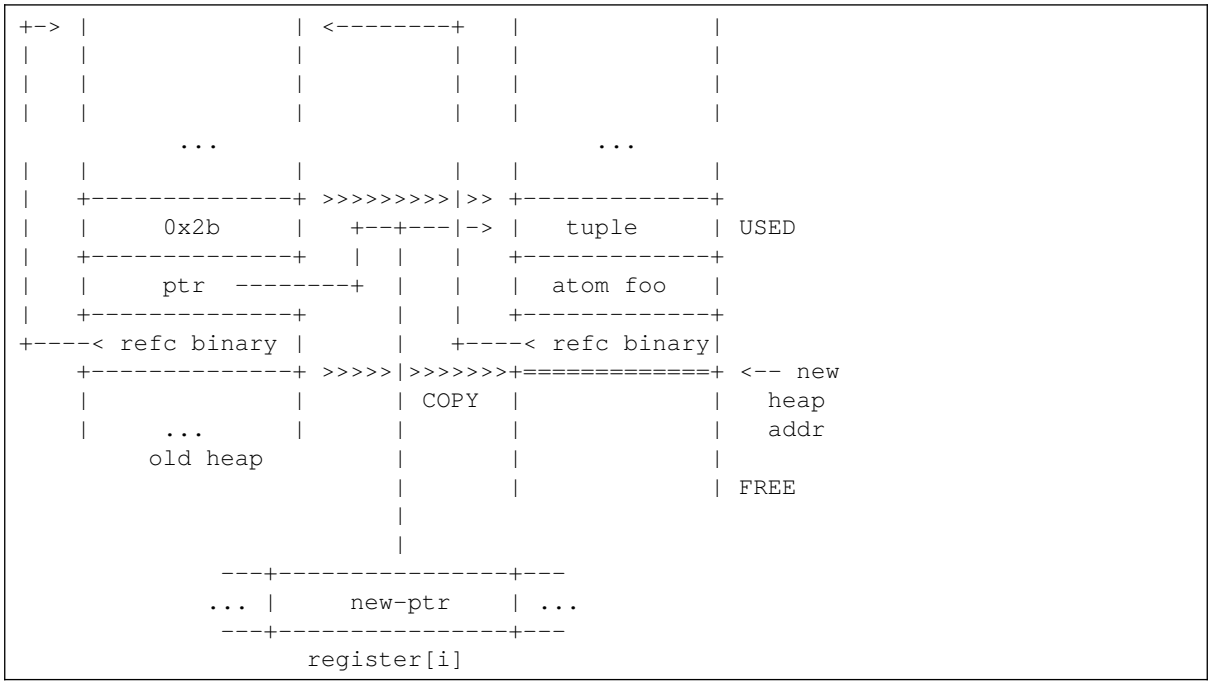
This process is best illustrated with a motivating example:

```
{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:



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:



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

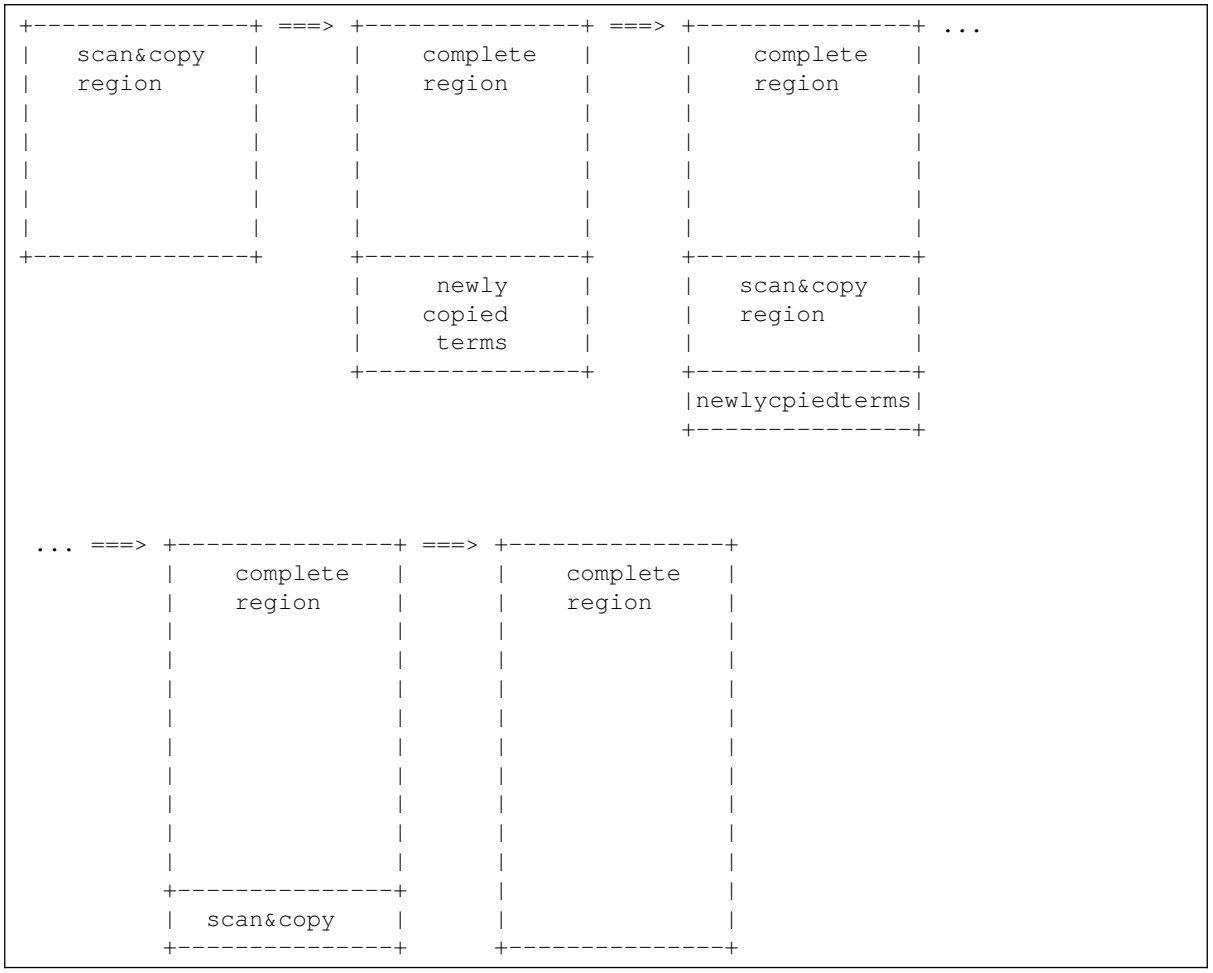
new heap consists of a set of contiguous copied boxed terms from the old heap, starting from the base address of the heap, to some higher address in the heap, but less than or equal to the maximum heap size on the new heap.

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 requion, but which in turn may create a new region of copied terms;
- The process is repeated until no new reqions have been introduced.

The following sequence of iterative additions to the new heap illustrates this process:



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,

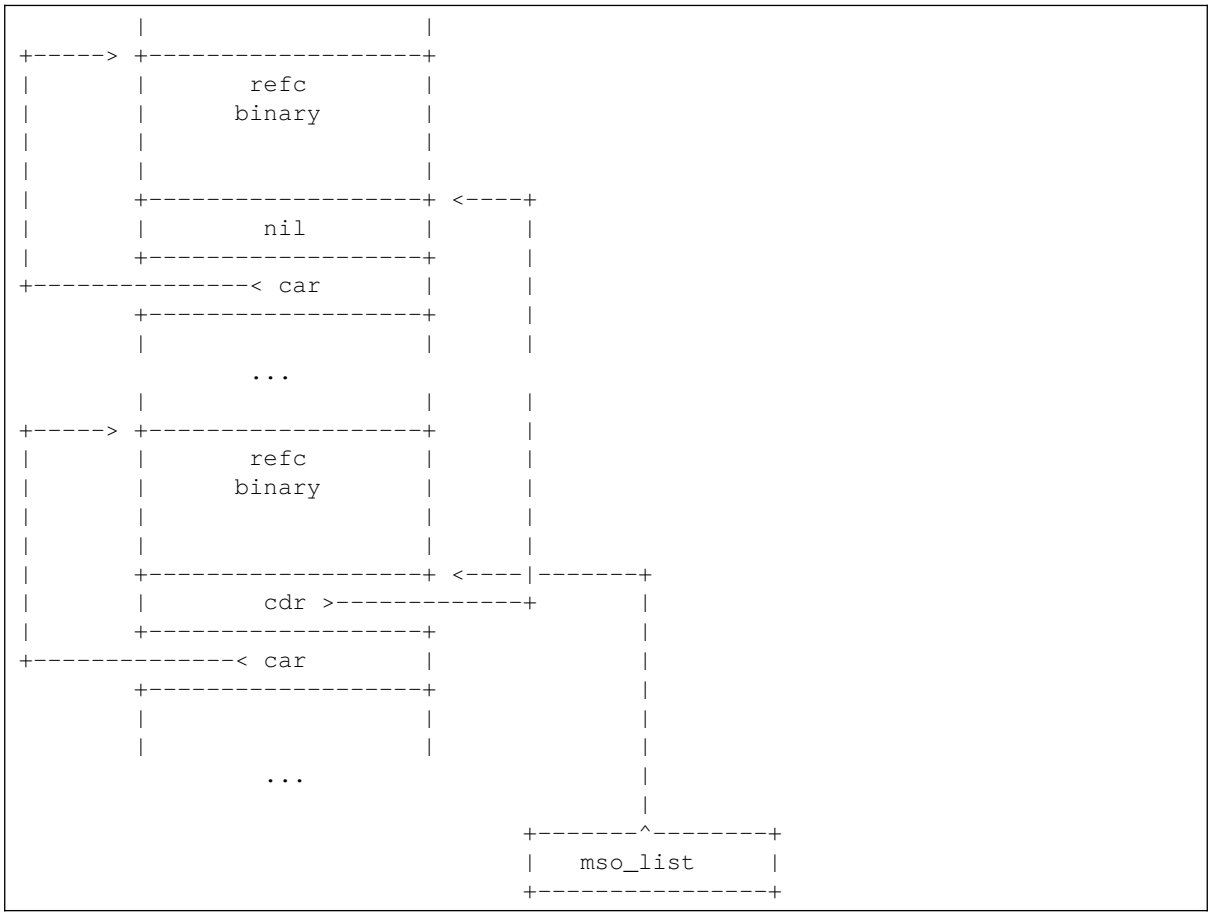
MSO Sweep

As mentioned in the section above on binaries, AtomVM supports refrence-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:



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 poont 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.

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.

9.1 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.

9.2 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.

9.3 File encodings

Each embedded file in a packbeam file contains a file header, followed by the file contents.

9.3.1 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.

9.3.2 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
```

9.3.3 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.

9.3.4 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.

> Note. 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.

9.3.5 end file

Packbeam files end with a special end header. The size field of the end header is 0 bytes.

Example

The following sequence of bytes encodes the end header:

```
00000000 00000000 00000000 656E6400
```

Chapter 10

API Reference Documentation

10.1 Erlang Libraries

- estdlib
- eavmlib
- alisp
- etest

10.2 AtomVMLib (C)

- libAtomVM

Contributing

Make sure to understand the license and the contribution guidelines before contributing and last but not least be kind.

11.1 Git Recommended Practices

- Commit messages should have a summary and a description
- Avoid trailing white spaces
- Always `git pull --rebase`
- Clean up your branch history with `git rebase -i`
- All your intermediate commits should build

11.2 Coding Style

11.2.1 C Code

Indentation

- K&R indentation and braces style
- Mandatory braces
- 4 spaces indentation

Good:

```
void f(int reverse)
{
    if (reverse) {
        puts("!dlroW olleH");
    } else {
        puts("Hello world");
    }
}
```

Bad:

```
void f(int reverse) {
    if (reverse)
        puts ("!dlroW olleH");
```

```
    else
        puts ("Hello world");
}
```

Names

- 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 function names (e.g. term_is_nil, term_is_integer, context_new, context_destroy)

Other Coding Conventions

- Pointer * 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.

11.2.2 Elixir Code

Just use Elixir formatter enforced style.

- genindex
- modindex
- search