

Emulashione

Emulashione is a multi-platform emulation framework focused on accuracy and portability.

- Core
 - System Definitions
 - Emulation Strategy

Core

Information on the core emulation library, which implements plugin management, system handling and synchronization of multiple devices.

System Definitions

A system's devices, busses, clock sources, and connections between all of these are defined in system definitions, human readable TOML structures.

Most items defined in the definition have an associated name, specified in the `name` key. This name is unique to that instance in that system, and is used to uniquely identify that particular bus, clock source, or device for purposes of connections. (This implies that each of these three types has a separate namespace.) This file can be roughly divided into four separate sections, each defined as a table in the file:

System Information

This table provides general metadata about the system under the `system` key. Data under this key is primarily used for presentation to the user. The following metadata values are defined:

- `name`: Name of the system, user visible.
- `manufacturer`: Manufacturer of the system, user visible.

Busses

Systems will contain one or more busses that connect devices together. These are abstract “pipes” for moving an (address, data) pair, each of which has a fixed width.

Mandatory for each bus is specifying the width of the data bus via the `dataWidth` property, and the width of the address bus via the `addrWidth` property.

```
[[busses]]
name = "68k"
addrWidth = 24
dataWidth = 16
pullup = 0xFFFF

[[busses.map]]
range = [ 0x000000, 0x3FFFFFF ]
target = "cartridge"

[[busses.map]]
range = [ 0xE00000, 0xFFFFFFF ]
```

```
target = "mainram"
```

Address Mapping

Address decoding of devices on the bus is handled internally by the emulation library. To enable this, the system definition defines a mapping of address ranges to which devices shall receive the bus transaction in this case. This is achieved through an array under the `map` key.

Each entry in the address map consists of two keys. First, `target` which indicates the name of the device that should be mapped at this address range. Next is `range`, which is an array of two unsigned integers corresponding to the [start, end] address range to connect this device to.

Devices are expected to implement any address wrap-around (as a result of incomplete decoding; e.g. 64K of RAM are mirrored through 2M of address space) instead of relying on the bus to handle it.

Pullup/Pulldown Support

It's common for peripherals to only drive part of a bus, leading to undefined contents on undriven bus lines. To guard against this, many systems have some sort of pullup or pulldown resistors on bus lines so they consistently read as a particular value if they're not being driven.

Specify either the `pulldown` or `pullup` key to indicate which bits “float” to 1 or 0, respectively, during a bus transaction if they are not being actively driven. It's not allowed to specify a bit to be both pulled up and down. (This makes no sense in the real world, either: the signal would likely end up being in the “indeterminate” zone between a 0 and 1.)

If these keys are omitted, or no pulls are specified for a particular bit, they will retain their most recently written value if not driven in a subsequent transaction.

Clock Sources

Every system should specify at least one clock source, from which models an oscillator in a real system. These clock sources are defined in an array stored under the `clocks` key in the definition. Common to all clock source types is the `name` key, which specifies their connection name.

A clock source may either be primary — with a directly specified frequency — or a derived clock, which applies a divisor to another clock to produce its output frequency. The system definition parser determines the clock type based on the presence of the `parent` key; if it's missing, the clock is assumed to be primary, otherwise derived.

Primary Clocks

For a primary clock, the only required key is `freqs`, which should be an array that defines the frequency “variants” for this clock source, of which there must always be at least one. Variants can be specified either as a plain frequency (double or integer is ok, in Hz) or as a table with the `name` and `freq` key, to allow a more user-friendly description of a frequency variant.

```
[[clocks]]
name = "MCLK"

[[clocks.freqs]]
name = "NTSC"
freq = 53693175
[[clocks.freqs]]
name = "PAL"
freq = 53203424
```

Derived Clocks

Most systems have more than one clock source. These clock sources are usually however all derived from one or more primary clock sources, usually through simple integer dividers. Derived clocks have a `parent` key, which indicates which clock source they use as their base; and a `divisor` key, which indicates a floating point number to apply as the frequency divisor. (The frequency of the parent is multiplied by 1/divisor)

```
[[clocks]]
name = "VCLK"
parent = "MCLK"
divisor = 7
```

Devices

Lastly, the system definition defines the actual devices operating in the system, which may be unique to the system and defined in external plugins, as an array under the `devices` key. Each entry in this array should be a table, with at least the `name` key.

The type of device to instantiate is defined by the contents of the `type` key. This is a comma separated list of short device names (thus these must be unique; there’s no requirement for the name format, but reverse DNS style works well) in descending preference order. This allows for more generic device types to be instantiated if the precise version of the device is not available.

```
[[devices]]
type = "psram,ram"
name = "mainram"
```

```
size = 65536
contents = "random"
access = 120

[[devices.connections]]
name = "busslot"
target = "68k"
type = "bus"
[[devices.connections]]
name = "clockslot"
target = "MCLK"
type = "clock"
```

Connections

Devices may be connected to busses, clock sources, or other devices. These connections are defined via the `connections` array key, which in turn contains tables with the following keys:

- `type`: Type of object being connected; may be one of `bus`, `clock`, or `device`.
- `name`: Name of the connection slot on the device being defined.
- `target`: Name of the device that will be connected to this slot.

Note that bus connections are implicitly made when a device is specified in a memory map, and are in most cases enough. Busses are usually only explicitly specified for devices that can perform bus mastering, such as processors and DMA controllers.

Additional Properties

Any keys not explicitly mentioned above (there is a list of ignored keys when constructing the arguments) are passed to the device constructor, and their interpretation is specific to the device class.

Schedulers

In addition to the primary objects that actually make up the system, there are also schedulers, which are responsible for emulating the system. They handle the execution of all devices that use the cooperative emulation method (effectively all processors) and ensuring they stay roughly in sync. This is accomplished by defining, on a per device basis, the maximum allowable runahead, and any dependencies between devices.

```
[[schedulers]]
name = "main"

[[schedulers.devices]]
```

```
target = "maincpu"
runahead = 1000
```

This consists of an array defining each scheduler. The required keys for the top level scheduler object are as follows:

- `name`: Unique identifier for this scheduler. This is used to establish relationships between schedulers.

Device Associations

A scheduler must have one or more devices associated to it to be useful; it is responsible for executing these devices. An array of devices associated with the scheduler are specified under the `devices` key; each entry should be a table with at least the following keys:

- `target`: Name of a device to execute on this scheduler

Constraints

The precise relationship between devices on the scheduler can be defined by several constraints. Simple constraints are available as just a single key:

- `runahead`: Maximum number of clock cycles the device may run ahead of all other devices on the scheduler. If there are per device synchronization requirements, this value acts as an upper cap on the runahead, and a default for all other devices.

Emulation Strategy

This page attempts to capture some of the ideas behind how the emulator is implemented, and how it in turn enables devices to be implemented to emulate actual hardware.

Devices can be emulated in one of two major ways: either by having the emulation core step them a bit at a time, and have the device indicate how much emulated time was consumed by the step – ideal for devices such as processors, with easily divisible operations; and by time slices, where the device runs continuously for some longer amount of time, which is ideal for devices with continuous outputs such as video and audio processors.

Stepping

Eventually, this will have more information.

Time Slices

Eventually, this will have more information.